Mid-term 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

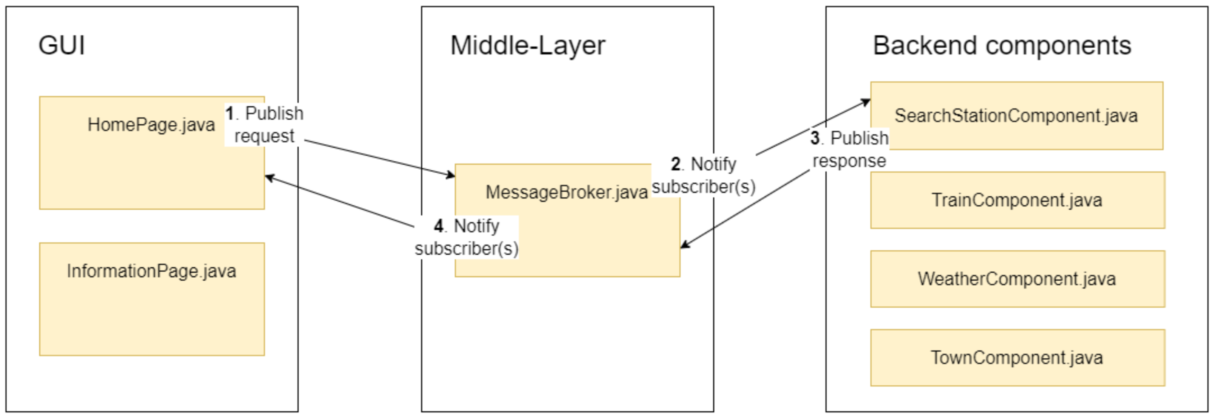
1. Description of application

Our application is a train timetable fetching app, that has extra features like weather information and interesting statistics about the departing and arriving cities. On the high-level, our application consists of 3 parts:

1. Frontend GUI. We have currently 2 frontend views that handle displaying correct data to user. These views are HomePage.java and InformationPage.java.

2. Middle layer message broker. This works as a centralized communication path between frontend and processing components. The core files for this are MessageBroker.java and MessageCallback.java.

3. Backend processing components. 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



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.

1.1 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.

1.2 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.

1.3 More detailed structure of components

All backend processing components use an abstract class BackendComponent.java, which contains the core functionalities for the backend components to function correctly. These include handleEvent(), initialize(), shutdown() and getPayload(). Only getPayload() is optional and backend components can use it if they wish to. Other methods are abstract and backend component must implement them.

MessageCallback.java interface is the key file for frontend views and BackendComponent.java to implement. It allows the receiving of events from message broker with its function onMessageReceived().

2. Self-evaluation

So far, a clear design of the application has supported the implementation of it 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.

Also we've seen reduction in spaghetti code as components can't call each other directly as they want to. This in turn has helped the components to stay independent without having any dependencies to each other.

We have been sticking to our original plan quite well. The only thing that has changed is that one backend processing component was removed (Abbre.vConvertComponent.java) and one was added (SearchStationComponent.java). Also we had to change the weather API from OpenWeather to Visualcrossing Weather as the OpenWeather free version did not support getting weather prediction.

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.