Final design document

Group Name: Here we go again

GitHub repo: https://github.com/alvin159/TrainWeatherInsight

Group members:

Alvin Wijaya

Juhana Kivelä

Yue Zhu

Chenshuo Dong

Table of contents

Contents

1. Description of application

Our application is a train timetable fetching app, augmented with weather data and demographic statistics about departing and arriving cities. At a high level, the architecture adheres to an event-driven design pattern, promoting modularity and scalability. The three key layers are:

1. Frontend GUI: Acts as the user interaction layer, employing a Factory design pattern to dynamically generate and initialize GUI views like HomePage and InformationPage. This ensures flexibility and consistency in UI instantiation and updates.

2. Middle-layer Message Broker: A Singleton instance manages centralized communication between the frontend and backend components. This Singleton ensures a single point of truth for event-driven messaging, reducing complexity and ensuring state consistency. The main files for this are MessageBroker.java and MessageCallback.java.

3. Backend Processing Components: Operate independently to handle specific functionalities such as train data retrieval, station searches, town demographics, and weather information. This independence supports horizontal scaling and simplifies future maintenance. These independent components receive requests from frontend, process the request and return response back to the frontend. These are TrainComponent.java, SearchStationComponent.java, TownComponent.java and WeatherComponent.java

1.2 Design Patterns

- **Singleton in Message Broker**: The MessageBroker is implemented as a Singleton to ensure that all communication between components is routed through a single, centralized instance. This design avoids synchronization issues, provides thread safety, and ensures a consistent communication pathway. The centralized MessageBroker eliminates direct dependencies between components, thereby enforcing loose coupling.

- **Factory Model for UI**: The application employs a Factory pattern to create and manage UI components. For example, the UIViewFactory dynamically generates views for HomePage and InformationPage based on user interaction or application state. This design enhances code reusability and reduces the risk of errors from manual instantiation of UI elements.

1.2 Reasoning for Design Solutions

High-Level Architectural Choices

The event-driven architecture was chosen for its ability to decouple components and facilitate asynchronous processing. This approach enhances responsiveness in the frontend, as tasks in the backend are handled in separate threads. By centralizing event handling in the MessageBroker, the system becomes more maintainable, and debugging is simplified.

Why Singleton for Message Broker?

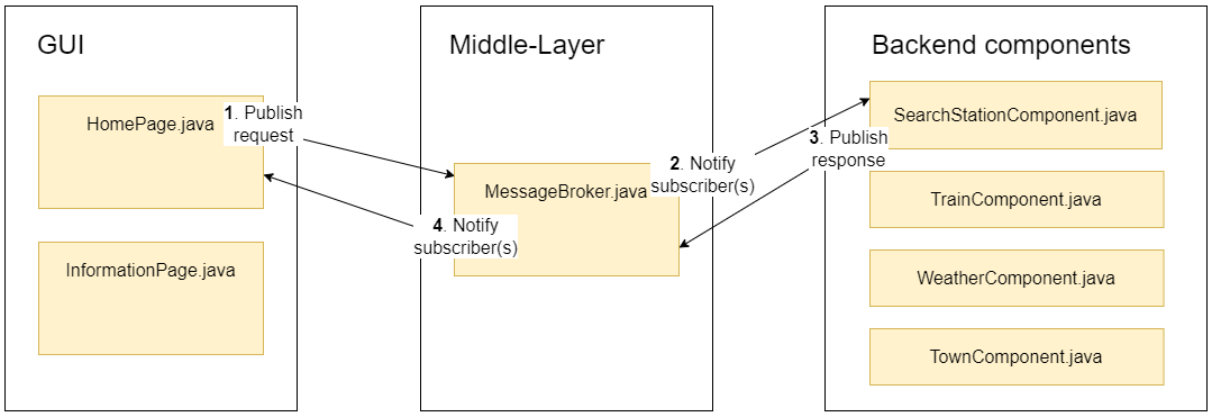
The decision to use the Singleton pattern for the MessageBroker stems from the need for a single, globally accessible instance to mediate communication. This prevents issues such as duplicate messages or conflicting states arising from multiple brokers. Additionally, the Singleton ensures that event registration and dispatching remain consistent across all components.

Why Factory for UI?

A Factory pattern was introduced in the UI to standardize the creation of views. Instead of hardcoding view creation within the frontend, the UIViewFactory takes parameters to generate the necessary view. This design allows for easy extensions—for instance, adding a new page for future features—without modifying the core logic.

