Eclipse Cargo Tracker - Applied Domain-Driven Design Blueprints for Jakarta EE

Project Overview:

The Eclipse Cargo Tracker application is a reference implementation of a Domain-Driven Design (DDD) shipping logistics system. It demonstrates the implementation of a real-world cargo tracking system using Jakarta EE technologies. The application allows shipping companies to book, route, track, and handle cargo throughout its journey from origin to destination.

The system is designed to showcase how DDD principles can be applied to create a maintainable, scalable, and business-focused application. It provides functionality for booking new cargo, selecting routes, registering handling events, and tracking cargo in real-time.

Date of creation: 2025-04-06

Author: Automatically generated by QuantalQ

Project Version: fb3cbe1

Link to Project: https://github.com/eclipse-ee4j/cargotracker

Contents

Eclipse Cargo Tracker - Applied Domain-Driven Design Blueprints for Jakarta EE

Introduction

Overview

The Eclipse Cargo Tracker application is a reference implementation of a Domain-Driven Design (DDD) shipping logistics system. It demonstrates the implementation of a real-world cargo tracking system using Jakarta EE technologies. The application allows shipping companies to book, route, track, and handle cargo throughout its journey from origin to destination.

The system is designed to showcase how DDD principles can be applied to create a maintainable, scalable, and business-focused application. It provides functionality for booking new cargo, selecting routes, registering handling events, and tracking cargo in real-time.

Purpose of the Software

Eclipse Cargo Tracker serves as a practical example of applying Domain-Driven Design principles to a complex business domain. It aims to:

- Demonstrate the implementation of DDD patterns and practices in a Jakarta EE environment
- Provide a reference architecture for enterprise applications
- Showcase the integration of various Jakarta EE technologies
- Serve as an educational tool for developers learning DDD and Jakarta EE

The application models the core business processes of cargo shipping and tracking, focusing on the essential domain concepts while abstracting away non-essential details.

Scope

The Cargo Tracker application covers the following key areas:

- Cargo booking and routing
- Handling event registration through multiple interfaces (web, mobile, REST, file)
- Cargo tracking and status monitoring
- Real-time updates of cargo status

The application does not include:

- Billing and payment processing
- Customer management
- Vessel management beyond basic voyage scheduling
- Crew management
- Detailed logistics planning

Target Audience

This documentation is intended for:

- Software architects who want to understand the overall structure and design of the application
- Software engineers who need to use, modify, or extend the system
- Technical leads evaluating DDD and Jakarta EE for enterprise applications
- Developers interested in learning about DDD implementation patterns
- System integrators who need to connect external systems with the Cargo Tracker application
- Product owners or product managers who would like to understand the product on a more technical level

Glossary

Term/Acronym	Full Form	Description
Aggregate	Aggregate Pattern	A cluster of domain objects that can be treated as a single unit, with the aggregate root being the entity through which all access occurs
API	Application Programming Interface	A set of rules and protocols for building and interacting with software applications
Arquillian	-	A testing platform that allows for in-container testing of Java EE applications
CDI	Contexts and Dependency Injection	A Jakarta EE technology for dependency injection and contextual lifecycle management
DDD	Domain-Driven Design	A software development approach focusing on modeling software to match a domain according to input from domain experts
DTO	Data Transfer Object	An object that carries data between processes or layers in an application
EJB	Enterprise JavaBeans	A server-side component architecture for modular construction of enterprise applications
Entity	Entity Pattern	An object defined primarily by its identity rather than its attributes
JAX-RS	Jakarta RESTful Web Services	A Jakarta EE API for creating RESTful web services
JMS	Jakarta Messaging	A Java API that allows applications to create, send, receive, and read messages in a distributed system
JPA	Jakarta Persistence API	A Java specification for managing relational data in Java applications
JSF	JavaServer Faces	A Jakarta EE technology for building component-based user interfaces for web applications
JUnit	Java Unit Testing Framework	A popular testing framework for Java applications
MDB	Message-Driven Bean	An enterprise bean that allows Jakarta EE applications to process messages asynchronously
Repository	Repository Pattern	A mechanism for encapsulating storage, retrieval, and search behavior
REST	Representational State Transfer	An architectural style for distributed hypermedia systems, commonly used for web services
ShrinkWrap	-	A Java API for creating archives like JAR, WAR, and EAR without using file system
Specification	Specification Pattern	A pattern that separates the statement of how to match a candidate from the candidate object

Term/Acronym	Full Form	Description
SSE	Server-Sent Events	A server push technology enabling a client to receive automatic updates from a server via HTTP connection
UN/LOCODE	United Nations Code for Trade and Transport Locations	A geographic coding scheme developed and maintained by the United Nations to identify locations related to international trade
Value Object	Value Object Pattern	An immutable object that is distinguishable only by the state of its properties

For a comprehensive glossary of all terms used across the Cargo Tracker documentation, see Consolidated Glossary.

Getting Started

Installation Guide

To install and run the Cargo Tracker application, follow these steps:

Prerequisites Before installing the Cargo Tracker application, ensure you have the following prerequisites:

- Java Development Kit (JDK) 11 or later
- Maven 3.6.0 or later
- Git client
- Docker (optional, for containerized deployment)

Installation Steps

1. **Clone the repository**:

   [] git clone https://github.com/eclipse-ee4j/cargotracker.git cd cargotracker

2. **Build the application**:

   [] mvn clean package

3. **Deploy to a Jakarta EE server**: You can deploy the application to any Jakarta EE 9+ compatible application server. The project includes configuration for Liberty:

   [] mvn liberty:run

4. **Docker deployment** (optional): The project includes a Dockerfile for containerized deployment:

   [] docker build -t cargo-tracker . docker run -p 9080:9080 -p 9443:9443 cargo-tracker

5. **Access the application**: Once deployed, the application can be accessed at:

   `http://localhost:9080/cargo-tracker/`

System Requirements

Minimum Requirements

- **Processor**: Dual-core CPU, 2 GHz or faster
- **Memory**: 4 GB RAM
- **Disk Space**: 1 GB available space
- **Operating System**: Any OS that supports Java (Windows, macOS, Linux)
- **Java**: JDK 11 or later
- **Database**: Any database supported by JPA (H2 included for development)

Recommended Requirements

- **Processor**: Quad-core CPU, 2.5 GHz or faster
- **Memory**: 8 GB RAM
- **Disk Space**: 2 GB available space
- **Operating System**: Linux (preferred for production)
- **Java**: JDK 17 or later
- **Database**: PostgreSQL or Oracle for production use

