Università degli Studi di Salerno
Corso di Ingegneria del Software

System Design Document - HeavyRoute

Versione 1.0

Informazioni Generali

Corso di Laurea:	Informatica
Università:	Università degli Studi di Salerno (UNISA)
Docente:	Chiar.mo Prof. DE LUCIA Andrea
Data:	28/10/2025
Anno Accademico:	2025/2026

Membri Del Gruppo

MANFREDINI Umberto	Matricola 0512119797
MANZO Ugo	Matricola 0512119071 *(Coordinatore)*
ROMANO Pino Fiorello	Matricola 0512120259

Revision History

Data	Versione	Descrizione	Autore
17/11/2025	1.0	Creazione del SDD	Pino Fiorello Romano
—	—	—	—
–	–	–	–

Indice

1. Introduction

1.1. Purpose of the system

Lo scopo di questo documento è definire e descrivere in dettaglio l'architettura software del sistema *HeavyRoute*. La piattaforma è progettata per digitalizzare, automatizzare e ottimizzare l'intero processo di gestione dei trasporti eccezionali per l'azienda Logistica Mediterranea S.P.A.
Il sistema sostituirà le attuali procedure manuali, frammentate e soggette a errori, con un flusso di lavoro digitale, centralizzato e tracciabile. Questo sarà realizzato attraverso un'architettura a 3 livelli (3-Tier) che comprende un'applicazione web per il personale di back-office e i committenti, un'applicazione mobile per gli autisti, un backend applicativo che gestisce tutta la logica di business e un DBMS per gestire la persistenza dei dati.
Il System Design Document (SDD), pertanto, funge da guida tecnica per lo sviluppo, illustrando le scelte architetturali, la scomposizione del sistema in componenti, le tecnologie adottate e le strategie implementative che garantiranno il raggiungimento degli obiettivi di progetto.

1.2. Design goals

L'architettura del sistema *HeavyRoute* è guidata da una serie di obiettivi di progettazione derivati direttamente dai requisiti non funzionali (NFR) definiti nel Requirements Analysis Document. Ogni scelta progettuale deve contribuire a soddisfare i seguenti criteri qualitativi.

- **Progettare per l'Usabilità (NFR1, NFR2):** L'architettura deve supportare la creazione di interfacce utente distinte e ottimizzate per ogni ruolo. In particolare, è cruciale sviluppare un'interfaccia *mobile-first* per l'autista, che sia sicura e semplice da usare, e interfacce web più complesse per il personale di back-office. L'intero sistema sarà progettato per supportare nativamente la localizzazione in lingua italiana.
- **Garantire Alte Prestazioni (NFR3, NFR4, NFR5, NFR6):** Il design deve mirare a minimizzare la latenza e in condizioni di rete standard non deve superare i 3 secondi per il caricamento. L'architettura deve essere ottimizzata per garantire tempi di risposta dell'interfaccia utente inferiori a 1 secondo. Le scelte tecnologiche per il backend e le notifiche devono permettere la gestione di eventi critici in tempo quasi reale (entro 10 secondi). La logica di business, come il calcolo dei percorsi, deve essere efficiente (sotto i 20 secondi) e l'infrastruttura deve essere dimensionata per sostenere il carico di almeno 50 utenti concorrenti.

- **Assicurare Massima Affidabilità (NFR7, NFR8, NFR9, NFR10):** La continuità del servizio è un obiettivo primario. L'architettura deve essere progettata per raggiungere un uptime del 99.8%. La gestione dei dati, tramite un database transazionale (ACID), deve garantire l'integrità assoluta di ogni informazione. Il design deve includere una strategia di backup e ripristino con un RTO massimo di 4 ore. Inoltre, l'architettura dell'applicazione mobile deve implementare meccanismi di caching e sincronizzazione per garantire il funzionamento offline.
- **Implementare Sicurezza a Ogni Livello (NFR11, NFR12, NFR13):** La sicurezza deve essere integrata nel design ("Security by Design"). L'architettura deve prevedere la memorizzazione sicura delle password tramite hashing. Tutta la comunicazione tra i componenti del sistema deve avvenire obbligatoriamente su canali cifrati (HTTPS/TLS). La logica applicativa, sia a livello di frontend che di backend, deve implementare una validazione rigorosa di tutti gli input utente per prevenire vulnerabilità.
- **Favorire la Manutenibilità e l'Evoluzione (NFR14):** L'obiettivo è creare un sistema che possa durare ed evolvere nel tempo. La scelta di un'architettura modulare a 3 livelli è fondamentale per raggiungere questo scopo. La scomposizione del backend in sottosistemi a responsabilità singola disaccoppia i componenti, permettendo di aggiornare o sostituire parti del sistema (es. il servizio di calcolo percorsi) senza impattare il resto dell'applicazione.

