INSA Lyon

Computer Sciences Department

PLD AGILE : Optimod'Lyon Technical Documentation

Authors: Hazim Asri
Nihal Boutadghart
Jassir Habba
Ana Martin
Junior Noukam
Simon Perret

Teacher:
Mme. Laforest

December 19, 2024

Contents

Introduction

This technical documentation aims to provide a comprehensive and detailed overview of the **[Optimod'Lyon]** application. It is intended for developers and technical personnel who seek to understand the technical decisions, architecture, and core functionalities of the system, as well as the steps required to configure, use, and maintain the code.

The project follows an **object-oriented development** approach, adhering to the principles of modularity, reusability, and maintainability. We adopted the **Agile Scrum methodology** to ensure iterative and incremental deliveries, allowing for quick adjustments to the needs and challenges encountered. The application is built on a **Model-View-Controller (MVC)** architecture and leverages modern technologies such as **Java**, **React**, and the **Google Maps API**.

This document is structured as follows:

- A description of the architecture and the technical choices made;
- The main components of the system and their responsibilities;
- The processes for installation, configuration, and deployment;
- The tests conducted to validate the application's functionality;
- A section dedicated to best practices for contributors.

With this documentation, we aim to ensure a seamless transition for future teams tasked with maintaining or extending the application, while facilitating onboarding for new contributors.