Quick Start / First Run

After installing the application, follow these steps to get started:

1. **Access the application** at http://localhost:9080/cargo-tracker/
2. **Explore the public tracking interface**:
   - Navigate to the "Public Tracking" section
   - Enter one of the sample tracking IDs (e.g., ABC123)
   - View the cargo's current status and history
3. **Explore the admin interface**:
   - Navigate to the "Administration" section
   - View the dashboard showing all cargo
   - Try booking a new cargo:
     - Click "Book" in the navigation menu
     - Enter origin, destination, and arrival deadline
     - Submit the booking form
     - Note the tracking ID for future reference
4. **Register a handling event**:
   - Navigate to the "Event Logger" section
   - Select the cargo you just booked
   - Choose an event type (e.g., RECEIVE)
   - Select a location
   - Set the completion time
   - Submit the event
5. **Track the cargo again** to see the updated status based on the handling event you registered

Prerequisites

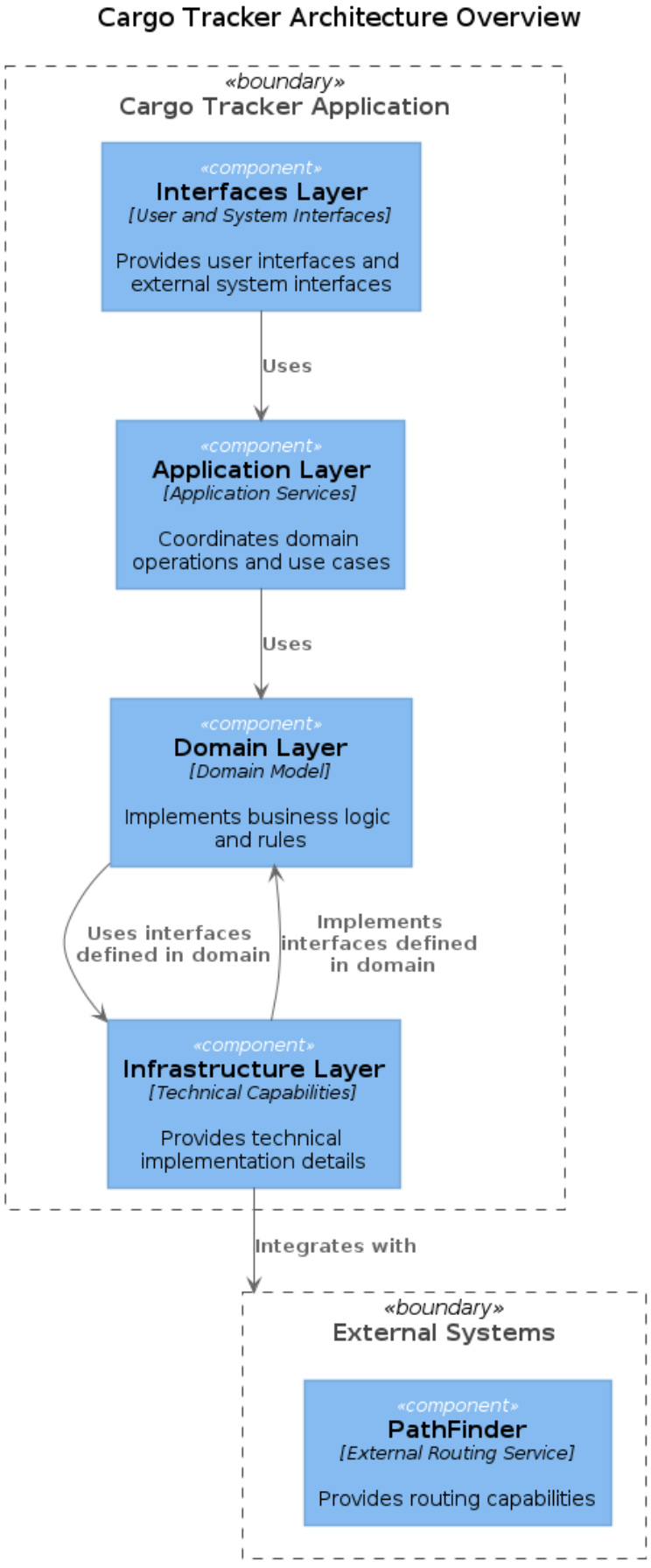
Before using the Cargo Tracker application, you should have:

- Basic understanding of shipping and logistics concepts
- Familiarity with web applications
- Knowledge of Jakarta EE technologies (for developers)
- Understanding of Domain-Driven Design principles (for developers)

Architecture and Design Overview

System Architecture

The Cargo Tracker application follows a layered architecture based on Domain-Driven Design principles, with a clear separation of concerns between different modules. The system is organized into four main layers:


Cargo Tracker Architecture Overview
«boundary»
Cargo Tracker Application
«component»
Interfaces Layer
[User and System Interfaces]
Provides user interfaces and external system interfaces
Uses
«component»
Application Layer
[Application Services]
Coordinates domain operations and use cases
Uses
«component»
Domain Layer
[Domain Model]
Implements business logic and rules
Uses interfaces defined in domain
Implements interfaces defined in domain
«component»
Infrastructure Layer
[Technical Capabilities]
Provides technical implementation details
Integrates with
«boundary»
External Systems
«component»
PathFinder
[External Routing Service]
Provides routing capabilities

For more detailed information about the architecture, refer to the High-Level Documentation.