1.3. Definitions, acronyms, and abbreviations

Acronimi

ACID Atomicity, Consistency, Isolation, Durability. Proprietà che garantiscono l'integrità delle transazioni in un database.

API Application Programming Interface. Un insieme di definizioni e protocolli per la costruzione e l'integrazione di software applicativo.

DBMS Database Management System. Un sistema software progettato per la gestione di database.

ETA Estimated Time of Arrival. Orario di arrivo stimato.

HTTP/S HyperText Transfer Protocol / Secure. Il protocollo di comunicazione fondamentale per il World Wide Web, con un layer di sicurezza crittografica.

JPA Jakarta Persistence API. Specifica Java per la gestione della persistenza e dell'object-relational mapping.

JSON JavaScript Object Notation. Un formato standard basato su testo per la rappresentazione di dati strutturati.

JWT JSON Web Token. Uno standard aperto per la creazione di token di accesso che asseriscono un numero di "claims" tra due parti.

RAD Requirements Analysis Document. Il documento che definisce i requisiti funzionali e non funzionali del sistema.

RBAC Role-Based Access Control. Una politica di controllo degli accessi basata sui ruoli assegnati agli utenti.

REST Representational State Transfer. Uno stile architetturale per la progettazione di applicazioni in rete, comunemente usato per le API web.

RTO Recovery Time Objective. Il tempo massimo tollerabile entro cui un sistema deve essere ripristinato dopo un disastro.

SDD System Design Document. Il presente documento.

SDK Software Development Kit. Un insieme di strumenti di sviluppo software in un unico pacchetto installabile.

SQL Structured Query Language. Il linguaggio standard per l'interazione con i database relazionali.

UI User Interface. L'interfaccia utente.

UML Unified Modeling Language. Il linguaggio standard per la modellazione visuale di sistemi software.

Definizioni

3-Tier Architecture Uno stile architetturale che scompone l'applicazione in tre livelli logici e fisici: il *Presentation Tier*, il *Logic Tier* e il *Data Tier*.

Dart Il linguaggio di programmazione, ottimizzato per i client, utilizzato per sviluppare applicazioni con Flutter.

Endpoint Un URL specifico di un'API REST che espone una particolare risorsa o funzionalità (es. `/api/viaggi`).

Flutter Un UI toolkit open-source di Google per la creazione di applicazioni cross-platform compilate in modo nativo per mobile, web e desktop da un'unica codebase.

Hashing Un processo crittografico unidirezionale che trasforma una stringa (come una password) in una stringa di lunghezza fissa, rendendola irriconoscibile ma verificabile.

Stateless Un attributo di un componente (tipicamente il backend) che non memorizza alcuna informazione sullo stato delle sessioni client.

Subsystem Un componente o modulo logico del sistema con un insieme ben definito di responsabilità.

Token Un "gettone" digitale (JWT) rilasciato a un utente dopo un'autenticazione con successo e utilizzato per autorizzare le richieste successive.

Widget L'elemento costruttivo fondamentale di un'interfaccia utente in Flutter. Ogni elemento visibile è un widget.

1.4. References

Tutte le decisioni di progettazione descritte in questo documento sono state prese per soddisfare i requisiti definiti nel seguente documento:

1. *Requirements Analysis Document (RAD) - HeavyRoute, Versione 1.2*

1.5. Overview

Questo System Design Document descrive l'architettura software del sistema *HeavyRoute*. La Sezione 2 analizza brevemente l'approccio non sistematico attualmente in uso. La Sezione 3 costituisce il nucleo del documento, dettagliando l'architettura a 3 livelli proposta, la scomposizione in sottosistemi, la mappatura hardware/software e le strategie per la gestione dei dati, la sicurezza, il controllo e le condizioni al contorno. Infine, la Sezione 4 delinea in modo preliminare i servizi esposti da ciascun sottosistema, che serviranno come base per la successiva fase di Object Design.