Backend Design and Modularity

Backend processing components derive from the BackendComponent abstract class, ensuring uniformity in their implementation. Common methods like initialize, shutdown, and handleEvent provide a consistent interface, simplifying integration and reducing bugs. The modularity also allows backend components to be converted into standalone microservices in the future if the application grows.



Picture 1. High-level architecture of the application

Our application follows event-driven architecture, where Events can be Requests or Responses e.g. SEARCH_STATION_REQUEST and SEARCH_STATION_RESPONSE. In Picture 1 there is an example request + response communication between frontend and backend.

1.3 Components and their responsibilities

Component name	Type	Responsibility of the component
HomePage.java	Application	Main entry point of the program GUI. User can select the stations and day that (s)he wishes to see trains routes.
InformationPage.java	Application	Display the train arrival and departure stations, timetable, weather conditions, and information about the departure and arrival towns to the user.
MessageBroker.java	Class	Receives Events from frontend views and backend components and forwards the Events to correct subscribers.
SearcStationComponent.java	Class	Responsible for searching train stations based on users current text on search field. Responds a list of stations that can be shown to user.

TrainComponent.java	Class	Fetches the information of trains between departure and arrival stations.
TownComponent.java	Class	Responsible for fetching data (Population, Area, Density) about town/city and sending it back to frontend in a proper data structure.
WeatherComponent.java	Class	Get weather data from the API and convert it into a format that can be use by the program in GUI.

Table 1, The most important classes and interfaces, and their responsibilities.

1.4 Information flow between components

On frontend, views can create other views and initialize them directly. This allows fast and seamless communication between them. When frontend needs to fetch something from backend, they must send a new Event request to message broker that then redirects the Event to the correct backend processing component. These Event requests are not networking requests, but just direct calls to the message broker. This Event must contain a Payload, which contains all the information that is required for the Event to be processed successfully.

For example, if frontend needs to fetch train timetables between Helsinki and Rovaniemi on 30.10.2024, then they send an event request TRAIN_REQUEST with a following payload

```
public Payload(Date departingDate, String departureStationShortCode,
                            String arrivalStationShortCode) {
    this.departingDate = departingDate;
    this.departureStationShortCode = departureStationShortCode;
    this.arrivalStationShortCode = arrivalStationShortCode;
}
```

When backend processing component receives the Event, it extracts the data from Payload and does the business logic processing. After that, it will create new Event with response Payload to message broker, which will then further direct the Event to correct frontend view.

1.5 More detailed structure of components

The application's architecture ensures modularity, reusability, and adherence to standardized practices. This is achieved using design patterns like Singleton and Factory, along with abstract classes and interfaces that enforce consistency in implementation. Additionally, all classes are thoroughly documented using Javadoc, providing clear documentation for developers. This ensures that the codebase is not only easy to understand but also facilitates effective collaboration and maintenance. The docstrings can be seen by opening the files, or by generating an easy-to-read report with command “mvn clean javadoc:javadoc”.

Frontend

The frontend uses a **Factory Model** to dynamically create UI components, ensuring a consistent and modular approach. A UIViewFactory generates and initializes views like HomePage and InformationPage based on user navigation or interaction. This design promotes extensibility, making it easier to add new pages or modify existing ones without significant code changes.

Message Broker

The **MessageBroker** acts as the core communication hub between the frontend and backend. Implemented as a **Singleton**, it ensures that only one instance exists throughout the application's lifecycle. This centralized approach eliminates synchronization issues and enforces loose coupling between components. All communication between the layers is handled through the onMessageReceived() callback method defined in the MessageCallback interface. This interface is essential for enabling seamless integration between the frontend and backend.

Backend

All backend processing components derive from the abstract class BackendComponent.java, which provides a standardized framework for backend functionality. This class includes the following key methods:

- **handleEvent()**: Processes events dispatched from the MessageBroker.

- **initialize()**: Prepares the component for operation, ensuring all dependencies and configurations are properly set up.
- **shutdown()**: Cleans up resources when the component is no longer needed.
- **getPayload()** *(optional)*: Allows components to retrieve and manage payload data in a structured manner.

By inheriting from this class, backend components maintain a uniform structure, which simplifies their integration and testing.

Key backend components include:

- **TrainComponent.java**: Handles train schedule retrieval and processing.
- **SearchStationComponent.java**: Searches for train stations based on user input.
- **TownComponent.java**: Fetches demographic data for cities (population, area, density).
- **WeatherComponent.java**: Retrieves and formats weather data for the application.

