Cargo Tracker Docs

v3

2026-11-12

The Eclipse Cargo Tracker project demonstrates how you can develop applications with the Jakarta EE platform using widely adopted architectural best practices like Domain-Driven Design (DDD). The project is based on the well-known Java DDD sample application developed by DDD pioneer Eric Evans' company Domain Language and the Swedish software consulting company Citerus. The cargo example actually comes from Eric Evans' seminal book.

- The *Getting Started* section provides brief orientation on the project.
- The *Visual Studio Code* section provides detailed instructions on how to get started with Visual Studio Code.
- The *Jakarta EE and DDD* section provides a brief overview of DDD as it relates to this project and Jakarta EE.
- The *Characterization* section overviews how the basic building blocks of the domain - entities, value objects, aggregates, services, repositories and factories - are implemented in the application using Jakarta EE.
- The *Layers* section explains the architectural layers in the application and how they relate to various Jakarta EE APIs.
- The *Resources* section provides some useful resources for learning more about DDD and Jakarta EE.

You should also check out the project website for further context.

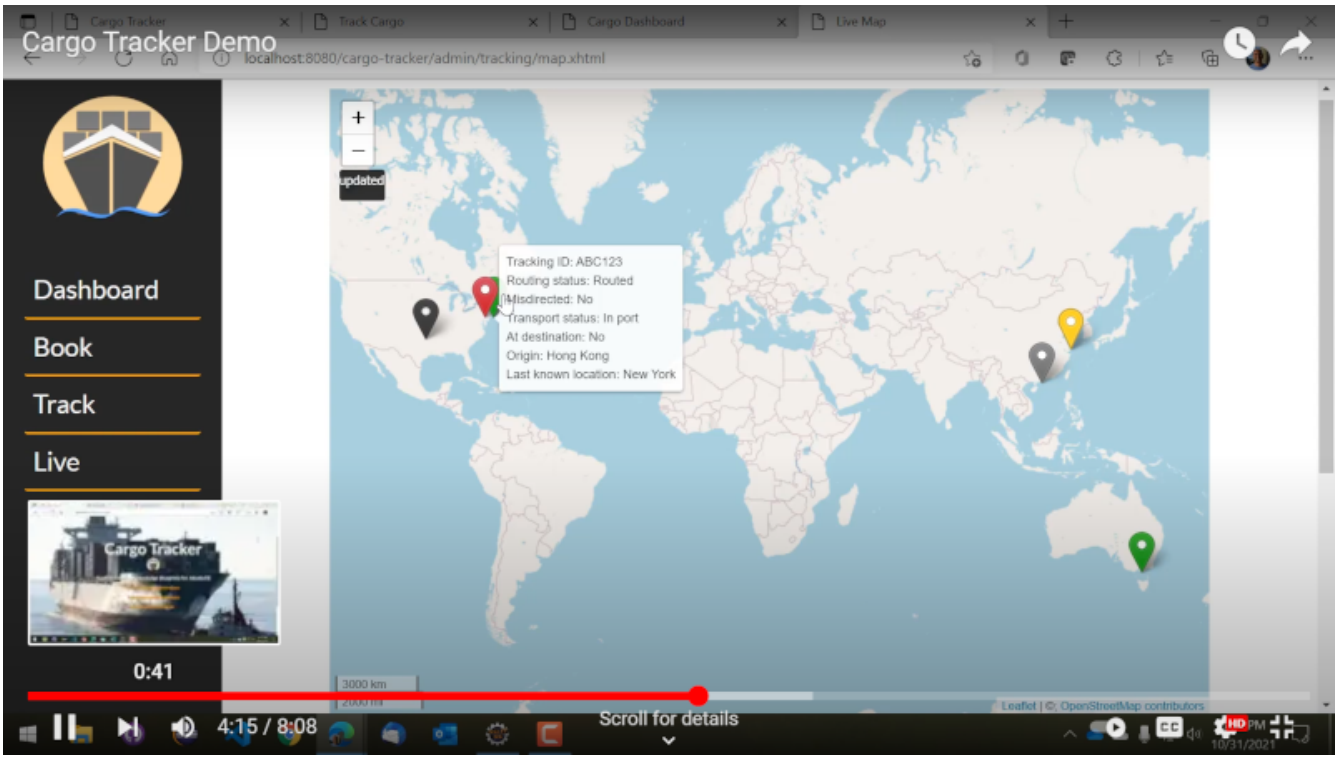
Getting Started

Screen Cast

Before exploring the code, it may be helpful to view a demo of the application functionality. The video below demos the major functionality of the Eclipse Cargo Tracker application. It is intended to be a helpful point to start exploring how the application implements Domain-Driven Design (DDD) using Jakarta EE.

The following is the functionality highlighted in the demo:

- Tracking cargo using the public interface
- Monitoring cargo using both the static dashboard and live map
- Booking and routing cargo
- Registering cargo life-cycle events using the mobile interface
- Registering cargo life-cycle events in bulk using the batch file processing interface



Exploring the Code

All of the code is available on GitHub. You can download it as a zip or browse the repository online.

Running the Application

The project is Maven based, so you should be able to easily build it or set it up in your favorite IDE. We currently have instructions for Visual Studio Code.

You can also run the application directly from the command line using the Cargo Maven plugin. All you need to do is navigate to the project source root and type the following (please make sure you have Java SE 11/Java SE 17 and JAVA_HOME set correctly):

```
mvn clean package cargo:run
```

Once the application starts up, just open up a browser and navigate to http://localhost:8080/cargo-tracker/.

This will run the application with Payara Server by default. The project also has Maven profiles to support GlassFish and Open Liberty. For example you can run using GlassFish using the following command:

```
mvn clean package -Pglassfish cargo:run
```

Similarly, you can run using Open Liberty using the following command:

```
mvn clean package -Popenliberty liberty:run
```

Cloud Demo

Cargo Tracker is deployed to Kubernetes on the cloud using GitHub Actions workflows. You can find the demo deployment on the Scaleforce cloud at https://cargo-tracker.j.scaleforce.net. This project is very thankful to our sponsors Jelastic and Scaleforce for hosting the demo! The deployment and all data is refreshed nightly.

Visual Studio Code

This section outlines how to set up the application in Visual Studio Code. It uses the project default Payara Server. The project also has Maven profiles to support GlassFish and Open Liberty. GlassFish or Open Liberty can be very similarly used with Visual Studio Code.

Prerequisites

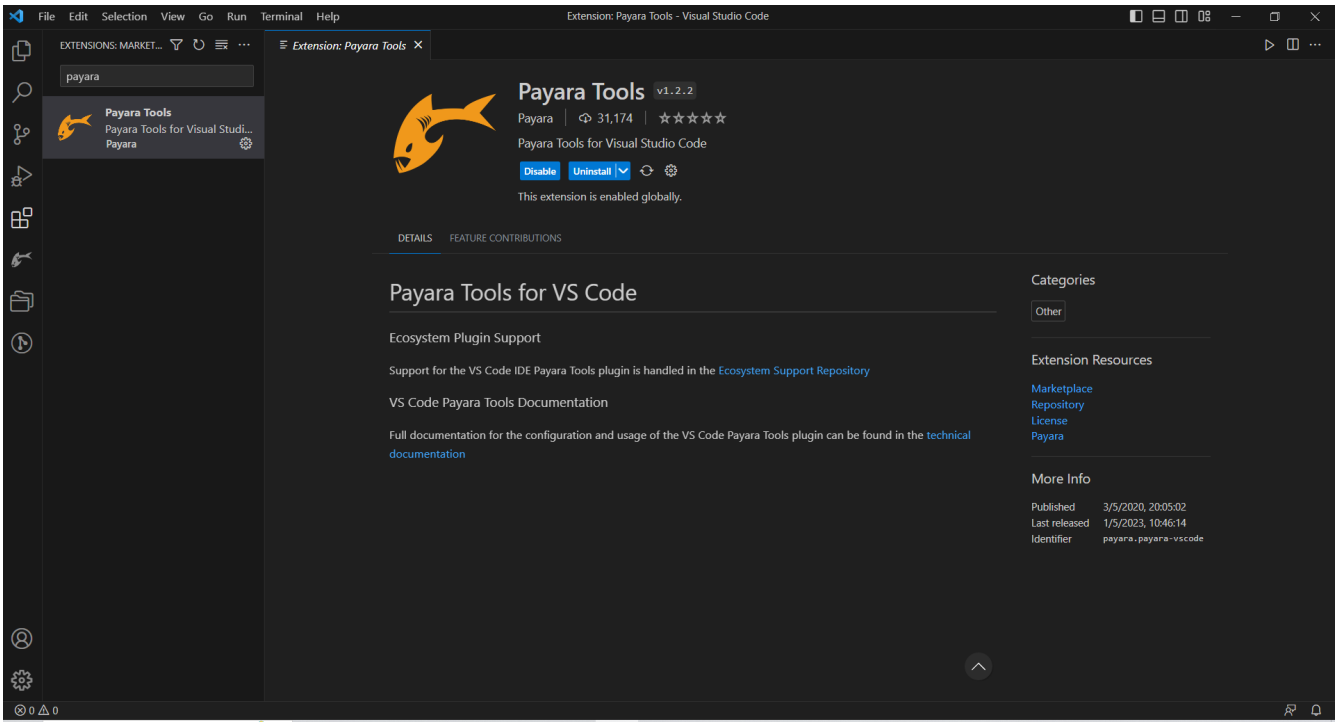
- Java SE 11 or Java SE 17 is required. Please make sure that you have properly set up the JAVA_HOME environment variable.
- Payara Server 6 is required. You can download Payara Server 6 from here.
- Visual Studio Code is required. You can download Visual Studio Code from here.
- Ensure that the Visual Studio Code Extension Pack for Java is installed.