Domain Layer The domain layer is the core of the application, containing the business logic and rules of the cargo shipping domain. It is organized into four main aggregates:

- **Cargo Aggregate**: Represents shipping cargo and its routing information
- **Handling Aggregate**: Represents cargo handling events at various locations
- **Location Aggregate**: Represents shipping locations using UN/LOCODE
- **Voyage Aggregate**: Represents vessel voyages between locations

The domain layer also includes domain services and shared utilities:

- **Routing Service**: Provides routing capabilities for cargo
- **Specification Pattern**: Implements business rules as specifications

Application Layer The application layer serves as an intermediary between the domain model and external interfaces. It handles use cases and coordinates domain operations through services:

- **Booking Service**: Handles cargo booking and routing operations
- **Handling Event Service**: Processes handling events for cargo
- **Cargo Inspection Service**: Inspects cargo status based on handling events

The application layer implements the following key responsibilities: - Coordinating the execution of domain operations - Managing transactions - Handling application-specific validation - Coordinating with infrastructure services - Publishing domain events

Infrastructure Layer The infrastructure layer provides the technical capabilities and implementations that support the domain model:

- **Events**: CDI-based event infrastructure for domain events
- **Logging**: Logging capabilities through CDI producer methods
- **Messaging**: Asynchronous messaging using JMS for domain events
- **Persistence**: Repository implementations using JPA for data persistence
- **Routing**: Adapter to an external routing service (PathFinder)

The infrastructure layer implements the interfaces defined in the domain layer, providing concrete implementations for: - Repository interfaces - Domain services that require external resources - Technical services like messaging and logging

Interfaces Layer The interfaces layer provides user-facing and system-facing interfaces for the application:

- **Booking Interfaces**: Web interfaces and facades for booking and routing cargo
- **Handling Interfaces**: File processing, mobile interfaces, and REST APIs for handling events
- **Tracking Interfaces**: Web interfaces for tracking cargo, including real-time updates via SSE

The interfaces layer is responsible for: - Transforming between domain objects and presentation models (DTOs) - Handling user input and validation - Rendering views and responses - Integrating with external systems

User Guide

Features Overview

The Cargo Tracker application provides several key features for managing and tracking cargo:

Cargo Booking The cargo booking feature allows shipping clerks to register new cargo for transport. It captures essential information such as:

- Origin location (using UN/LOCODE)
- Destination location (using UN/LOCODE)
- Arrival deadline

Upon successful booking, the system generates a unique tracking ID that can be used to track the cargo throughout its journey.

Cargo Routing Once a cargo is booked, it needs to be assigned a route. The routing feature allows shipping clerks to:

- View cargos that need routing
- Request possible routes for a cargo
- Select and assign the most appropriate route
- View the legs of the selected route (voyage, from location, to location, loading time, unloading time)

Cargo Tracking The tracking feature allows customers and shipping clerks to track the progress of cargo:

- Public tracking interface for customers
- Administrative tracking interface with more details for shipping clerks
- Real-time tracking updates via Server-Sent Events
- Map visualization of cargo location
- Status information (Not Received, In Port, Onboard Carrier, Claimed, Unknown)
- Handling history

Handling Event Registration The application provides multiple interfaces for registering handling events:

- Mobile web interface for cargo handlers
- REST API for external systems
- File upload for batch processing

Handling events include:

- RECEIVE (receiving cargo at a location)
- LOAD (loading cargo onto a voyage)
- UNLOAD (unloading cargo from a voyage)
- CUSTOMS (customs clearance)
- CLAIM (customer claiming cargo at destination)

Cargo Management The application provides features for managing cargo after it has been booked:

- Change destination
- Change arrival deadline
- View cargo details
- List all cargo with different filtering options (routed, not routed, claimed)

User Interface Guide

The Cargo Tracker application provides several user interfaces for different user roles:

Public Tracking Interface The public tracking interface is accessible to customers and allows them to track their cargo using a tracking ID.

Key elements:

- Tracking ID input field
- Track button
- Cargo status display
- Map visualization

Administrative Dashboard The administrative dashboard provides an overview of all cargo in the system and is intended for shipping company staff.

Key elements:

Figure 1: Public Tracking Interface

- Navigation menu (Dashboard, Book, Track)
- Cargo lists (Not Routed, Routed, Claimed)
- Action buttons (Route, Details, Change Destination, Change Deadline)

Booking Interface The booking interface allows shipping clerks to register new cargo.

Key elements:

- Origin location dropdown
- Destination location dropdown
- Arrival deadline date picker
- Book button

Event Logger The event logger interface allows cargo handlers to register handling events.

Key elements:

- Cargo selection dropdown
- Event type selection
- Location selection
- Voyage selection (for LOAD and UNLOAD events)
- Completion time input
- Submit button

Step-by-Step Tutorials

How to Book New Cargo

1. Navigate to the "Administration" section
2. Click "Book" in the navigation menu
3. Select the origin location from the dropdown
4. Select the destination location from the dropdown
5. Choose the arrival deadline using the date picker
6. Click "Book" to submit the booking
7. Note the tracking ID displayed on the confirmation page

How to Route Cargo

1. Navigate to the "Administration" section
2. Click "Dashboard" in the navigation menu
3. Find the cargo you want to route in the "Not Routed" list
4. Click the "Route" button for that cargo

5. The system will display possible routes
6. Review the routes and select the most appropriate one
7. Click “Assign” to assign the selected route to the cargo

How to Register a Handling Event

1. Navigate to the “Event Logger” section
2. Select the cargo from the dropdown
3. Choose the event type (RECEIVE, LOAD, UNLOAD, CUSTOMS, CLAIM)
4. Select the location
5. If the event type is LOAD or UNLOAD, select the voyage
6. Set the completion time
7. Click “Submit” to register the event

How to Track Cargo

1. Navigate to the “Public Tracking” section
2. Enter the tracking ID in the input field
3. Click “Track”
4. View the cargo’s current status, location, and handling history
5. Click “Show on Map” to see the cargo’s location on a map

Usage Scenarios / Use Cases

Scenario 1: Booking and Routing Cargo A shipping clerk needs to book a new cargo shipment from Hong Kong to Stockholm with a deadline of 3 months from now.

1. The clerk accesses the booking interface
2. Selects “CNHKG” (Hong Kong) as the origin
3. Selects “SESTO” (Stockholm) as the destination
4. Sets the arrival deadline to 3 months from today
5. Submits the booking
6. The system generates a tracking ID
7. The clerk accesses the dashboard and finds the new cargo in the “Not Routed” list
8. The clerk clicks “Route” to request possible routes
9. The system displays several route options with different voyages
10. The clerk selects the route that best meets the customer’s needs
11. The system assigns the route to the cargo and updates its status to “Routed”

Scenario 2: Tracking Cargo Through Its Journey A customer wants to track their cargo from Hong Kong to Stockholm.