1 Class and Package Diagrams

1.1 Class Diagram

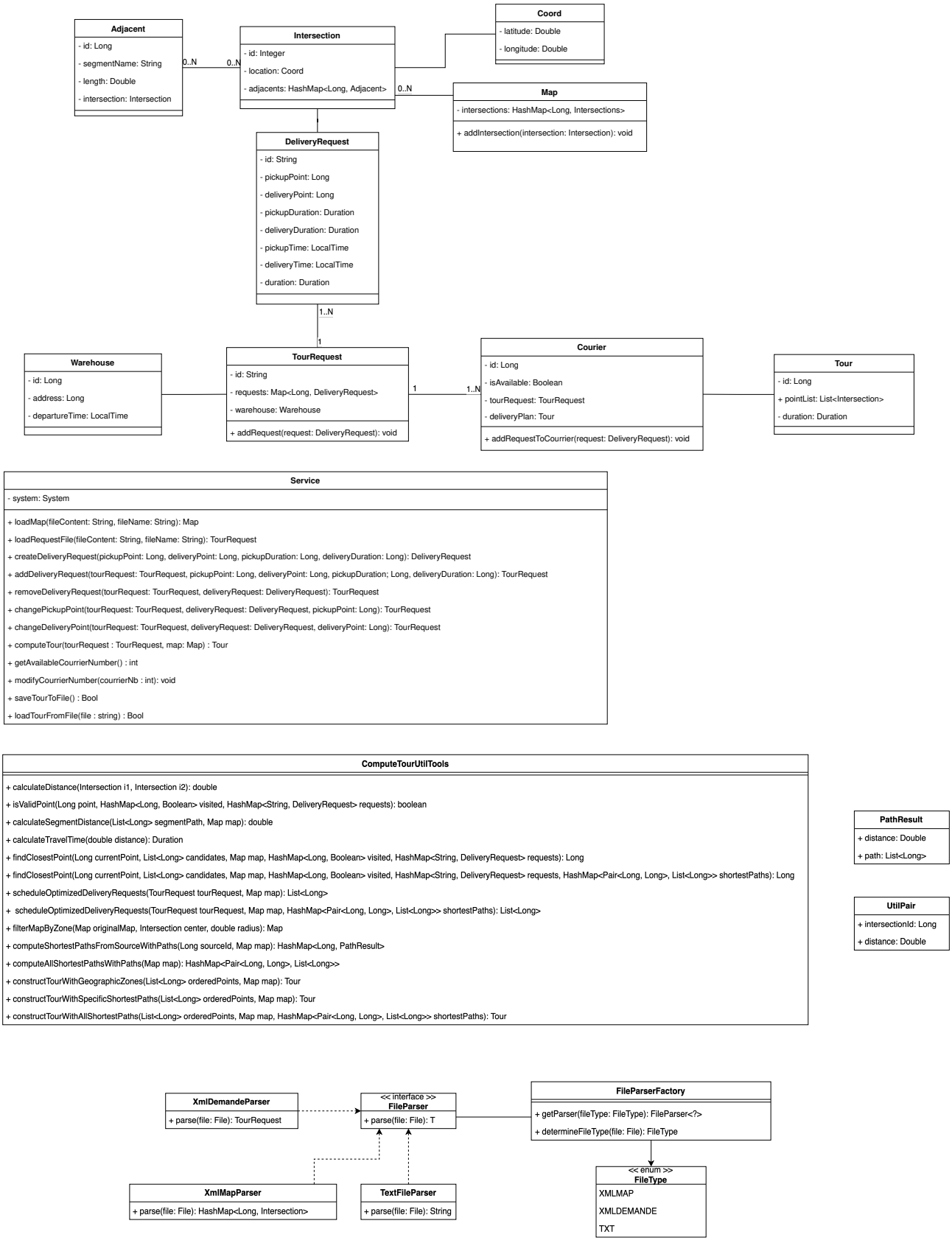

Figure 1: Class Diagram

The class diagram provides an overview of the structure of the system, including key classes, their attributes, and relationships. It reflects the modularity and reusability principles applied during the design phase.

1.2 Package Diagram

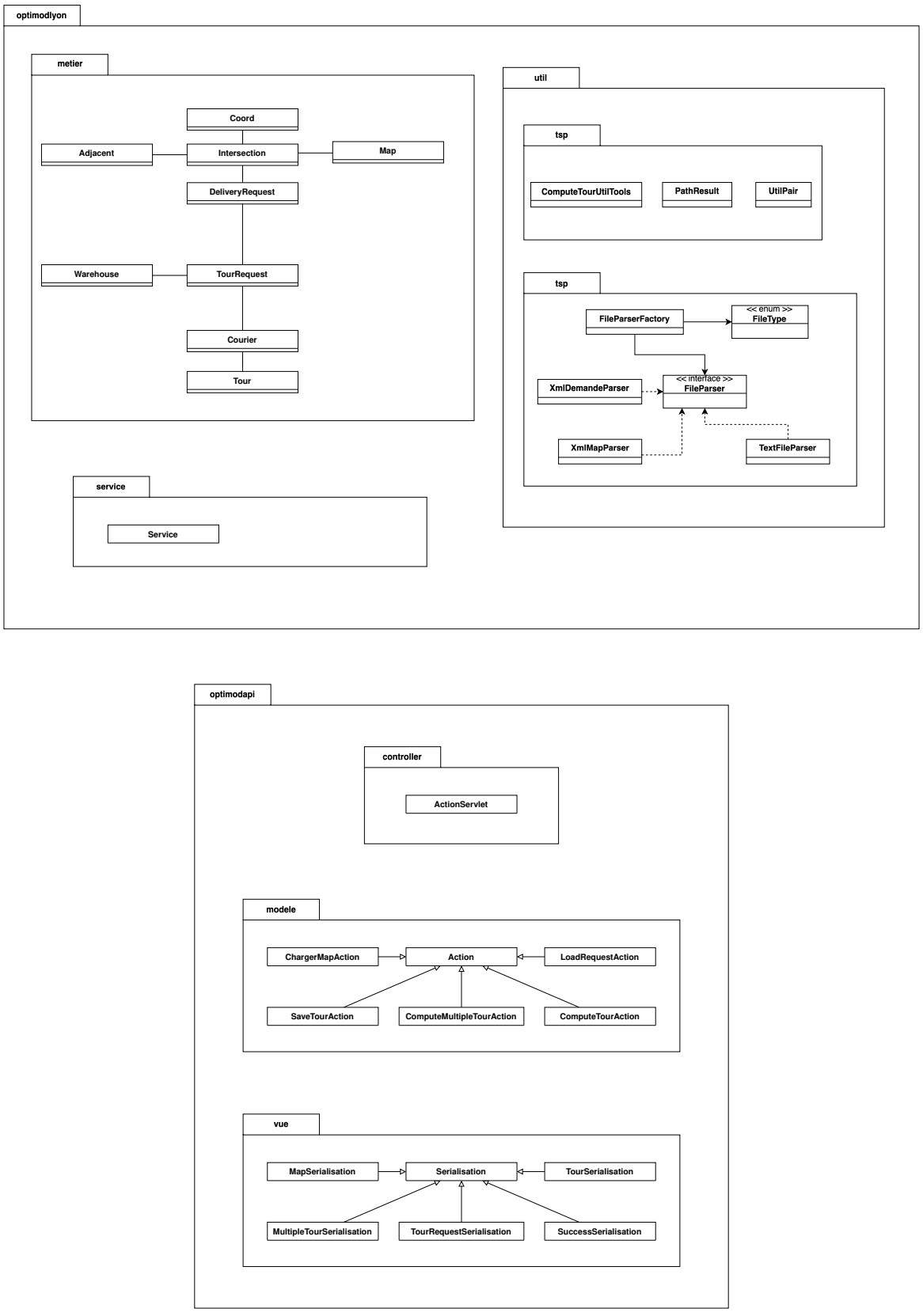

Figure 2: Package Diagram

The package diagram illustrates the organization of the codebase into logical groupings. It showcases the separation of concerns, where each package handles a specific set of responsibilities, ensuring scalability and maintainability.