Download

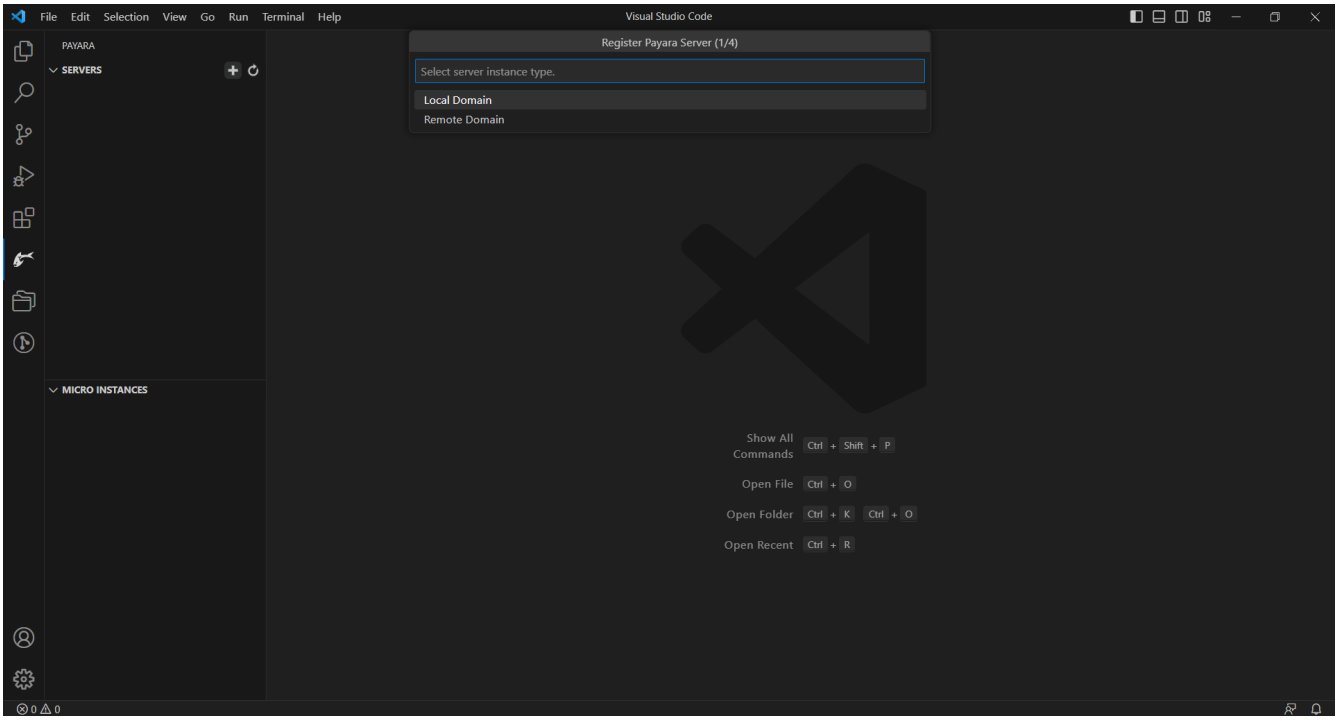
Download the source code zip file and expand it somewhere in your file system. Note that this is a Maven project.

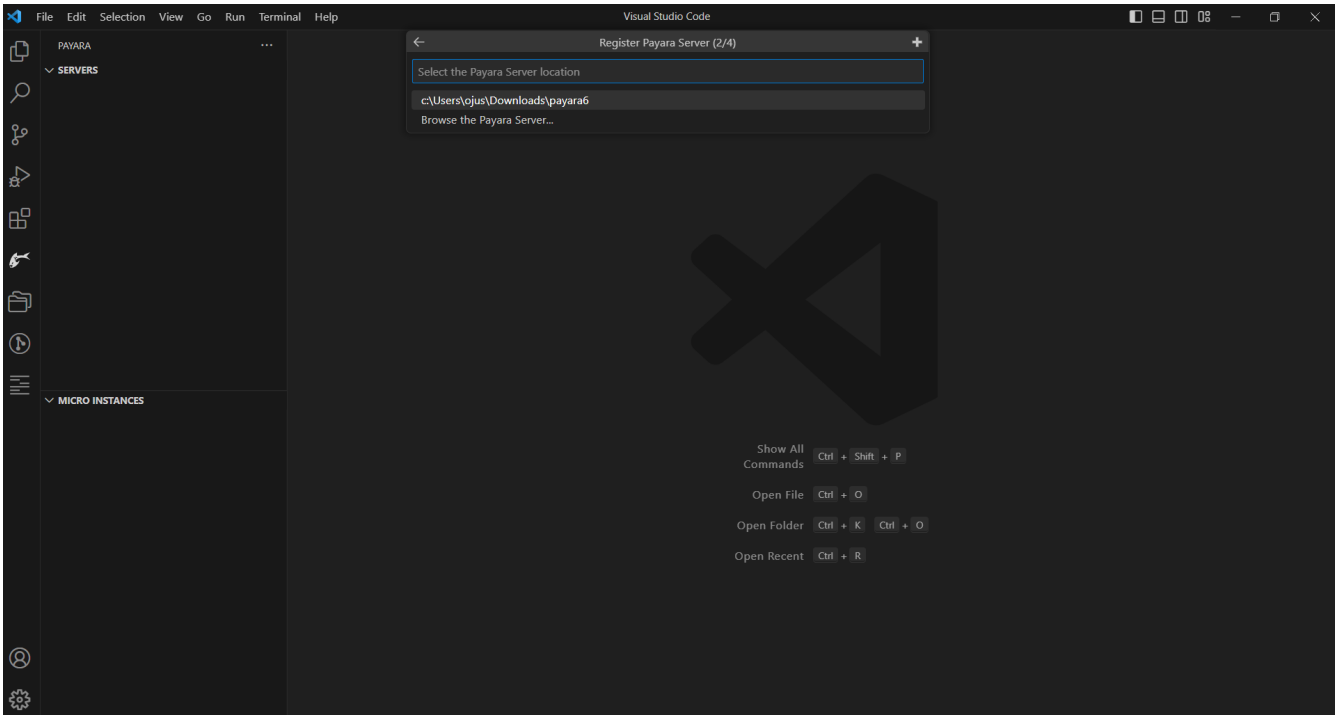
Visual Studio Code Setup

Start Visual Studio Code. Go to → Extensions (Ctrl+Shift+X). Search for Payara Tools. Install Payara Tools.