1. The customer accesses the public tracking interface
2. Enters their tracking ID
3. The system displays the cargo’s current status, location, and handling history
4. As the cargo progresses through its journey, handling events are registered:
   - RECEIVE at Hong Kong
   - LOAD onto voyage V100 at Hong Kong
   - UNLOAD from voyage V100 at Hamburg
   - LOAD onto voyage V200 at Hamburg
   - UNLOAD from voyage V200 at Stockholm
   - CLAIM at Stockholm
5. Each time the customer checks the tracking, they see the updated status and location

Scenario 3: Changing Cargo Destination A shipping clerk needs to change the destination of a cargo from Stockholm to Helsinki due to a customer request.

1. The clerk accesses the dashboard

2. Finds the cargo in the list
3. Clicks “Change Destination”
4. Selects “FIHEL” (Helsinki) as the new destination
5. Submits the change
6. The system updates the cargo’s route specification with the new destination
7. If the cargo was already routed, it is marked as misrouted
8. The clerk can then route the cargo again to find a suitable route to Helsinki

Troubleshooting Common Issues

Issue: Cargo Not Found When Tracking Possible causes:

- Incorrect tracking ID entered
- Cargo not yet registered in the system

Solution:

- Verify the tracking ID is correct
- Check with the shipping clerk if the cargo has been booked

Issue: No Routes Available for Cargo Possible causes:

- No voyages available between the origin and destination
- Arrival deadline too soon to accommodate available voyages

Solution:

- Try extending the arrival deadline
- Check if alternative destinations are acceptable

Issue: Handling Event Registration Fails Possible causes:

- Invalid cargo tracking ID
- Invalid location code
- Invalid voyage number
- Incompatible event type and voyage (e.g., RECEIVE with a voyage)

Solution:

- Verify all input data is correct
- Ensure the event type is compatible with the provided information
- Check that the cargo exists in the system

Issue: Real-time Tracking Updates Not Working Possible causes:

- Browser does not support Server-Sent Events
- Network issues preventing SSE connection

Solution:

- Try using a modern browser (Chrome, Firefox, Safari, Edge)
- Check network connectivity
- Refresh the page to reestablish the SSE connection

Developer Guide

Codebase Overview

The Cargo Tracker codebase is organized according to Domain-Driven Design principles, with a clear separation between layers:

```
src/main/java/org/eclipse/cargotracker/
 application/           # Application services
    internal/          # Service implementations
    util/              # Application utilities
 domain/                # Domain model
    model/             # Domain entities and value objects
       cargo/         # Cargo aggregate
       handling/      # Handling aggregate
       location/      # Location aggregate
       voyage/        # Voyage aggregate
    service/           # Domain services
    shared/            # Shared domain utilities
 infrastructure/        # Infrastructure implementations
    events/            # Event infrastructure
    logging/           # Logging infrastructure
    messaging/         # Messaging infrastructure
    persistence/       # Persistence infrastructure
    routing/           # Routing infrastructure
 interfaces/            # User and system interfaces
     booking/           # Booking interfaces
        facade/        # Booking facade
        sse/           # Server-sent events
        web/           # Web interface
     handling/          # Handling interfaces
        file/          # File upload
        mobile/        # Mobile interface
        rest/          # REST API
     tracking/          # Tracking interfaces
         web/           # Web interface
```

The codebase follows a package-by-feature structure within each layer, making it easy to locate and understand the components related to each feature.

Folder Structure & Key Components

Application Layer

- **ApplicationEvents.java**: Interface for application events
- **BookingService.java**: Interface for booking services
- **CargoInspectionService.java**: Interface for cargo inspection
- **HandlingEventService.java**: Interface for handling events
- **internal/**: Contains implementations of the service interfaces
- **util/**: Contains utilities like ApplicationSettings and DateConverter

Domain Layer

- **model/cargo/**: Contains the Cargo aggregate (Cargo, Itinerary, Leg, RouteSpecification, etc.)
- **model/handling/**: Contains the Handling aggregate (HandlingEvent, HandlingHistory, etc.)
- **model/location/**: Contains the Location aggregate (Location, UnLocode)
- **model/voyage/**: Contains the Voyage aggregate (Voyage, VoyageNumber, Schedule, etc.)
- **service/**: Contains domain services like RoutingService
- **shared/**: Contains shared utilities like Specification pattern implementations

Infrastructure Layer

- **events/**: Contains CDI event infrastructure
- **logging/**: Contains logging infrastructure
- **messaging/**: Contains JMS messaging infrastructure

- **persistence/**: Contains JPA repository implementations
- **routing/**: Contains the external routing service adapter

Interfaces Layer

- **booking/facade/**: Contains the booking service facade
- **booking/sse/**: Contains server-sent events for real-time tracking
- **booking/web/**: Contains web interfaces for booking
- **handling/file/**: Contains file upload handling
- **handling/mobile/**: Contains mobile interfaces for handling
- **handling/rest/**: Contains REST API for handling
- **tracking/web/**: Contains web interfaces for tracking

Installation for Development

To set up a development environment for the Cargo Tracker application:

1. **Clone the repository**:

   [] git clone https://github.com/eclipse-ee4j/cargotracker.git cd cargotracker

2. **Import the project into your IDE**:

   - For Eclipse: File > Import > Maven > Existing Maven Projects
   - For IntelliJ IDEA: File > Open > Select the project directory

3. **Build the project**:

   [] mvn clean install

4. **Run the application in development mode**:

   [] mvn liberty:dev

5. **Access the application**:

   - http://localhost:9080/cargo-tracker/

Build and Deployment Process

Building the Application To build the Cargo Tracker application:

[] mvn clean package

This will:

- Compile the Java source code
- Process resources
- Run unit tests
- Package the application as a WAR file in the `target` directory

Deployment Options

Liberty Server The project includes configuration for Open Liberty:

[] # Run with Liberty mvn liberty:run

Run in development mode with hot reloading mvn liberty:dev

Docker Deployment The project includes a Dockerfile for containerized deployment:

[] # Build the Docker image docker build -t cargo-tracker .

