Relazione Tecnica

PAGINA 1 - FRONTESPIZIO

Titolo progetto: Damose - Rome Transit Tracker

Anno accademico: 2024/2025

Docente: Mattia Samory

Sviluppatore: Giorgio Caruso

Matricola: 2212508

Video Introduttivo (Molto consigliato per l'esperienza completa):
https://youtu.be/WSMEx0U1YT8

1. Introduzione

La consegna era un'app di Tracciamento di autobus e di aiuto per la mobilità a Roma. Penso di essere riuscito a raggiungere un risultato più che ampio con un bellissimo stile e il mio tocco personale.

Requisiti funzionali principali:

- Ricerca fermate per nome/codice e apertura del dettaglio arrivi.
- Ricerca linee e visualizzazione del percorso su mappa con gestione direzione.
- Visualizzazione mezzi in tempo reale su mappa (quando disponibili).
- Predizione arrivi alla fermata combinando dati statici e realtime.
- Gestione preferiti (fermate e linee) con persistenza locale per utente.
- Login/registrazione utente con database locale.
- Switch online/offline con continuità operativa dell'app.

Requisiti non funzionali:

- Prestazioni: aggiornamento realtime periodico a 30 secondi, parsing e rendering ottimizzati (cache feed deduplica e aggiornamenti incrementali).
- Robustezza: gestione errori rete/parsing con fallback operativo, controllo null e validazione input GTFS.

- Usabilità: UI Swing con controlli immediati, pannelli contestuali, scorciatoie tastiera nella ricerca, feedback visivo stato connessione.

Vincoli di progetto rispettati:

- Java 17+ (Maven compiler source/target 17).
- Interfaccia realizzata in Swing (con FlatLaf).
- Rate limit API: 1 chiamata ogni 30 secondi (RT_UPDATE_INTERVAL_MS = 30000).
- Supporto offline tramite dataset GTFS statico locale e logica di calcolo arrivi senza feed realtime.

3. Feature implementate

3.1 Mappa funzionalità richieste -> funzionalità realizzate

Feature richiesta	Stato (Completata/Parziale/Non fatta)
Ricerca fermate	Completo
Ricerca linee	Completo
Visualizzazione mappa	Completo
Aggiornamento realtime (30s)	Completo
Modalità offline	Parziale (Vedere Nota sottostante).
Predizione arrivi	Completo
Preferiti	Completo

Autenticazione	Completo
Qualità del servizio/dashboard	Completo

Nota: Nonostante ho iniziato a scrivere del codice per implementare le tiles offline da uno zip ti openstreetmap sarebbe impossibile implementare la mappa offline senza caricare un numero di tiles esponenziale e aumentare di Gigabyte la grandezza dell'applicazione e anche per questo ho preferito non farlo e lasciare la funzionalità online solo parziale:

Se offline si usano le tiles in cache, se però l'app viene inizializzata offline niente da fare.

3.2 Livello raggiunto

- Avanzato

Ho raggiunto tutte le feature richieste per raggiungere il grado 27-30 da dev singolo. Anche se in realtà ho implementato anche le feature richieste per il livello avanzato dei gruppi.

4. Architettura software

Panoramica ad alto livello:

Architettura MVC/layered:

- model: entità dominio (Stop, Trip, StopTime, VehiclePosition, ecc.).
- view: GUI Swing, overlay, pannelli e rendering mappa.
- controller: orchestrazione casi d'uso e flussi UI.
- service: logica applicativa (arrivi, realtime, preferiti, qualità servizio).
- data.loader e data.mapper: ingestione GTFS statico e matching dati realtime.
- database: persistenza utenti/sessione/preferiti.

Moduli principali:

- MainController: coordinatore centrale.
- ControllerDataLoader + ControllerDataContext: caricamento statico e composizione dipendenze.
- RealtimeService + RealtimeUpdateScheduler: fetch/polling feed e cicli di aggiornamento.
- RouteService, ArrivalService, TripMatcher, StopTripMapper: servizi core di dominio.
- MainView, MapOverlayManager, SearchOverlay, FloatingArrivalPanel, RouteSidePanel: UI e interazione.

Flussi principali:

- Caricamento dati statici: all'avvio vengono letti stops, trips, stop_times, shapes, routes, calendar_dates; poi vengono costruiti mapper/servizi e iniettati nel controller.
- Aggiornamento realtime: polling periodico dei feed vehicle_positions e trip_updates, parsing protobuf, aggiornamento arrivi realtime e posizioni veicoli.
- Aggiornamento UI: tutte le modifiche Swing avvengono su EDT (SwingUtilities.invokeLater), con refresh mappa e pannelli contestuali.
- Fallback offline: se i feed non sono disponibili in avvio o in tentativo di switch, il sistema imposta modalità offline, interrompe polling realtime e continua con funzionalità statiche (arrivi programmati, ricerca, navigazione).

5. Decisioni progettuali

Decisione 1: separare caricamento dati e controller principale.