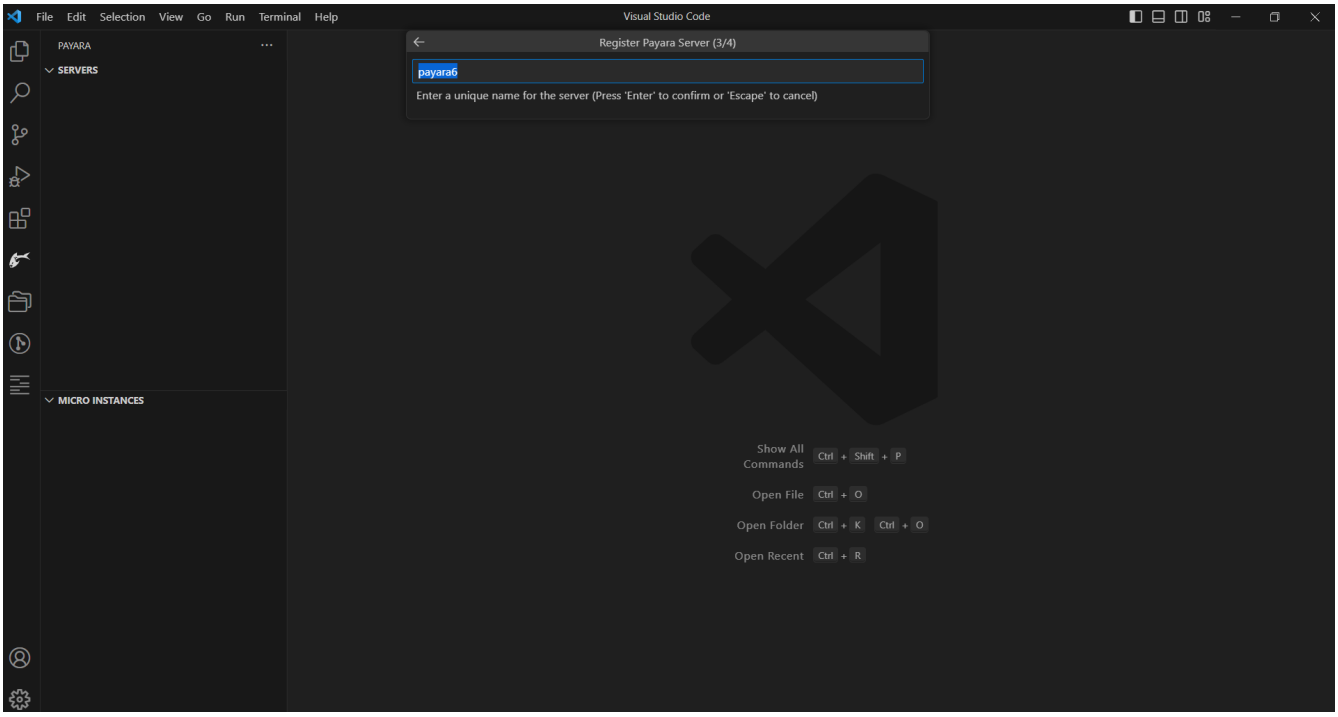


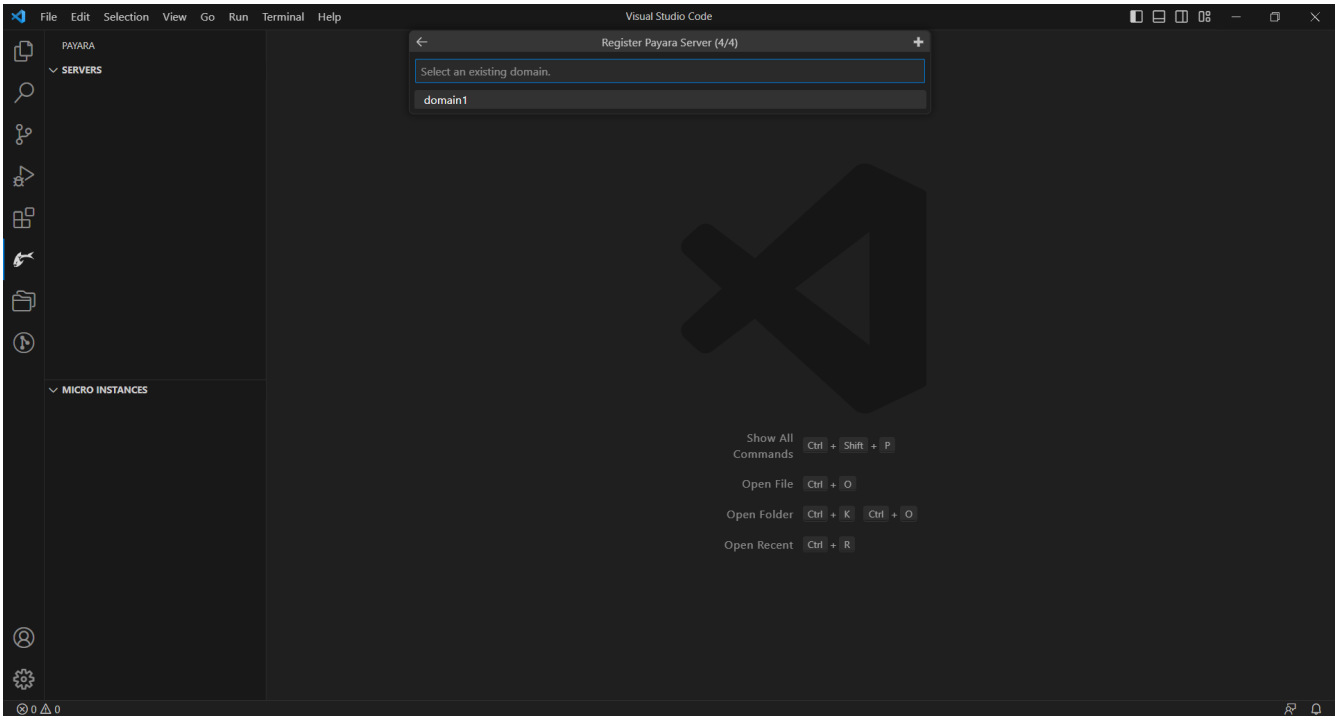
Once the Payara Tools plugin is installed, click on the Payara icon on the left side bar below the Extensions icon, then click on the '+' icon on the Servers. Choose 'Local Domain' → 'Browse the Payara Server', then select the directory where you installed Payara 6.



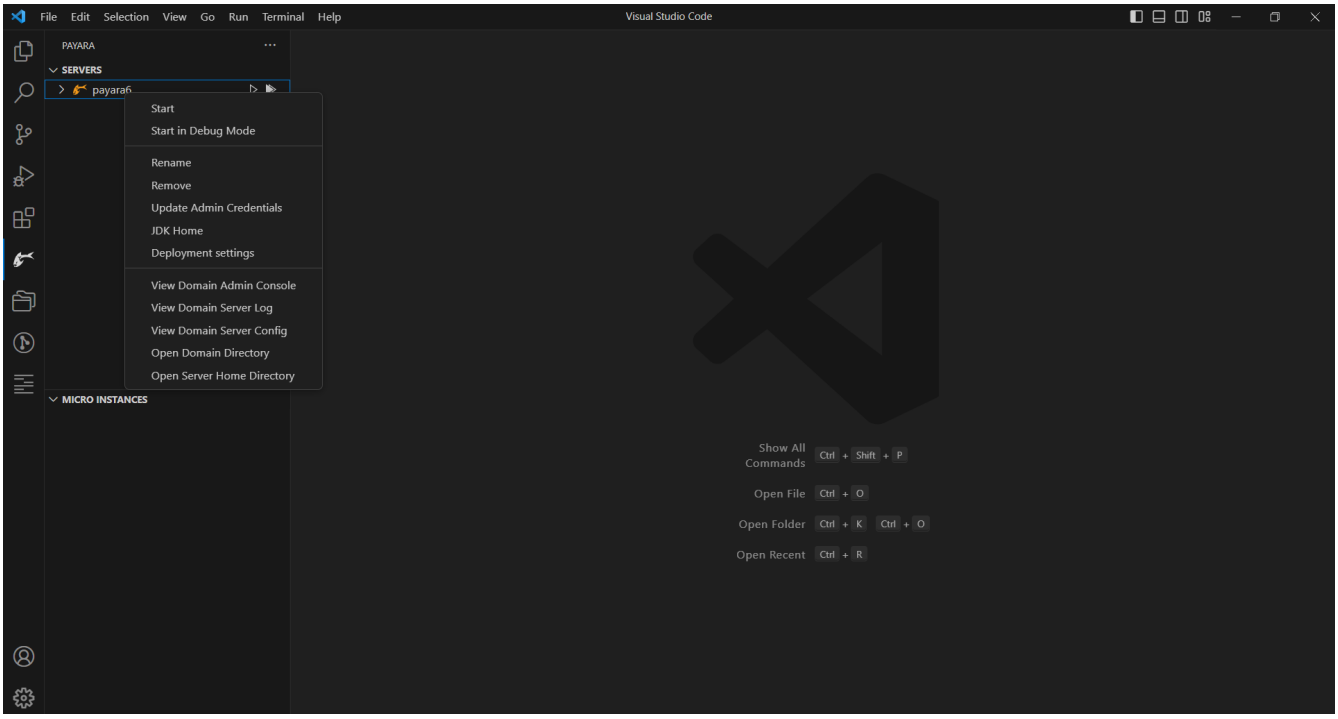


Name the instance Payara 6 and hit next. Select the default domain1.