Run the container docker run -p 9080:9080 -p 9443:9443 cargo-tracker

Manual Deployment You can deploy the generated WAR file to any Jakarta EE 9+ compatible application server:

1. Build the application: `mvn clean package`
2. Locate the WAR file in the `target` directory
3. Deploy the WAR file to your application server according to its documentation

Coding Standards and Conventions

The Cargo Tracker application follows these coding standards and conventions:

Java Code Style

- **Naming Conventions**:
  - Classes: PascalCase (e.g., `CargoRepository`)
  - Interfaces: PascalCase (e.g., `RoutingService`)
  - Methods: camelCase (e.g., `findByTrackingId`)
  - Variables: camelCase (e.g., `trackingId`)
  - Constants: UPPER_SNAKE_CASE (e.g., `DEFAULT_DEADLINE_DAYS`)
- **Package Structure**:
  - Follows DDD layering: domain, application, infrastructure, interfaces
  - Package names are all lowercase (e.g., `org.eclipse.cargotracker.domain.model.cargo`)
- **Code Organization**:
  - One primary class or interface per file
  - Related classes (like inner classes) in the same file
  - Static imports used sparingly

Documentation Standards

- **Javadoc**:
  - All public classes and methods should have Javadoc comments
  - Include `@param`, `@return`, and `@throws` tags where applicable
- **Code Comments**:
  - Use comments to explain "why" not "what"
  - Complex algorithms should be documented
  - TODO comments should include a description of what needs to be done

Domain-Driven Design Conventions

- **Ubiquitous Language**:
  - Use consistent terminology throughout the codebase
  - Class and method names should reflect domain concepts
- **Aggregates**:
  - Clearly defined aggregate roots (Cargo, HandlingEvent, Location, Voyage)
  - Aggregates accessed only through their roots
- **Value Objects**:
  - Immutable
  - Equality based on attributes, not identity
  - No side effects

API Documentation

The Cargo Tracker application exposes several APIs for integration with external systems:

REST APIs

Handling Report API

- **Endpoint**: `POST /handling/reports`
- **Description**: Registers a handling event for cargo
- **Consumes**: `application/json`, `application/xml`
- **Request Body**:

  [] { "completionTime": "2023-04-01T12:00:00", "trackingId": "ABC123", "eventType": "LOAD", "unLocode": "USNYC", "voyageNumber": "V100" }
- **Response**: HTTP 204 No Content (success), HTTP 400 Bad Request (validation error)

PathFinder API

- **Endpoint**: `GET /graph-traversal/shortest-path`
- **Description**: Finds possible routes between two locations
- **Parameters**:
  - `origin`: UN/LOCODE of the origin location
  - `destination`: UN/LOCODE of the destination location
  - `deadline`: Deadline for arrival (YYYYMMDD format)
- **Produces**: `application/json`, `application/xml`
- **Example Request**: `/graph-traversal/shortest-path?origin=CNHKG&destination=USNYC&deadline=20230501`
- **Example Response**:

  [] [{ "edges": [{ "voyageNumber": "V100", "fromUnLocode": "CNHKG", "toUnLocode": "USNYC", "fromDate": "2023-04-01T12:00:00", "toDate": "2023-04-15T12:00:00" }] }]

Application Services

BookingService Key methods:

- `bookNewCargo(origin, destination, arrivalDeadline)`: Books a new cargo
- `requestPossibleRoutesForCargo(trackingId)`: Requests possible routes for a cargo
- `assignCargoToRoute(trackingId, itinerary)`: Assigns a route to a cargo
- `changeDestination(trackingId, destination)`: Changes the destination of a cargo
- `changeDeadline(trackingId, deadline)`: Changes the deadline of a cargo

HandlingEventService Key methods:

- `registerHandlingEvent(completionTime, trackingId, voyageNumber, unLocode, type)`: Registers a handling event

CargoInspectionService Key methods:

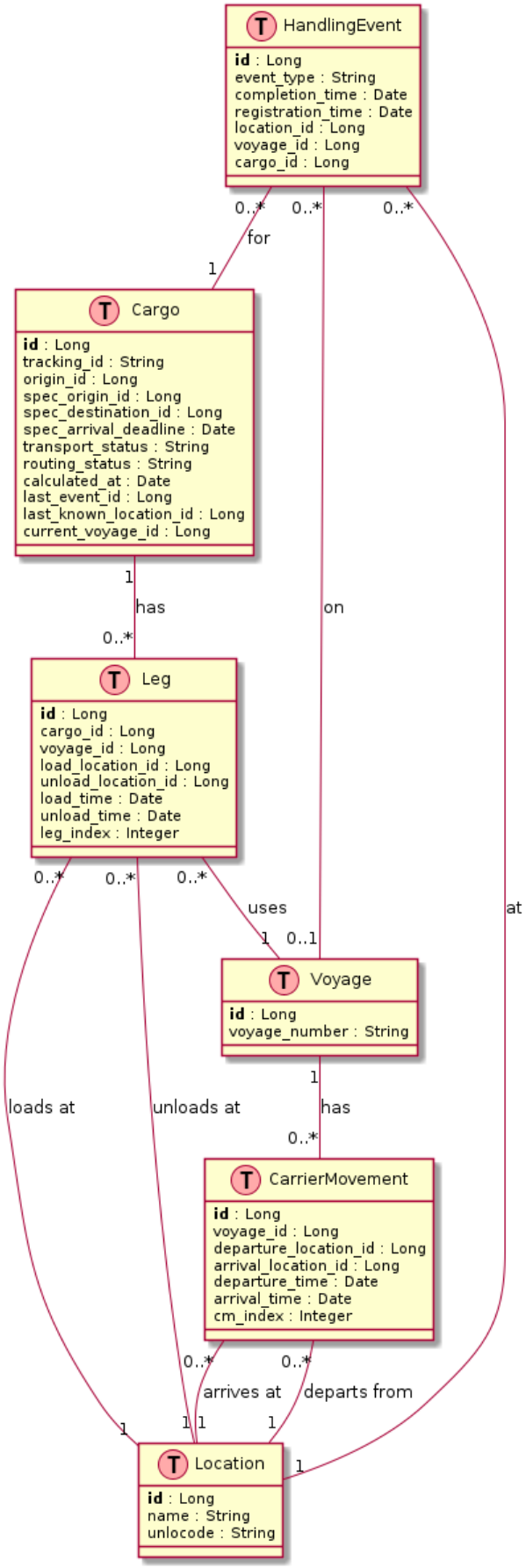
- `inspectCargo(trackingId)`: Inspects a cargo to update its status

Domain Model The domain model exposes several key entities and value objects:

- **Cargo**: Represents a cargo shipment
- **Itinerary**: Represents a planned route for a cargo
- **HandlingEvent**: Represents a handling event for a cargo
- **Location**: Represents a location with a UN/LOCODE
- **Voyage**: Represents a voyage between locations

Database Schema and Interaction

The Cargo Tracker application uses JPA for database interaction. The main entities and their relationships are:


HandlingEvent
id : Long
event_type : String
completion_time : Date
registration_time : Date
location_id : Long
voyage_id : Long
cargo_id : Long
0..*
0..*
0..*
for
1
Cargo
id : Long
tracking_id : String
origin_id : Long
spec_origin_id : Long
spec_destination_id : Long
spec_arrival_deadline : Date
transport_status : String
routing_status : String
calculated_at : Date
last_event_id : Long
last_known_location_id : Long
current_voyage_id : Long
1
has
0..*
on
Leg
id : Long
cargo_id : Long
voyage_id : Long
load_location_id : Long
unload_location_id : Long
load_time : Date
unload_time : Date
leg_index : Integer
0..*
0..*
0..*
uses
1
0..1
at
Voyage
id : Long
voyage_number : String
1
loads at
unloads at
has
0..*
CarrierMovement
id : Long
voyage_id : Long
departure_location_id : Long
arrival_location_id : Long
departure_time : Date
arrival_time : Date
cm_index : Integer
0..*
0..*
arrives at
departs from
1
1
1
1
Location
id : Long
name : String
unlocode : String
1

The application uses JPA repositories to interact with the database:

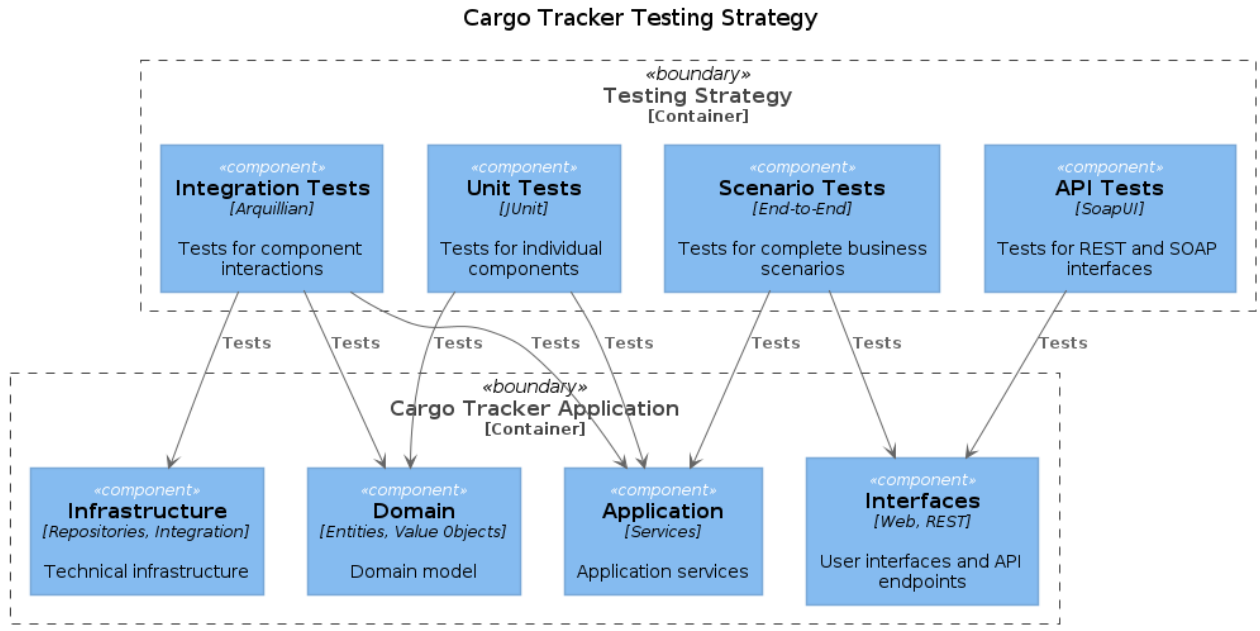
- **CargoRepository**: Manages Cargo entities
- **HandlingEventRepository**: Manages HandlingEvent entities
- **LocationRepository**: Manages Location entities
- **VoyageRepository**: Manages Voyage entities

These repositories are defined as interfaces in the domain layer and implemented in the infrastructure layer using JPA.

Testing

Testing Strategy Overview

The Cargo Tracker application employs a multi-layered testing strategy to ensure quality and correctness:



Types of Tests

The Cargo Tracker application includes several types of tests:

Unit Tests Unit tests focus on testing individual components in isolation:

- **Domain Model Tests**: Verify the behavior of domain entities, value objects, and aggregates.
- **Application Service Tests**: Verify the behavior of application services with mocked dependencies.
- **Infrastructure Component Tests**: Verify the behavior of infrastructure components with mocked dependencies.

Key unit test files:

- `CargoTest.java`: Tests for the Cargo entity
- `ItineraryTest.java`: Tests for the Itinerary value object
- `RouteSpecificationTest.java`: Tests for the RouteSpecification value object
- `HandlingEventTest.java`: Tests for the HandlingEvent entity
- `BookingServiceTest.java`: Tests for the BookingService

Integration Tests Integration tests verify the interaction between components:

- **Application Layer Integration Tests**: Test application services with real dependencies.
- **Infrastructure Layer Integration Tests**: Test infrastructure components with real dependencies.

- **Repository Tests**: Verify the persistence layer with a test database.

Key integration test files:

- `ExternalRoutingServiceTest.java`: Tests for the external routing service integration

Scenario Tests Scenario tests verify complete business scenarios:

- **End-to-End Tests**: Test the entire application flow from user interface to database.
- **Business Process Tests**: Verify that business processes work correctly across multiple components.

Key scenario test files:

- `CargoLifecycleScenarioTest.java`: Tests the complete cargo lifecycle from booking to delivery

API Tests API tests verify the external interfaces:

- **REST API Tests**: Verify the REST endpoints using SoapUI.
- **SOAP API Tests**: Verify the SOAP interfaces using SoapUI.

Key API test files:

- `cargo_tracker_soapUI_project.xml`: SoapUI project for testing REST APIs

Running Tests Locally

To run the tests locally:

[] # Run all tests mvn test

Run a specific test class mvn test -Dtest=CargoTest

Run a specific test method mvn test -Dtest=CargoTest#testRoutingStatus

To run the tests with code coverage:

[] mvn test jacoco:report

The coverage report will be generated in `target/site/jacoco/index.html`.