- Problema: rischio di avere MainController troppo grande e poco manutenibile.
- Alternative considerate: caricamento diretto nel controller; layer dedicato.
- Scelta finale e motivazione: ControllerDataLoader + ControllerDataContext per isolare ingestione e wiring.
- Trade-off: più classi da gestire, ma maggiore chiarezza e testabilità.

Decisione 2: polling realtime a intervallo fisso di 30 secondi.

- Problema: rispettare il vincolo API e mantenere dati aggiornati.
- Alternative considerate: polling più frequente; refresh manuale.
- Scelta finale e motivazione: timer periodico a 30s in RealtimeService/RealtimeUpdateScheduler.
- Trade-off: possibile latenza fino al ciclo successivo, compensata da stabilità e conformità al rate limit.

Decisione 3: matching robusto tra trip statici e realtime.

- Problema: mismatch fra identificativi nei feed.
- Alternative considerate: solo match esatto; normalizzazione con varianti.
- Scelta finale e motivazione: TripIdUtils, TripMatcher, ArrivalMatchingUtils con varianti e fallback.
- Trade-off: logica più complessa, ma migliore accuratezza.

Decisione 4: separare i flussi UI in componenti controller dedicate.

- Problema: gestione complessa di route panel, stop panel e veicolo seguito.
- Alternative considerate: tutto in MainController; flussi specializzati.
- Scelta finale e motivazione: RoutePanelFlow, StopPanelFlow, VehicleFollowFlow, RouteViewportNavigator.
- Trade-off: più oggetti da coordinare, ma migliore SRP e manutenzione futura.

6. Design pattern adottati

- MVC: applicato tra model, view, controller; beneficio principale: separazione responsabilità.
- Facade: ControllerDataLoader espone un punto unico di inizializzazione; beneficio: riduzione complessità.
- DTO: ControllerDataContext aggrega dati e servizi; beneficio: passaggio strutturato delle dipendenze.
- Singleton: ServiceQualityTracker e SessionManager; beneficio: accesso centralizzato a stato globale.
- Factory/Simple Factory: MapFactory (creazione mappa) e WindowControlButtonFactory; beneficio: costruzione coerente e riuso.

- Observer/Listener: listener Swing e callback setOn..., oltre a RealtimeService.setOnDataReceived; beneficio: disaccoppiamento eventi-logica.
- Strategy (via composizione): RouteVehicleMarkerBuilder con interfaccia RouteLookup; beneficio: sostituibilità della strategia di risoluzione route.
- Service Layer: servizi dedicati (ArrivalService, RouteService, RealtimeService, FavoritesService); beneficio: logica riusabile e indipendente dalla UI.

7. Paradigmi (generico, funzionale, stream) (obbligatorio)

7.1 Programmazione generica

Nel codice sono usate collezioni tipizzate e API generiche in modo esteso: List<Stop>, List<Trip>, Map<String, List<GeoPosition>>, Map<String, Map<Integer, String>>, callback tipizzate con Consumer<Stop> e Supplier<ConnectionMode>.

7.2 Programmazione funzionale

Sono usate lambda e interfacce funzionali in listener UI, scheduler e flussi controller. Esempi: callback setOn..., method reference (RoutesLoader::getRouteById), uso di Supplier/Consumer nel ciclo realtime. Sono presenti anche scelte di immutabilità locale (List.copyOf, snapshot list).

7.3 Stream API

Le Stream API vengono usate per filtri, mapping, ordinamenti e raccolte. Esempi principali:

- RouteService: filtri su direzione/headsign, distinct, sorted, collect.
- ArrivalService: ordinamento e formattazione arrivi con pipeline stream.
- LineSearchDataBuilder: costruzione lista linee con map/sorted/toList.
- ServiceQualityPanel: statistiche min/max su serie con mapToInt.

8. Gestione dati GTFS statico/realtime e modalita online/offline

Parsing GTFS statico:

I file statici sono inclusi in src/main/resources/gtfs_static e caricati da loader dedicati (StopsLoader, TripsLoader, StopTimesLoader, RoutesLoader, ShapesLoader, CalendarLoader). I dati vengono poi indicizzati/ mappati da StopTripMapper e TripMatcher.

Parsing GTFS Realtime:

I feed protobuf vehicle_positions e trip_updates vengono scaricati periodicamente e parsati da GtfsParser.

Durante il parsing vengono applicate normalizzazioni sugli ID e filtri sulle entità non valide.

Strategia di cache/simulazione durante sviluppo:

Il sistema mantiene in memoria gli ultimi feed realtime e usa cache per evitare parsing ridondante quando il timestamp del feed non cambia. La mappa usa cache locale delle tile OSM. In assenza feed realtime, la logica opera con dati statici locali.

Switch online/offline e gestione errori feed:

All'avvio viene verificata la disponibilità dei feed. Se non disponibili, l'app passa in offline e continua a funzionare con schedule statica. Anche durante l'uso, in caso di errore connessione, il passaggio a offline mantiene operativa ricerca, mappa e pannelli.

Politica di refresh (30 secondi):

La frequenza di aggiornamento è fissata a 30 secondi, in linea con il vincolo del progetto.