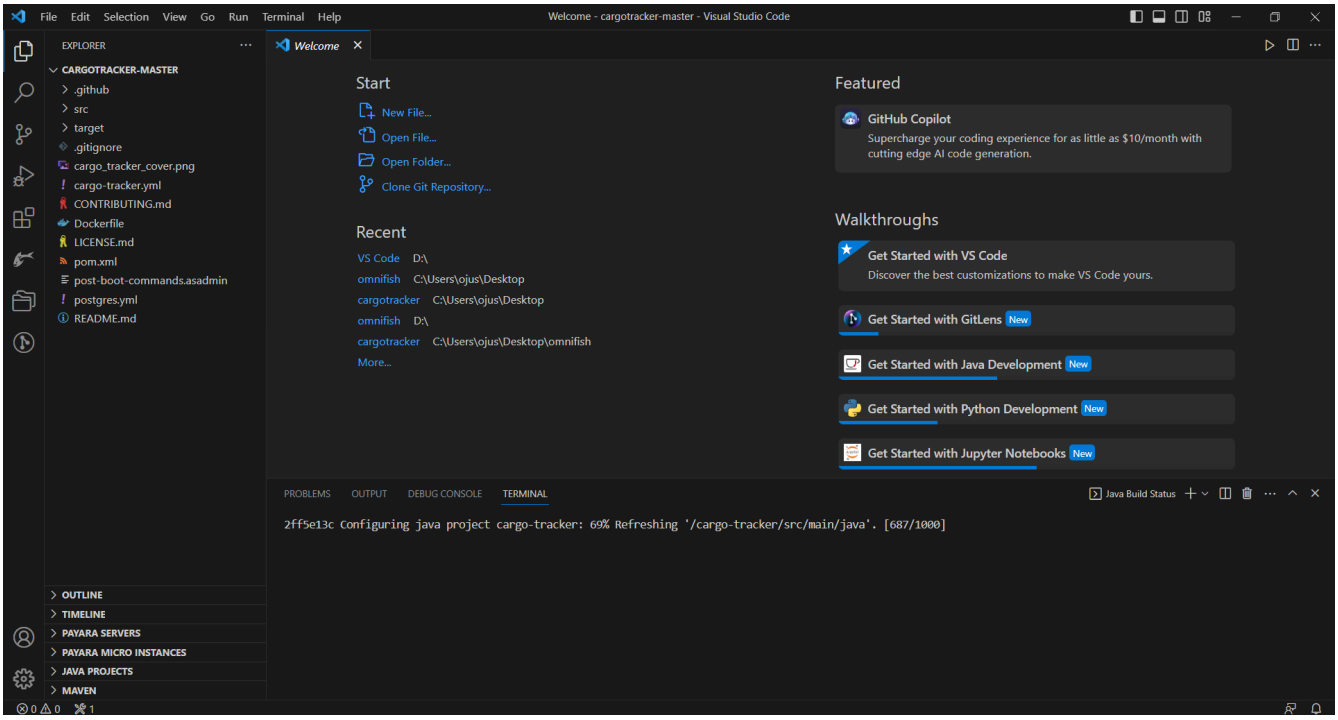


Now you will see the instance under Servers. Right-click on the instance and select 'Start'. Once Payara starts, you can verify it is up by visiting http://localhost:8080.

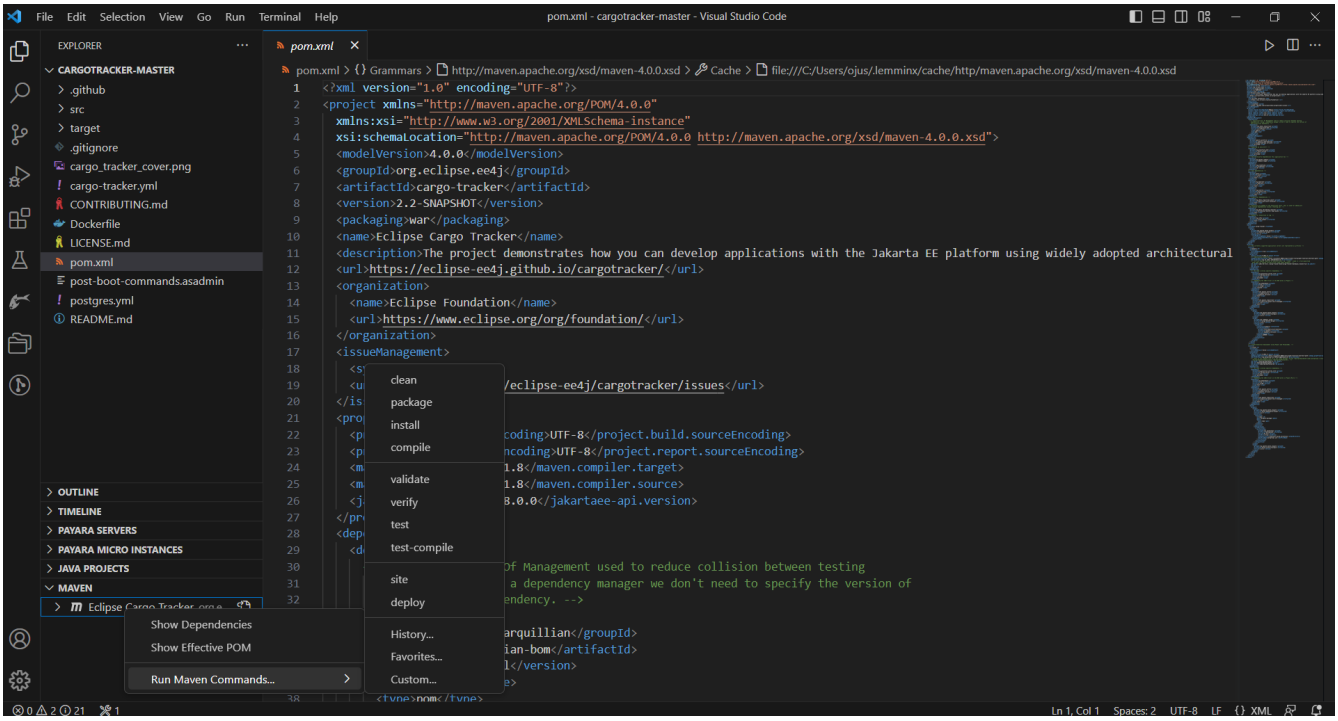


Running the Application

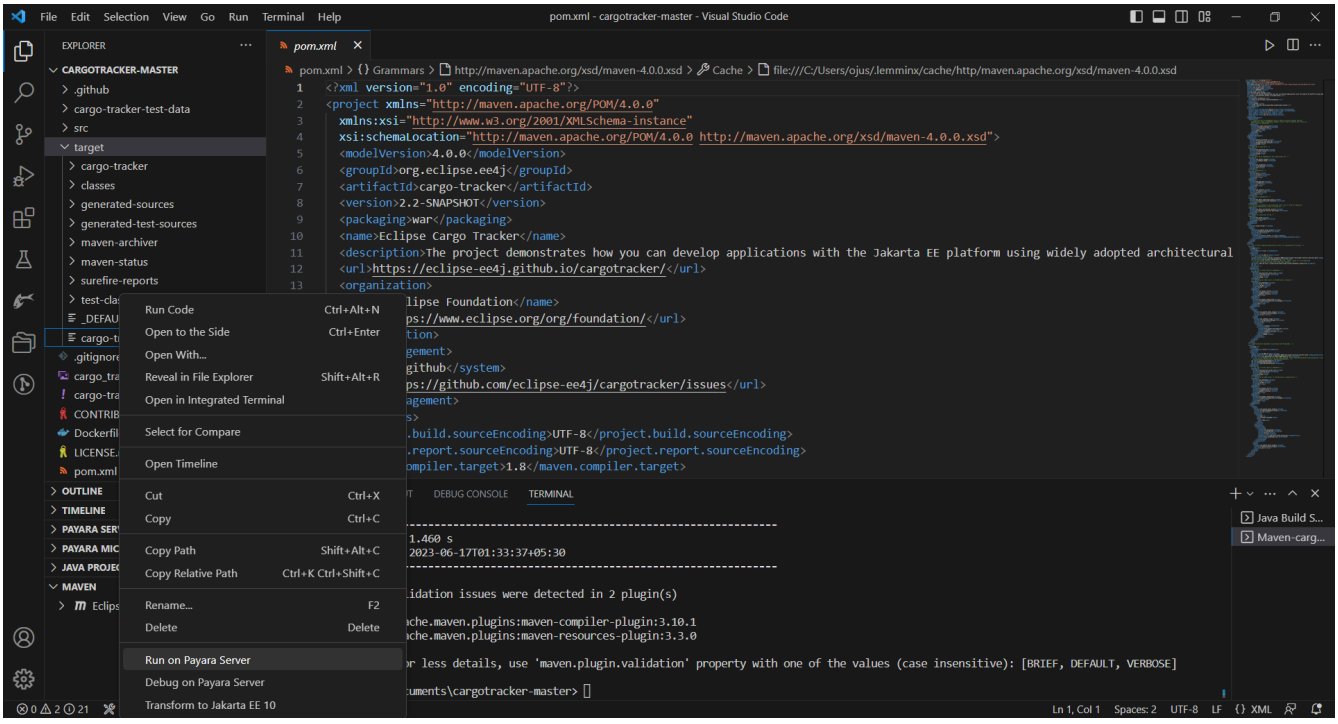
You will now need to get the application into Visual Studio Code. Go to File → Open Folder → Select the root directory of the Cargo Tracker project in your file system, and hit finish. Visual Studio Code will automatically identify it as a Maven project. It will take a few minutes to import the project for the first time.