Continuous Integration & Testing

The Cargo Tracker application uses GitHub Actions for continuous integration and testing:

- **Pull Request Builds**: All tests are run when a pull request is opened or updated.
- **Main Branch Builds**: All tests are run when changes are pushed to the main branch.
- **Nightly Builds**: Comprehensive test suites are run nightly to catch regressions.

The CI workflow is defined in `.github/workflows/main.yml` and includes:

- Building the application
- Running unit tests
- Running integration tests
- Deploying to a test environment

Known Issues and Test Results

The following issues have been identified in the testing process:

- **HandlingEventService Testing**: The HandlingEventServiceTest is incomplete and does not provide adequate coverage.
- **Web Interface Testing**: Limited automated testing for the web interface.
- **Error Handling Testing**: Limited testing for error conditions and exception handling.
- **Performance Testing**: No automated performance testing.
- **Security Testing**: Limited security testing.

Recent test results show good coverage for the domain model but gaps in the application services, infrastructure, and interfaces layers.

Configuration and Deployment

Configuration Management

The Cargo Tracker application can be configured through various configuration files:

Application Configuration

- **server.xml**: Located in `src/main/liberty/config/`, this file configures the Liberty server, including:
  - HTTP and HTTPS endpoints
  - Database connections
  - Security settings
  - JMS resources
- **bootstrap.properties**: Located in `src/main/liberty/config/`, this file contains properties used during server startup.
- **persistence.xml**: Located in `src/main/resources/META-INF/`, this file configures JPA persistence units.

Environment Variables The application can be configured using the following environment variables:

Variable	Description	Default Value
DB_HOST	Database host	localhost
DB_PORT	Database port	5432
DB_NAME	Database name	cargotracker
DB_USER	Database username	postgres
DB_PASSWORD	Database password	postgres
HTTP_PORT	HTTP port	9080
HTTPS_PORT	HTTPS port	9443

Deployment Guide

Development Environment To deploy the application in a development environment:

1. **Clone the repository**:

   [] git clone https://github.com/eclipse-ee4j/cargotracker.git cd cargotracker

2. **Build and run the application**:

   [] mvn liberty:dev

3. **Access the application**:

   - http://localhost:9080/cargo-tracker/

Test Environment To deploy the application in a test environment:

1. **Build the application**:

   [] mvn clean package

2. **Deploy to a test server**:

   [] # Example for Liberty cp target/cargo-tracker.war /path/to/liberty/usr/servers/testServer/dropins/

3. **Start the server**:

   [] /path/to/liberty/bin/server start testServer

Production Environment To deploy the application in a production environment:

1. **Build the application**:

   [] mvn clean package -P production

2. **Configure the production server**:

   - Set up a database server (PostgreSQL recommended)
   - Configure the application server with appropriate resources
   - Set up HTTPS with proper certificates

3. **Deploy the application**:

   [] # Example for Liberty cp target/cargo-tracker.war /path/to/liberty/usr/servers/productionServer/dropins/

4. **Start the server**:

   [] /path/to/liberty/bin/server start productionServer

Scaling Considerations

The Cargo Tracker application can be scaled in several ways:

Horizontal Scaling

- **Multiple Application Instances**: Deploy multiple instances of the application behind a load balancer.
- **Session Replication**: Configure session replication between instances for seamless failover.
- **Stateless Design**: The application is designed to be stateless, making it suitable for horizontal scaling.

Vertical Scaling

- **Increase Resources**: Allocate more CPU, memory, and disk space to the application server.
- **Database Optimization**: Optimize database queries and indexes for better performance.

Caching

- **Second-Level Cache**: Configure JPA second-level cache for frequently accessed data.
- **Application-Level Cache**: Implement caching for expensive operations like route calculation.

Backup and Restore Procedures

Database Backup

1. **Regular Backups**: Schedule regular database backups using database-specific tools.

   [] # Example for PostgreSQL pg_dump -U postgres -F c -b -v -f cargotracker_backup.dump cargotracker

2. **Transaction Log Backups**: For point-in-time recovery, configure transaction log backups.

Application Backup

1. **Configuration Backup**: Back up all configuration files:
   - `src/main/liberty/config/server.xml`
   - `src/main/liberty/config/bootstrap.properties`
   - `src/main/resources/META-INF/persistence.xml`
2. **Custom Data Backup**: If the application uses file-based storage for any custom data, back up those files.

Restore Procedure

1. **Database Restore**:

   [] # Example for PostgreSQL pg_restore -U postgres -d cargotracker -v cargotracker_backup.dump

2. **Application Restore**:

- Restore configuration files
- Redeploy the application WAR file
- Restart the application server

Integration and APIs

API Endpoints Documentation

The Cargo Tracker application exposes several API endpoints for integration with external systems:

Handling Report API

- **Endpoint**: `POST /handling/reports`
- **Description**: Registers a handling event for cargo
- **Consumes**: `application/json`, `application/xml`
- **Response**: HTTP 204 No Content (success), HTTP 400 Bad Request (validation error)

Request Parameters:

Parameter	Type	Description	Constraints
completionTime	String	When the handling event occurred	Required, ISO-8601 format (e.g., “2023-04-01T12:00:00”)
trackingId	String	The tracking ID of the cargo	Required, minimum 4 characters
eventType	String	The type of handling event	Required, one of: RECEIVE, LOAD, UNLOAD, CUSTOMS, CLAIM
unLocode	String	The UN location code where the event occurred	Required, exactly 5 characters
voyageNumber	String	The voyage number (for LOAD and UNLOAD events)	Optional, 4-5 characters

Example Request (JSON):

[] { "completionTime": "2023-04-01T12:00:00", "trackingId": "ABC123", "eventType": "LOAD", "unLocode": "USNYC", "voyageNumber": "V100" }

Example Request (XML):

[] <?xml version="1.0" encoding="UTF-8"?> <handlingReport> <completionTime>2023-04-01T12:00:00</completionTime> <trackingId>ABC123</trackingId> <eventType>LOAD</eventType> <unLocode>USNYC</unLocode> <voyageNumber>V100</voyageNumber> </handlingReport>

PathFinder API

- **Endpoint**: `GET /graph-traversal/shortest-path`
- **Description**: Finds possible routes between two locations
- **Produces**: `application/json`, `application/xml`
- **Response**: List of possible transit paths

Request Parameters:

Parameter	Type	Description	Constraints
origin	String	The UN location code of the origin	Required, 5 characters, format: [a-zA-Z]{2}[a-zA-Z2-9]{3}