2 System Architecture

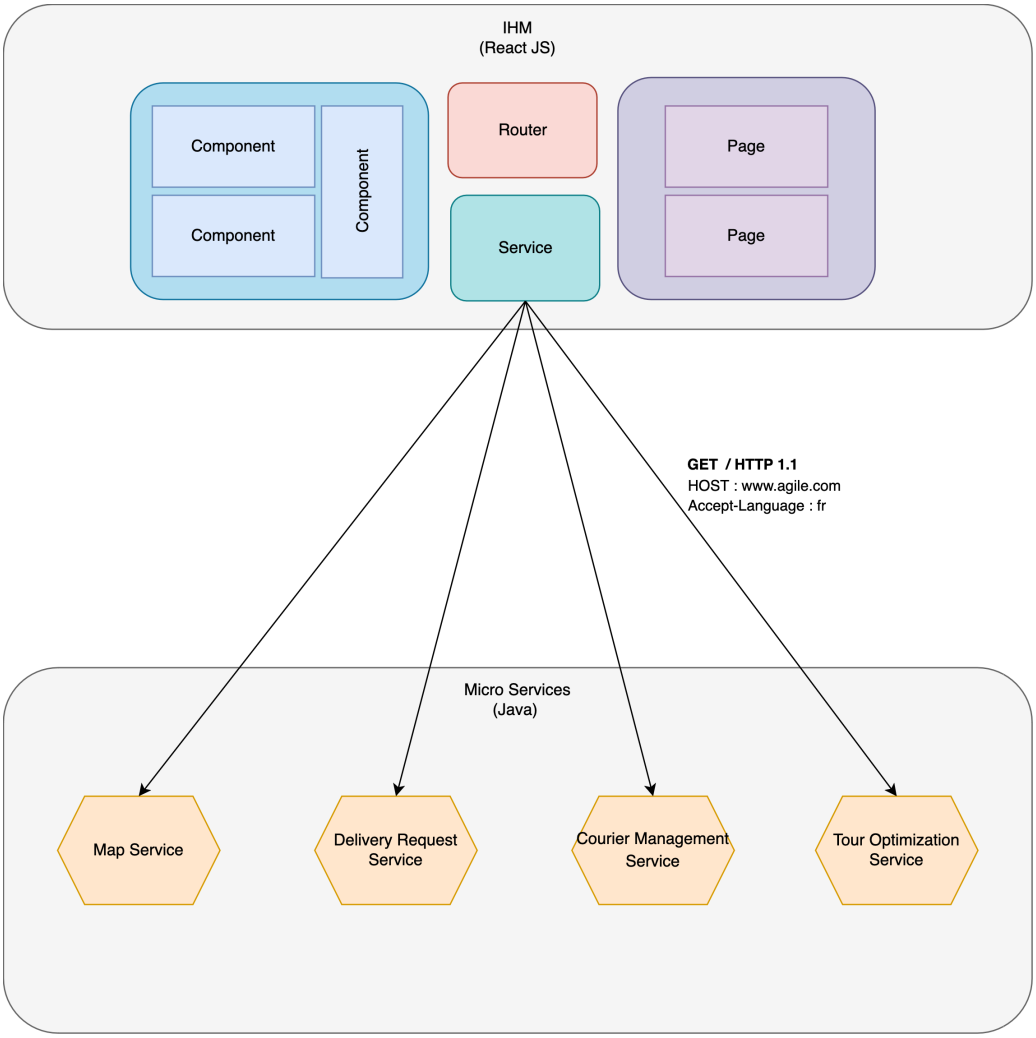

Figure 3: Package Diagram

The system is designed using a **Model-View-Controller (MVC)** architecture. The backend, developed in **Java**, serves as the core of the application, providing APIs to communicate with the frontend. The frontend, developed in **React**, interacts with the Google Maps API to display the network and manage user interactions.

3 Algorithms

3.1 Assigning Tours

- Step 1: Ranking Delivery Requests
  The requests are sorted by increasing distance between the pickup point and the warehouse. This step ensures that the nearest requests are processed first.

- Step 2: Assigning Requests to Delivery Persons
  The sorted requests are assigned in a round-robin fashion to the available delivery persons. Each delivery person receives a TourRequest containing the requests assigned to him.