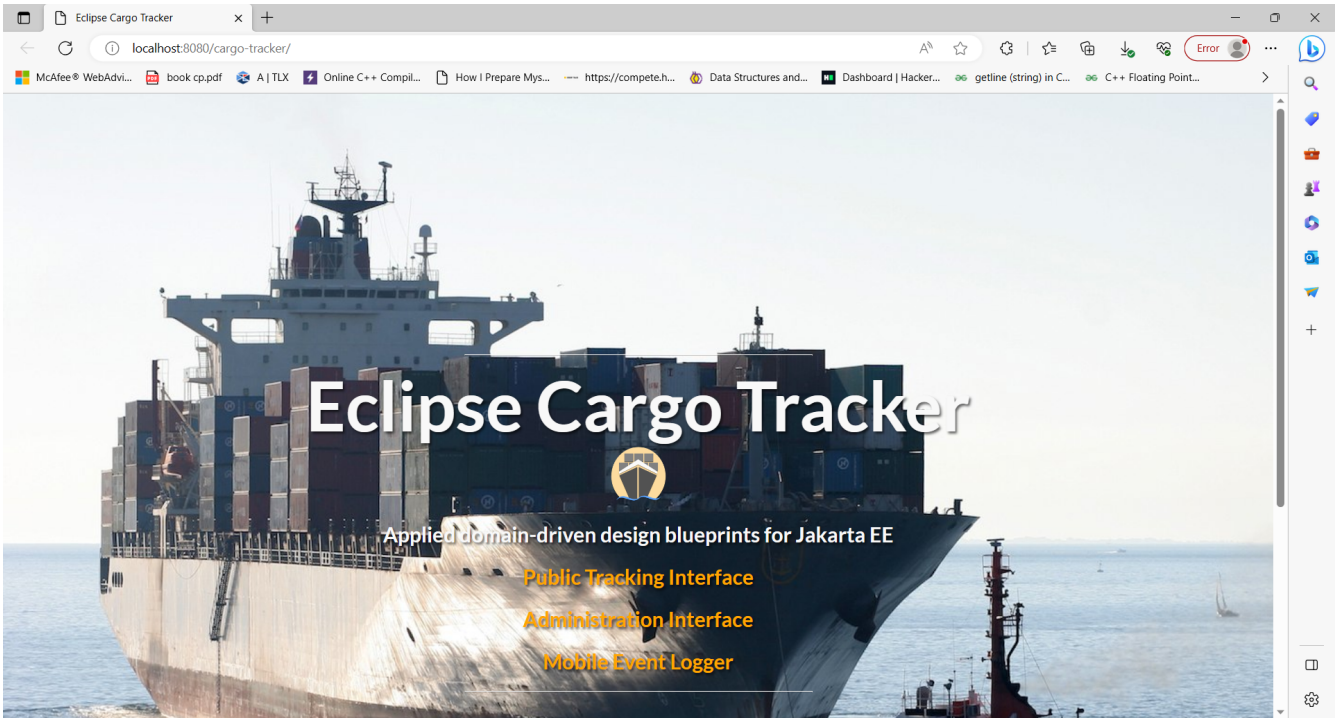
After the project loads, go to the Maven tab on the left bottom side. You will see that Eclipse Cargo Tracker is a recognized Maven project. Right click on it and run the 'clean' Maven command. Finally, run the 'package' Maven command. It will take a while the very first time as Maven downloads dependencies.



After this, a war file named 'cargo-tracker.war' will be built under the 'target' directory. Right-click on the war file and select the 'Run on Payara Server' option.



The first time start up might take a bit of time. Once the deployment is done, Visual Studio Code will automatically open up a default browser window with the application.



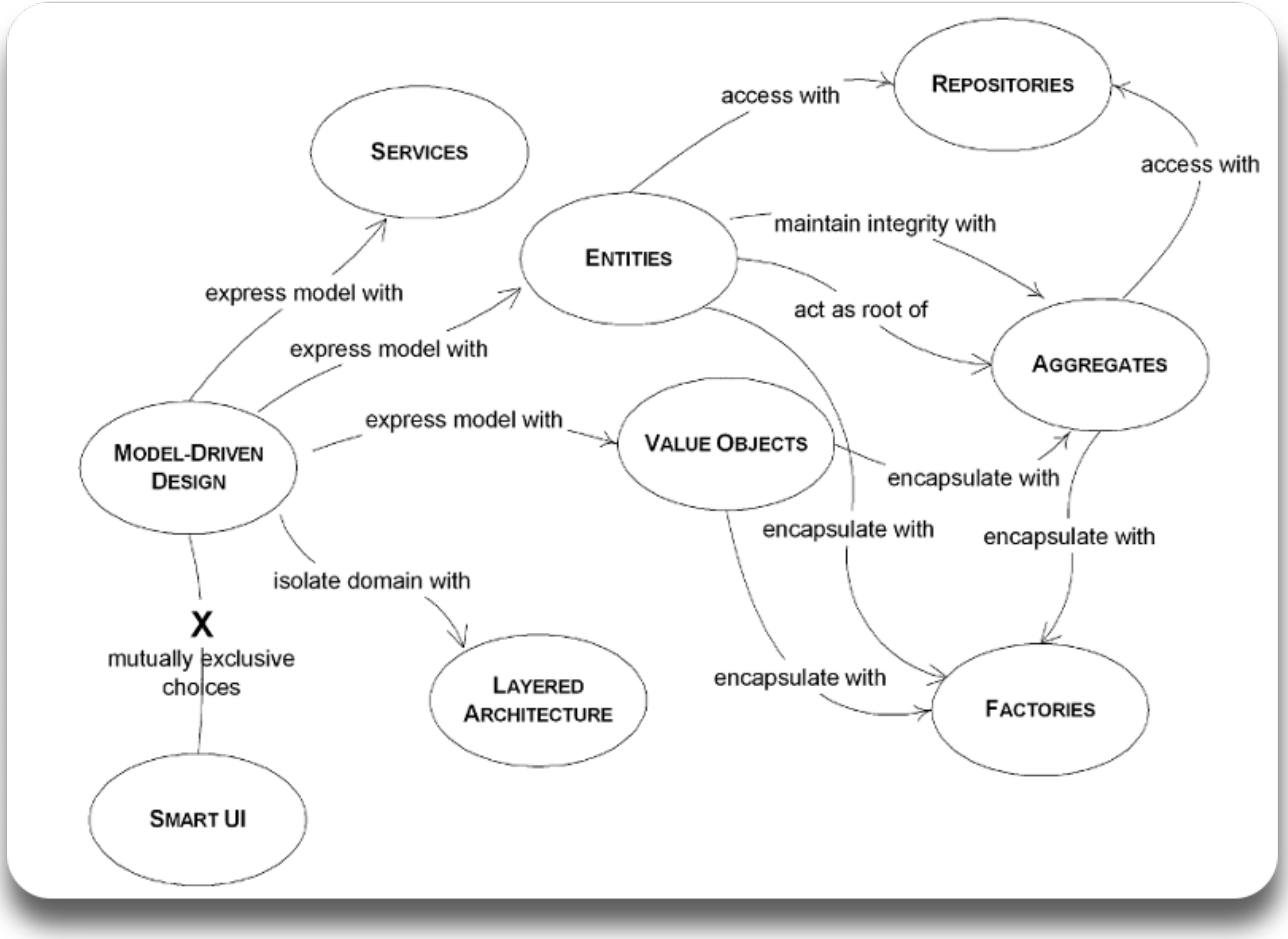
There is a tracking interface to track the current status of cargo and a booking interface to book and route cargo. You should explore both interfaces before diving into the code. You should also check out the REST and file processing interfaces to register handling events as well as the mobile web interface. You can test against the REST interfaces using our soapUI tests.

Once you are done, click on the 'x' icon on the Payara server instance to stop the Cargo Tracker application.

Jakarta EE and DDD

As the name implies, Domain-Driven Design is an approach to software design and development that focuses on the core domain and domain logic. The domain is implemented through a careful focus on traditional OOAD (Object Oriented Analysis and Design) and modeling the real world problem the software is trying to solve as closely as possible.