2. Current software architecture

Attualmente, Logistica Mediterranea S.P.A. non adotta un sistema software integrato per la gestione dei trasporti eccezionali. Le operazioni sono supportate da quella che può essere definita un'architettura "ad-hoc" e frammentata, basata sull'interazione manuale tra persone e strumenti di produttività individuale non comunicanti tra loro.
L'architettura implicita corrente può essere scomposta nei seguenti "componenti":

- **Canali di Comunicazione Esterni (Interfaccia Cliente):** Le richieste di trasporto vengono ricevute tramite canali non strutturati, principalmente client di posta elettronica e telefonate. Questi canali fungono da "interfaccia di input", ma non prevedono alcuna validazione o standardizzazione dei dati.
- **Fogli di Calcolo (Logica di Business e Persistenza Dati):** La pianificazione dei viaggi, la gestione della disponibilità di autisti e veicoli, e il tracciamento dello stato delle spedizioni sono mantenuti su fogli di calcolo (es. Microsoft Excel). Questi file fungono contemporaneamente da logica applicativa (tramite l'inserimento manuale di formule e dati) e da "database" non relazionale e non concorrente.
- **Comunicazioni Interne (Flusso di Controllo):** Il flusso di lavoro e il passaggio di consegne tra i diversi ruoli (Committente, Pianificatore Logistico, Traffic Coordinator, Autista) sono orchestrati tramite comunicazioni sincrone (telefonate) e asincrone (e-mail, messaggi). Questo approccio funge da "bus di messaggi" manuale.
- **Archivi Locali e Cartacei (Gestione Documentale):** La documentazione legale e operativa (permessi, autorizzazioni, prove di consegna) è gestita in formato cartaceo o come file digitali sparsi su archivi locali o scambiati via e-mail.

Questa architettura manuale introduce una serie di problematiche significative che il nuovo sistema si propone di risolvere:

- **Mancanza di Integrità e Coerenza dei Dati:** L'assenza di un database centrale e di validazioni porta a un alto rischio di errori umani, duplicazione di dati e informazioni obsolete.
- **Assenza di Tracciabilità in Tempo Reale:** È impossibile avere una visione d'insieme e aggiornata dello stato di tutti i trasporti, delle risorse e delle richieste.
- **Inefficienza Operativa:** I tempi di risposta sono elevati a causa della necessità di reperire, trascrivere e sincronizzare manualmente le informazioni tra persone e strumenti diversi.
- **Scarsa Sicurezza:** I dati non sono protetti da meccanismi di controllo degli accessi granulari e la loro gestione non è tracciata.

3. Proposed software architecture

3.1. Overview

L'architettura software del sistema *HeavyRoute* è basata su un modello a *3 livelli (3-Tier Architecture)*, scelto per separare nettamente le responsabilità tra interfaccia, logica di business e gestione dei dati. Per massimizzare il riutilizzo del codice e garantire un'esperienza utente coerente su tutte le piattaforme, è stata adottata una strategia basata su un'unica tecnologia per il frontend.
I tre livelli sono così definiti:

1. **Presentation Tier (Livello di Presentazione):** Questo livello è responsabile dell'intera interfaccia utente (UI) e sarà sviluppato interamente con il framework **Flutter**. Grazie alle sue capacità cross-platform, Flutter sarà utilizzato per creare:

   - Una **Applicazione Web** per il personale di back-office (PL, TC, Gestori Account) e per i Committenti.
   - Una **Applicazione Mobile** per Android e iOS, dedicata esclusivamente all'Autista.

   L'adozione di un'unica codebase in **Dart/Flutter** permetterà di condividere la logica di presentazione, i modelli di dati e i servizi di comunicazione con il backend, accelerando lo sviluppo e semplificando la manutenzione, pur creando interfacce ottimizzate per ogni specifico caso d'uso (dashboard complesse per il web, interfaccia semplice e sicura per il mobile).

2. **Logic Tier (Livello Logico/Applicativo):** Il cuore del sistema, implementato con il framework **Spring Boot**, conterrà tutta la logica di business. Esporrà un'unica **API RESTful** che fungerà da singolo punto di contatto per entrambe le applicazioni Flutter (web e mobile), garantendo coerenza, sicurezza e un flusso di dati centralizzato.