9. Interfaccia utente

Scelte UX/UI principali:

Interfaccia desktop Swing con tema dark, mappa centrale e overlay contestuali. La navigazione è orientata a operazioni rapide: ricerca, selezione, filtro e ispezione dettagli.

Principali schermate e comportamento:

- Schermata principale con mappa interattiva (pan/zoom).
- Overlay di ricerca con modalità Fermate, Linee, Preferiti.
- Pannello flottante fermata con arrivi imminenti e vista completa delle corse giornaliere.
- Pannello laterale linea con fermate, direzione e marker veicoli.
- Pannello qualità servizio con stato sintetico e trend.

Interazioni chiave:

- Ricerca live con tastiera (Tab, Enter, Esc).
- Click su fermata per centrare mappa e aprire arrivi.
- Selezione linea per mostrare percorso e filtrare veicoli.
- Selezione veicolo per follow su mappa e dettaglio dedicato.
- Pulsante connessione per switch online/offline con stato grafico.

10. Testing e qualità del codice

Strategia di test:

Sono presenti test unitari JUnit 5 focalizzati su modello, mapper, servizi e utility geografiche. Non sono

presenti test di integrazione UI end-to-end.

Copertura funzionale dei test:

I test coprono:

- Modelli (Stop, Trip, StopTime, TripServiceCalendar, VehicleType).
- Utility mapping (TripIdUtils).
- Servizi (RouteService, caso specifico ArrivalService).
- Utility mappa (GeoUtils).
-

Pratiche di qualità:

La qualità è supportata da modularità per package, separazione responsabilità, refactoring orientato SRP, aggiornamenti UI thread-safe su EDT, gestione dipendenze con Maven e documentazione API generata.

11. Ruoli e contributi

11.1 Ruoli assegnati

Membro	Ruolo(i)
Giorgio Caruso	Backend, Frontend , Tester

12. Risorse esterne e integrazione:

- Nome risorsa: Open Data Roma Mobilità (feed GTFS Realtime) Link: https://romamobilita.it/ Scopo d'uso: sorgente dati realtime per posizioni veicoli e aggiornamenti corse Parte del sistema: AppConstants, RealtimeService, GtfsParser, scheduler/controller realtime
- Nome risorsa: GTFS Realtime bindings Link: https://github.com/MobilityData/gtfs-realtime-bindings Scopo d'uso: parsing del formato GTFS-RT protobuf Parte del sistema: servizi realtime e parser feed
- Nome risorsa: Protocol Buffers Java Link: https://github.com/protocolbuffers/protobuf Scopo d'uso: deserializzazione payload protobuf Parte del sistema: stack di parsing realtime
- Nome risorsa: JXMapViewer2 Link: https://github.com/msteiger/jxmapviewer2 Scopo d'uso: componente mappa, pan/zoom, overlay painter Parte del sistema: MapFactory, MapOverlayManager, renderer mappa
- Nome risorsa: OpenStreetMap Link: https://www.openstreetmap.org Scopo d'uso: tiles cartografici Parte del sistema: factory mappa e cache tile locale
- Nome risorsa: FlatLaf Link: https://www.formdev.com/flatlaf/ Scopo d'uso: look & feel moderno per UI Swing Parte del sistema: inizializzazione tema in MainView
- Nome risorsa: SQLite JDBC Link: https://github.com/xerial/sqlite-jdbc Scopo d'uso: persistenza locale utenti e dati associati Parte del sistema: DatabaseManager, UserService, gestione sessione/preferiti

- Nome risorsa: Maven Link: https://maven.apache.org/ Scopo d’uso: build automation, gestione dipendenze, esecuzione test Parte del sistema: pom.xml, ciclo di build/test/package

12.1 Uso di strumenti AI

Non nascondo di aver fatto molto utilizzo di AI per questo progetto essendo la prima volta che mi affacciavo ad un progetto di queste dimensioni ed importanza. La mia strategia nel loro utilizzo era:

1. Ho un'idea sul come fare una classe

2. Lascio un AI generare la struttura della classe

3. Faccio test, aggiusto bug e aggiungo metodi e piccole rifiniture perché capisco come farla funzionare alla perfezione.

Tutte le modifiche sono state verificate, adattate e validate manualmente nello specifico contesto del progetto. Essendo il mio primo progetto mai fatto e soprattutto il mio primo lavoro in Java mi sono potuto facilitare molto con l’utilizzo di IA per conoscenza generale di utilizzo di metodi e sintassi e analisi dei dati con il quale stavolavorando. Sono molto contento di come son riuscito a lavorare utilizzando questo potente strumento.

14. Conclusioni

Sono estremamente contento del lavoro svolto e penso che sia un lavoro eccellente. Questo progetto mi ha aiutato a capire come un dev lavora e quali sono i suoi strumenti più utili (GitHub, Uso del Terminale...). Non vedo l'ora di iniziare un nuovo progetto e mettere in pratica tutte le conoscenze che sono riuscito ad apprendere lavorando a questo.