The basic building blocks of the domain are entities, value objects, aggregates, services, repositories, and factories. The *Characterization* section overviews how these concepts are implemented in the application using Jakarta EE. Logical layers partitioning distinct concerns are super-imposed on the core concept of the domain. These layers generally consist of the UI/interface layer, the application layer, the domain layer (of course!), and the infrastructure layer respectively. The *Layers* section explains the architectural layers in the application and how they relate to various Jakarta EE APIs.



Characterization

Careful characterization of classes is a key activity when doing Domain-Driven Design. Fortunately, most of the time, it is fairly obvious what category a particular class belongs to. Other times, it is not as easy to sort out, and careful analysis, team collaboration, and refactoring are necessary.

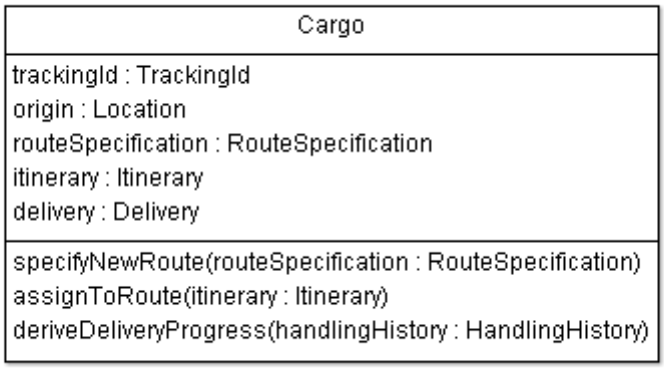
The trickiest ones to classify are typically Entities, Aggregates, Value Objects and Domain Events. When possible, you should favor Value Objects over Entities or Domain Events, because they require less attention during implementation. Value Objects can be created and thrown away at will, and since they are immutable we can pass them around as we wish. We must be much less cavalier with Entities as identity and life-cycle have to be carefully managed.

Below is a short walk-through of key classes in the application and the motivation behind their implementation choice.

Entities

Cargo has both a clear identity and a life-cycle with state transitions that we care about, so it's an entity. It is obviously a key concept in the domain. Many cargo instances will exist in the system simultaneously. The different instances may have the same origin and destination, they may even contain the same kind of things, but it is important for us to be able to track individual cargo instances. In our case the cargo's identity is its tracking ID. The tracking ID is assigned upon creation and is never changed. This is the handle used to track the cargo's progress (e.g. from the tracking website).

The cargo's delivery state will change during its lifetime. Its transport status will start out as NOT_RECEIVED, i.e., booked but not yet handed over to the shipping company at the port, and in the normal case ends its life as CLAIMED (note that this is a property of the cargo's Delivery), tracking the current state of the cargo. During a cargo's lifetime, it may be assigned new destinations, its itinerary may be changed many times, and its delivery will be recalculated as new HandlingEvents are received.



Voyage is a vessel's trip from origin to destination, typically made up of several segments (CarrierMovements). In the application, a Voyage consists of a Schedule with the different CarrierMovements in it and has a very clear notion of identity, VoyageNumber. This ID could be something like a flight number for air shipments or a vessel voyage number for a ship; it is not the

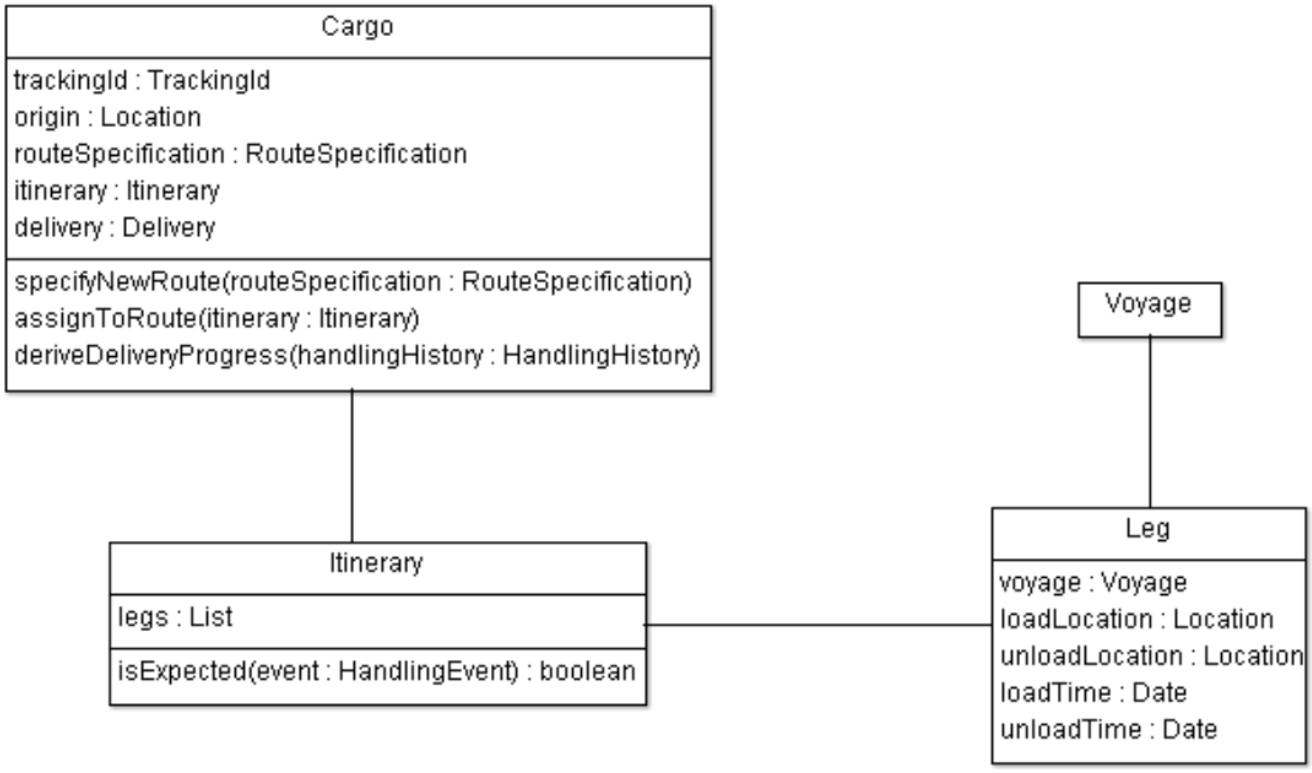
name or the identification of the actual vessel.

Unsurprisingly, domain entities are usually implemented as Jakarta Persistence (JPA) entities. Note, however, that sometimes it is necessary or convenient to implement value objects as Jakarta Persistence (JPA) entities as well.

Value Objects

A Leg consists of a starting point and an ending point (to Location and from Location), and a reference to a voyage. A leg has no sense of identity; two legs with the same from location, end location, and voyage are in our model completely interchangeable. As a result, we implement Leg as an immutable Value Object.

An Itinerary consists of a list of Legs, with the load location of the first leg in the list as the starting point of the itinerary and the unload location of the last leg as the final destination. The same reasoning applies to itineraries as to Legs: they do not have identity and are implemented as Value Objects. Now, a cargo can certainly have its itinerary updated. One way to accomplish this would be to keep the original itinerary instance and update the Legs in the itinerary's list. In this case, the itinerary must be mutable and has to be implemented as an entity. With the itinerary as a Value Object, as in the case of our application model and implementation, updating it is a simple operation of acquiring a complete new itinerary from the RoutingService and replacing the old one. Implementation of a cargo's itinerary management is much simplified by having the itinerary as a Value Object.



Value Objects are typically implemented as Jakarta Persistence (JPA) embedded objects. However, it is sometimes useful and valid to implement them as Jakarta Persistence (JPA) entities.

Domain Events

Some things clearly have identity but no life-cycle, or an extremely limited life-cycle with just one state. We call these things Domain Events and they can be viewed as a hybrid of Entities and Value Objects.