The MessageCallback.java interface plays a critical role in both frontend views and backend components by facilitating communication with the MessageBroker. It defines the **onMessageReceived()** function, which enables components to react to events efficiently.

This architecture ensures that backend components are independent, loosely coupled, and adhere to predefined patterns, making them easier to test, maintain, and extend.

2. Self-evaluation

We have successfully adhered to the original design pattern since we have certain guidelines on how to implement components and how the information is sent and received via Events and message broker. This common approach has helped the overall application structure and information handling as components must be implemented in a certain way to function properly. This approach ensure that our implementation aligns with the planned architecture and maintains consistency throughout the development process.

The usage of factory pattern for UI creation was the only thing we didn't plan at the start of the project. We only saw the need for such a pattern later on in the development phase and at that moment it was decided that the application will be better with that implementation.

3. Reasoning for design solutions

On a higher level, we wanted to create independent backend components that have pre-defined message receiving and sending. This makes the information flow easier to manage as developers can easily see what the backend component inputs and what it outputs back to frontend. We did not want to create separate services for backend components as it would be overkill for such a simple application. Our approach however supports that option if in the future if it was decided that the services should be run server-side and run separately in a Docker container for example. We still wanted the frontend to run smoothly and not freeze even if backend components take longer time to process Events. This was achieved by multithreading on the backend components.

We also wanted centralized messaging component for this application so that all the traffic between frontend and backend would go through this component. We ended up creating our own message broker that is able to direct the Events to correct components. This was selected for its simplicity and to achieve the freedom to customize it. There were other options like some sort of gateway or interface, but this seemed like the best choice. This way we were also able to follow the basic guidelines of event-driven architecture.

On the backend side, we wanted common static class that all backend processing components must use. This was selected so that all backend components follow the similar pattern and don't freestyle their implementation.

We also wanted pre-defined event types and payloads for each event so we created Events.java where everything is in a centralized location. This was selected instead of custom strings as Event names and custom payload structure because this approach adds clarity and simplifies the event sending and receiving.

The project utilizes several external libraries to enhance its functionality. JavaFX (org.openjfx:javafx-controls:19) is used for building the GUI components. Gson (com.google.code.gson:gson:2.9.0) provides JSON serialization and deserialization capabilities, while OkHttp (com.squareup.okhttp3:okhttp:4.2.0) serves as the HTTP client for API calls. JSON-Smart (net.minidev:json-smart:2.5.1) is a lightweight library for JSON processing. For testing, JUnit 5 (org.junit.jupiter:junit-jupiter-api and junit-jupiter-engine:5.9.3, scope: test) is

employed as the primary testing framework, complemented by Mockito (org.mockito:mockito-core:4.11.0, scope: test) for mocking dependencies during unit tests.

Name	Responsibility
Alvin Wijaya	Integrated the Weather API for seamless data retrieval and display, prototype Figma design, and UI implementation
Juhana Kivelä	Architectural design and modifying project to follow it, message broker, events, backend components structure, UI design & implementation, test cases
Yue Zhu	Design of data structure for different components, implementation related to Demographic component in backend and frontend, test cases
Chenshuo Dong	Front-end function implementation, back-end search train API data acquisition, conversion, storage, and front-end display.

Table 1. Distribution of Responsibilities

4. Use of AI

ChatGPT was utilized to enhance documentation and assist in resolving bugs. One of the prompts is “The output of my code is incorrect. Can you analyse this Java function and suggest corrections?”, “Ensure the verb tenses and grammatical structures are consistent throughout this text”, "Can you rewrite this text to make it simpler and easier to understand?" When the efficiency of a certain function of the program is too low after implementation, I will use ChatGPT to optimize this function.

GitHub Copilot was used to brainstorm and test mock implementations for message broker and backend components as well as their communication. The code was not copied directly to the implementation, but certain ideas like the usage of callback functions were gotten through that process. It helped us to think of different solutions and weight their pros and cons.

GitHub Copilot was also used to auto-complete singular lines in the code. This was done with the routine programming to make the development faster. It didn’t contribute to the design of the code, but just similar to VS Code IntelliSense or IntelliJ Code Completion it predicts the rest of the line and suggests that.

GitHub Copilot was used to create docstrings for classes. The docstrings were checked by developers and oftentimes modified to describe the class better. This made the development faster and improved the accuracy of documentation.