3.2 Calculating a Tour

- Step 1: Ordering Points
  The tour starts with the warehouse. Then, each point (pickup or delivery) is added respecting:
  - The proximity to the last added point.
  - The constraint that the pickup must precede the corresponding delivery.
  - The tour ends with a return to the warehouse.
- Step 2: Calculating Shortest Paths
  For each successive pair of points (current and next) in the ordered list:
  - The optimal path is calculated using Dijkstra's algorithm
  - All intersections and the total distance between these two points are extracted
- Step 3: Concatenation of Segments
  The calculated segments are concatenated to form a complete grand tour, passing through:
  - The warehouse at the start and end.
  - All pickups and deliveries in between.

3.3 Synthesis

1. Assigning Tours:
   - Sort the requests by proximity to the warehouse
   - Distribute them to the delivery people using the Round-Robin method.
2. Construction of a Tour
   - Order the points to respect the pickup before delivery constraint
   - Calculate and concatenate the paths between each pair of successive points
3. Final Result
   - A complete tour is obtained, ready to be saved or displayed

4 CI/CD Documentation

This pipeline configuration file defines a Continuous Integration and Continuous Deployment (CI/CD) workflow for a Java project using Maven as the build system. It automates essential tasks such as building, testing, optimizing execution time, and ensuring code quality. By automating these processes, the pipeline contributes to maintaining a robust, efficient, and secure delivery workflow. The pipeline is triggered by changes made to the develop branch, ensuring that all updates are validated continuously before being merged into production.

4.1 Workflow Triggers

Triggers determine the conditions under which the workflow is executed. For this pipeline:

- The workflow is automatically triggered on every push to the develop branch.
- This setup ensures continuous validation of new changes, promoting consistent code quality and reducing integration issues.

4.2 Jobs in the Workflow

The workflow is composed of multiple jobs that automate different stages of the CI/CD process.

- **Repository Checkout:** The flow uses the actions/checkout@v4 action to clone the current repository into the workspace. This step ensures that all project files are available for subsequent tasks, such as building and testing.
- **Java Configuration:** The environment is set up using actions/setup-java@v4 to install JDK 11. Maven is configured with dependency caching enabled. This optimization allows the workflow to reuse previously downloaded dependencies, significantly reducing execution time and improving overall efficiency.
- **Build and Test:** Maven is used to build the project and execute all unit and integration tests. Any build failures or test errors immediately halt the pipeline, ensuring that only valid changes progress further.

4.3 Dependency Graph Update

The pipeline generates a dependency graph that maps all project dependencies and their relationships.

- This graph helps identify outdated or vulnerable dependencies more accurately.
- By maintaining an up-to-date dependency graph, the pipeline improves the project's security posture and ensures compatibility with external libraries.

4.4 Benefits of the Pipeline

The CI/CD pipeline provides several advantages:

- **Improved Code Quality:** Automated tests ensure that new code adheres to the project's standards and does not introduce regressions.
- **Efficiency:** Dependency caching and automated processes reduce manual intervention and save time during development cycles.
- **Enhanced Security:** The dependency graph provides better visibility into potential vulnerabilities.
- **Scalability:** The pipeline supports a consistent and repeatable process, making it suitable for teams of all sizes.

This CI/CD pipeline is a cornerstone of maintaining high-quality and efficient continuous delivery in modern software development practices. Next Steps

5 Frontend Tests

During this phase of the project, the frontend was tested manually instead of using automated testing frameworks like Jest or React Testing Library. This decision was made due to the relatively small number of functionalities to validate, which made manual testing more practical and time-efficient.

5.1 Testing Scope

Manual testing was conducted on the following key functionalities:

- **Map Rendering:** Ensuring the map loads correctly with all delivery points displayed as markers.
- **Delivery Point Updates:** Verifying the ability to move pins on the map to modify delivery or pickup points.
- **API Integration:** Ensuring seamless communication with the backend for loading delivery requests and saving updated tours.
- **Edge Cases:** Checking basic error handling, such as invalid inputs or network connectivity issues.

5.2 Limitations of Manual Testing

While manual testing was effective for this phase, it has inherent limitations:

- It is less scalable as the application grows and additional features are introduced.
- Automated regression testing is not possible without proper test cases.
- Ensuring consistency across test runs can be challenging.

5.3 Future Possible Improvements

As the project evolves, transitioning to automated frontend testing is recommended to:

- Reduce the time spent on repetitive tests.
- Increase the accuracy and reliability of testing, especially for edge cases.
- Enable continuous validation of the frontend during CI/CD pipeline execution.