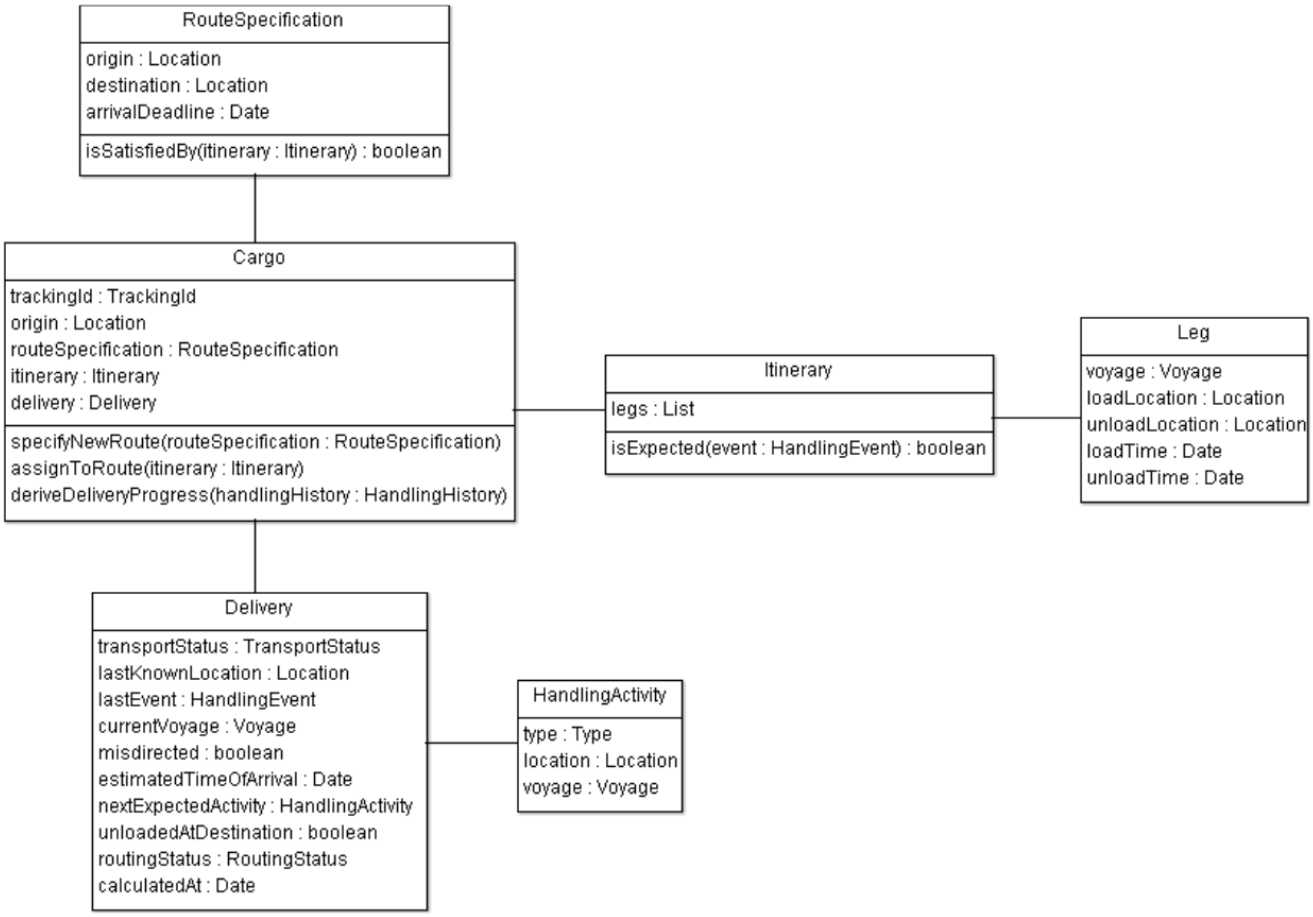
In our application, HandlingEvent is a Domain Event that represents a real-life event such as cargo being loaded or unloaded, customs cleared, etc. They carry both a completion time and a registration time. The completion time is the time when the event occurred, and the registration time is the time when the event was received by the system. The HandlingEvent ID is composed of the cargo, voyage, completion time, location, and type (LOAD, UNLOAD, etc).

Domain Events are usually implemented as Jakarta Persistence (JPA) entities.

Aggregates

In real life, most things are connected, directly or indirectly. Mimicking this approach when building large software systems tends to bring unnecessary complexity and poor performance. DDD provides tactics to help you sort these things out, with aggregates being one of the most important ones. Aggregates help with decoupling large structures by setting rules for relations between entities. An aggregate is essentially a very closely related set of entities and value objects. An aggregate root is a special kind of entity in the aggregate that controls external access to the set of closely related entities and value objects.

Cargo is the central aggregate in the application. The classes in the cargo aggregate are in the org.eclipse.cargotracker.domain.model.cargo package. Cargo is the aggregate root, and the aggregate also contains the Value Objects delivery, itinerary, leg, and a few more classes.

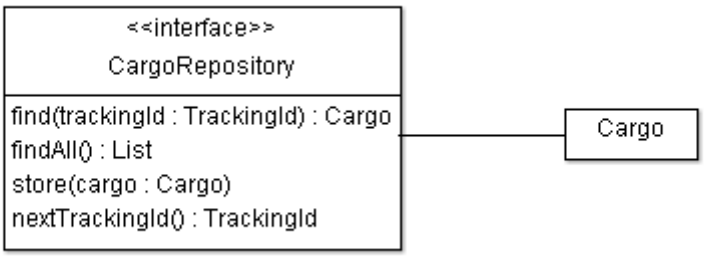


Handling is another important aggregate. It contains the handling events that are registered throughout a cargo's progress from NOT_RECEIVED to CLAIMED. The handling events have a relation to the cargo to which the event belongs. This is allowed since cargo itself is an aggregate root.

The main reason for not making handling events part of the cargo aggregate is performance. Handling events are received from external parties and systems, e.g., warehouse management systems, port handling systems, that call our HandlingReportService REST web service implementation. The number of events can be very high, and it is important that our web service can dispatch the remote calls quickly. To be able to support this use case, we need to handle the web service calls asynchronously, i.e., we do not want to load the big cargo structure synchronously for each received handling event. Since all relationships in an aggregate must be handled synchronously, we put the handling event in an aggregate of its own, enabling us to process the events quickly and at the same time eliminate lock contention in the system.

Repositories

With the aggregates and their roots identified, it is fairly trivial to identify the Repositories. Repositories retrieve and save aggregate roots from and to persistent storage. In our application, there is one Repository per aggregate root. For example, the CargoRepository is responsible for finding and storing cargo aggregates. The finders return Cargo instances or lists of Cargo instances.



While Repository interfaces are part of the domain layer, their implementations are part of the infrastructure layer. For example, the CargoRepository has a Jakarta Persistence (JPA) implementation in the infrastructure layer, JpaCargoRepository.

Repositories in Jakarta EE are typically implemented using Jakarta Persistence (JPA) and CDI. The Jakarta Data is particularly helpful in implementing repositories.

Factories

In some cases, creating entities is not as trivial as simply calling the `new` operator. In such cases, you will want to encapsulate entity creation using Factories. In our application, HandlingEventFactory is used to create handling events.

Factories are typically implemented using CDI.

Domain Services

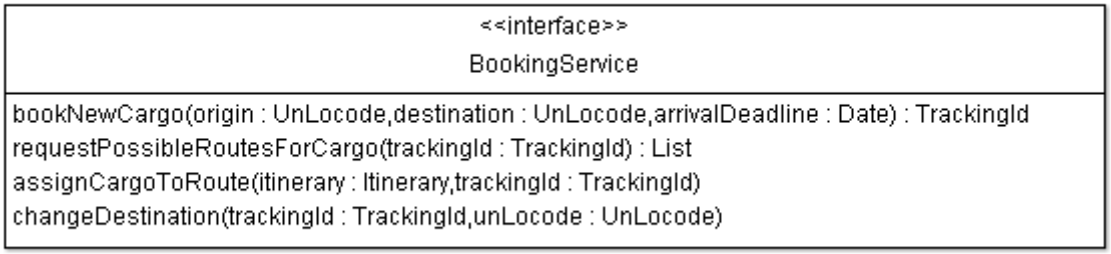
Domain services encapsulate key domain concepts that are not naturally modeled as things. However, domain service method arguments and return values are usually domain classes. Sometimes only the service interface (what the service does) is part of the domain layer, but the implementation (how the service does it) is part of the infrastructure layer. This is analogous to how repository interfaces are part of the domain layer, but the Jakarta Persistence (JPA) implementations are not.

A good example of this is the RoutingService, which provides access to the routing system and is used to find possible routes for a given specification. The implementation, ExternalRoutingService, communicates with another system and translates to/from an external API/data model in the infrastructure layer.

On the other hand, if the service can be implemented strictly using the domain layer, both the interface and the implementation could be part of the domain layer.