Parameter	Type	Description	Constraints
destination	String	The UN location code of the destination	Required, 5 characters, format: [a-zA-Z]{2}[a-zA-Z2-9]{3}
deadline	String	The deadline for arrival	Required, 8 characters (YYYYMMDD format)

Example Request:

```
GET /graph-traversal/shortest-path?origin=CNHKG&destination=USNYC&deadline=20230501
```

Example Response (JSON):

[] [{ "edges": [{ "voyageNumber": "V100", "fromUnLocode": "CNHKG", "toUnLocode": "USNYC", "fromDate": "2023-04-01T12:00:00", "toDate": "2023-04-15T12:00:00" }] }, { "edges": [{ "voyageNumber": "V200", "fromUnLocode": "CNHKG", "toUnLocode": "JNTKO", "fromDate": "2023-04-01T12:00:00", "toDate": "2023-04-05T12:00:00" }, { "voyageNumber": "V300", "fromUnLocode": "JNTKO", "toUnLocode": "USNYC", "fromDate": "2023-04-06T12:00:00", "toDate": "2023-04-20T12:00:00" }] }]

External Integrations and Dependencies

The Cargo Tracker application integrates with the following external systems:

PathFinder Service The PathFinder service is an external routing service that provides route planning capabilities. It is integrated through the `ExternalRoutingService` class, which adapts the external service to the `RoutingService` interface used by the domain layer.

Database The application requires a database for persistence. It is configured to work with any database supported by JPA, with PostgreSQL recommended for production use.

Authentication and Authorization

Information not available: The current implementation of the Cargo Tracker application does not include explicit authentication or authorization mechanisms. Security would typically be handled at the application or container level.

Webhooks and Callback Interfaces

Information not available: The current implementation does not appear to include webhooks or callback interfaces for external systems.

Data Exchange Formats

The Cargo Tracker application supports the following data exchange formats:

JSON Example of a handling report in JSON format:

[] { "completionTime": "2023-04-01T12:00:00", "trackingId": "ABC123", "eventType": "LOAD", "unLocode": "USNYC", "voyageNumber": "V100" }

XML Example of a handling report in XML format:

[] <?xml version="1.0" encoding="UTF-8"?> <handlingReport> <completionTime>2023-04-01T12:00:00</completionTime> <trackingId>ABC123</trackingId> <eventType>LOAD</eventType> <unLocode>USNYC</unLocode> <voyageNumber>V100</voyageNumber> </handlingReport>

Security

Security Assets Inventory

The Cargo Tracker application contains several security-sensitive assets:

Asset	Description	Security Measure	Assessment of Criticality
REST APIs	Handling Report API and PathFinder API	Bean Validation for input validation	Critical - No authentication/authorization
Database Credentials	Username and password for database access	Stored in configuration files	High - Default/weak credentials used
Cargo Data	Information about cargo shipments	Access through application services	Medium - No direct exposure, but no access control
Web Interfaces	User interfaces for booking, tracking, etc.	None identified	High - No authentication/authorization

Security Guidelines

The Cargo Tracker application, as a reference implementation, has several security considerations that should be addressed in a production environment:

Authentication and Authorization The current implementation does not include authentication or authorization mechanisms for the REST APIs or web interfaces. In a production environment, these would be critical components to ensure that only authorized users can access specific functionality:

- **REST APIs**: The Handling Report API and PathFinder API are currently accessible without authentication.
- **Web Interfaces**: The booking, tracking, and handling interfaces do not implement user authentication or role-based access control.

Data Privacy Considerations The application handles cargo tracking data, which may be considered sensitive in a real-world scenario:

- **Database Security**: Database credentials are stored in configuration files with default values ("usr"/"pwd").
- **No Data Encryption**: The application does not implement encryption for sensitive data at rest or in transit.

Vulnerability Management The application uses Bean Validation for input validation, which provides some protection against malicious input:

- **Validation Annotations**: The `HandlingReport` class uses annotations like `@NotBlank` and `@Size` to validate input fields.
- **Validation Errors**: The REST configuration is set up to return validation errors to clients when invalid data is submitted.
- **Incomplete Validation**: There are TODO comments in the code indicating that more thorough validation (using regular expressions) is needed.

Authentication and Authorization Mechanisms Information not available: The current implementation does not include authentication or authorization mechanisms.

License and Legal Information

Software Licensing

The Eclipse Cargo Tracker is open source software licensed under the Eclipse Public License 2.0 (EPL-2.0).

The full license text can be found in the LICENSE.md file in the repository.

Contribution Guidelines

Contributions to the Eclipse Cargo Tracker project are welcome. Contributors should follow these guidelines:

1. **Code of Conduct**: All contributors are expected to adhere to the Eclipse Code of Conduct.
2. **Pull Requests**: Contributions should be submitted as pull requests to the main repository.
3. **Testing**: All contributions should include appropriate tests.
4. **Documentation**: Changes should be documented appropriately.
5. **Licensing**: All contributions must be licensed under the EPL-2.0.

For more detailed information, see the CONTRIBUTING.md file in the repository.

Copyright Notices

Appendix

References

- Eclipse Cargo Tracker: https://eclipse-ee4j.github.io/cargotracker/
- Domain-Driven Design: https://domainlanguage.com/ddd/
- Jakarta EE: https://jakarta.ee/
- Security Analysis Documentation
- Interface Specifications Documentation
- Use Cases Documentation
- Testing Strategy Documentation
- Consolidated Glossary

List of Illustrations

1. Cargo Tracker Architecture Overview - Shows the layered architecture of the application
2. Cargo Tracker Testing Strategy - Shows the testing approach for different layers of the application
3. Database Schema - Shows the entity relationships in the application

Additional Resources and Further Reading

- **Books**:
  - Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans
  - Implementing Domain-Driven Design by Vaughn Vernon
  - Patterns of Enterprise Application Architecture by Martin Fowler
  - Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf
- **Online Resources**:
  - Jakarta EE Documentation: https://jakarta.ee/specifications/
  - Domain-Driven Design Community: https://dddcommunity.org/
  - Microservices.io: https://microservices.io/
  - Martin Fowler's Blog: https://martinfowler.com/
- **Related Projects**:
  - Eclipse MicroProfile: https://microprofile.io/
  - Jakarta NoSQL: https://jakarta.ee/specifications/nosql/

– Eclipse JKube: https://www.eclipse.org/jkube/