Application Services

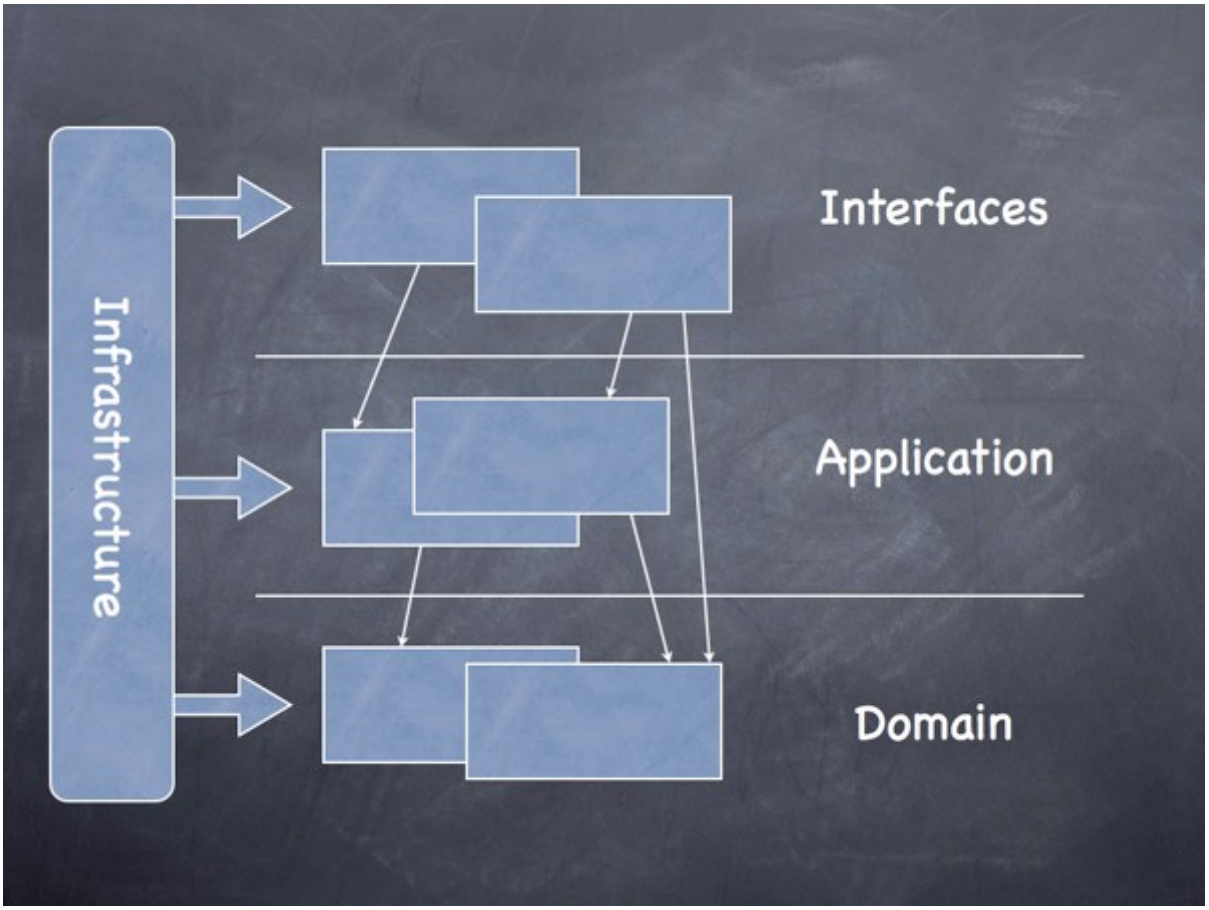
Application services represent the high-level business operations for the system and constitute the application layer. They provide a high-level abstraction for clients to use when interacting with the domain. The org.eclipse.cargotracker.application package contains all the services for the application, such as BookingService and HandlingEventService. The application services are a natural place to apply concerns such as pooling, transactions, and security. This is why application services are typically implemented using Enterprise Beans (EJB) or transactional CDI beans.



In some situations, e.g., when dealing with graphs of lazy-loaded domain objects or when passing services' return values over network boundaries, the services are wrapped in facades. The facades handle ORM session management issues and/or convert the domain objects to more portable Data Transfer Objects (DTOs) that can be tailored to specific use cases. In that case, we consider the DTO-serializing facade part of the interfaces layer. See BookingServiceFacade for an example.

Layers

The Eclipse Cargo Tracker application is layered as illustrated by the following picture.



As you can see, there are three vertical layers: interfaces, application, and domain, each supported by different kinds of infrastructure. In the application, the package naming reflects these layers.

Interfaces

This layer holds everything that interacts with other systems (including humans :-), such as REST web services, web UIs, inbound external messaging endpoints, and batch processes. It handles interpretation, input validation, and translation of incoming data. It also handles serialization of outgoing data, such as HTML or JSON across HTTP to web browsers or web service clients, or DTO classes and distributed facade interfaces for remote clients.

The org.eclipse.cargotracker.interfaces package holds all the interface classes for the application. We have two Web UI interfaces - one for booking and the other for tracking, as well as one REST interface and one scheduled file processor for ingesting handling events. The web interfaces are written in HTML/JavaScript/Jakarta Faces (JSF), the REST interface uses Jakarta REST (JAX-RS), while the file processor uses `@Scheduled`/Jakarta Batch. Modern web application frameworks naturally enforce MVC style separation of concerns, so there is not that much heavy lifting you will need to do. In addition, you can enforce a facade layer if necessary (e.g., your UI team is sufficiently separated from the application services team or you wish to remain strictly agnostic of the UI layer). Otherwise, you can use simple adapters to adapt domain classes to your UI view. We demonstrate both techniques in the application using well-understood existing design patterns

such as Facade, DTO, Assembler, and Adapter (in addition to MVC of course).

Other than Jakarta Faces (JSF) and REST (JAX-RS), other Jakarta EE technologies typically used in the interface layer are Jakarta Security, CDI, Validation, JSON Binding, Batch, and Messaging.

Application

The application layer is responsible for driving the workflow of the application, matching the business use cases at hand. These operations are interface-independent and can be both synchronous or asynchronous. This layer is well suited for spanning transactions, high-level logging, and security.

Note, the application layer is very thin in terms of domain logic - it merely coordinates the domain layer objects to perform the actual work. This is very different from traditional tiered architectures that tend to look more procedural with a lot of business logic in the application layer.

The org.eclipse.cargotracker.application package contains all the application classes. The application layer is typically implemented using Enterprise Beans (EJB) or CDI beans. Other Jakarta EE APIs likely to be used in this layer are Jakarta Security, Transactions (JTA), Concurrency, and Interceptors.

Domain

The domain layer is the heart of the software, and this is where the interesting stuff happens. There is one package per aggregate and each aggregate includes entities, value objects, domain events, a repository interface, and sometimes factories (see the *Characterization* section for details). The aggregate roots are Cargo, HandlingEvent, Location, and Voyage.

The core of the business logic, such as determining whether a handling event should be registered and how the delivery of a cargo is affected by handling, belongs here. The structure and naming of aggregates, classes, and methods in the domain layer should follow the real world as closely as possible. You should be able to explain to a business domain expert how this part of the software works by drawing a few simple diagrams and using the actual class and method names of the source code. Note that it is not uncommon for some domain classes to simply model data and not contain any business logic.

The domain layer is usually implemented primarily using Jakarta Persistence (JPA), Validation, and CDI.

Infrastructure

In addition to the three vertical layers, there is also the infrastructure. As the picture shows, it supports all of the three layers in different ways, sometimes facilitating communication between the layers. In the most common case, the infrastructure layer consists of Repository implementations interacting with a database. In simple terms, the infrastructure consists of everything that exists independently of our application: external/third-party resources, persistence/database engines, messaging back-ends, legacy systems, and so on.

All of the infrastructure classes are in the org.eclipse.cargotracker.infrastructure package. We use Jakarta Persistence (JPA), CDI, Messaging (JMS), and the REST (JAX-RS) client API in this layer. The application will mostly have Jakarta Persistence (JPA) code in this layer encapsulated with thin CDI bean Repository classes. Other Jakarta EE APIs commonly used in this layer include Jakarta Mail.

Are Layers Mandatory?

Very few things in DDD are mandatory, and layering is no exception. You should use layers if you believe they add value in keeping your application maintainable. Indeed, even if you do choose to use layers, you do not need to use them everywhere in your application as an absolute requirement.

If you look closely at the application as well as the opening image in this section, it is perfectly OK to skip layers per use case. For example, it is quite common for the interface layer to skip the application layer and communicate directly with the domain layer. In many other cases, having a layer will clearly add value by separating concerns, improving testability, reducing coupling, and enhancing semantic clarity. A helpful example of this is the BookingService implementation.

Resources

The following are some useful resources for learning more about DDD and Jakarta EE:

- Getting Started with Domain-Driven Design (free DZone Ref Card)
- Domain-Driven Design Quickly (free eBook on InfoQ)
- Domain-Driven Design: Tackling Complexity in the Heart of Software (Eric Evans' DDD book)
- Applied Domain-Driven Design Blueprints for Jakarta EE (slide deck on this project)
- Applied Domain-Driven Design Blueprints for Jakarta EE (video on this project)
- Official Jakarta EE Tutorial
- Official Jakarta EE Examples