3. **Data Tier (Livello Dati):** Responsabile della persistenza dei dati, questo livello sarà implementato con un DBMS relazionale **MariaDB**. L'accesso al database da parte del Logic Tier sarà gestito in modo astratto attraverso un layer di persistenza basato su Jakarta Persistence API (JPA).

La comunicazione tra il Presentation Tier e il Logic Tier avverrà esclusivamente tramite protocollo **HTTPS** e sarà protetta da un meccanismo di autenticazione basato su token (JWT), garantendo la sicurezza di tutte le interazioni, indipendentemente dalla piattaforma.

3.2. Subsystem decomposition

Il Logic Tier, implementato come un'applicazione monolitica ma modulare con Spring Boot, è stato scomposto in sottosistemi (o package) logici per applicare il principio della separazione delle responsabilità (Separation of Concerns). Ogni sottosistema incapsula una specifica area di competenza del dominio di business, esponendo i propri servizi agli altri sottosistemi o al Presentation Tier tramite l'API REST.
Questa scomposizione, illustrata nel Diagramma dei Componenti in Figura 1, migliora la manutenibilità, facilita i test e permette una più chiara divisione del lavoro di sviluppo.

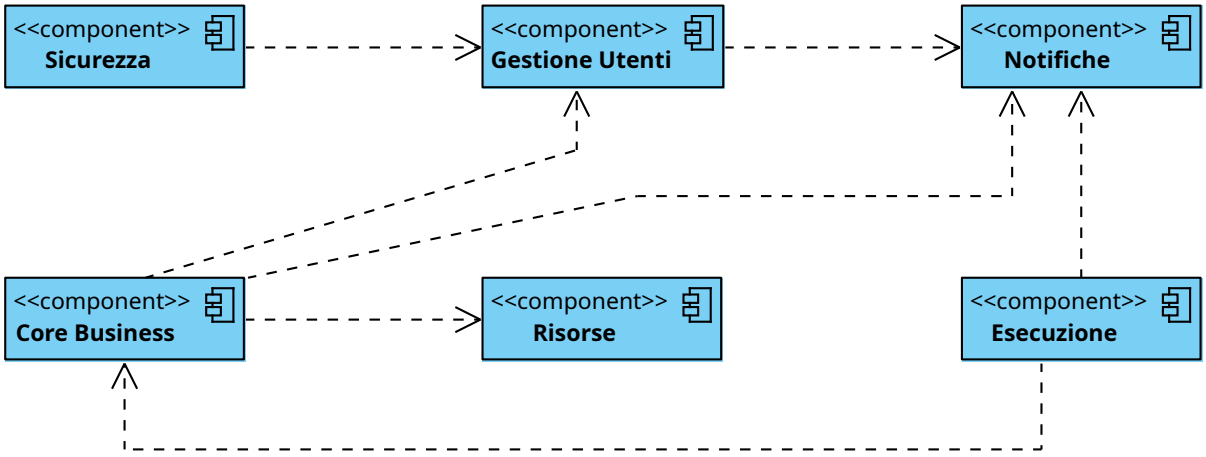

Figura 1: Diagramma dei Componenti del Backend.

I sottosistemi identificati sono i seguenti:

- **Sottosistema di Sicurezza e Autenticazione:**
  - **Responsabilità:** Gestisce tutti gli aspetti legati alla sicurezza. È responsabile dell'autenticazione degli utenti (UC1), della generazione e validazione dei token JWT, e dell'applicazione delle regole di autorizzazione (RBAC) su ogni endpoint dell'API.
  - **Dipendenze Principali:** Dipende dal sottosistema di 'Gestione Utenti' per recuperare i dati degli utenti e i loro ruoli.

- **Sottosistema di Gestione Utenti:**
  - **Responsabilità:** Gestisce il ciclo di vita di tutte le entità utente. Implementa la logica per la registrazione dei nuovi Committenti (UC2), la creazione e gestione degli utenti interni (UC14, UC15, UC16, UC17).
  - **Dipendenze Principali:** Dipende dal sottosistema 'Notifiche'.

- **Sottosistema di Gestione Richieste e Viaggi (Core Business):**
  - **Responsabilità:** Costituisce il cuore del sistema. Gestisce l'intero ciclo di vita di una richiesta e di un viaggio. Implementa la logica per la creazione di richieste e preventivi (UC3, UC12), la pianificazione (UC4), la validazione (UC5), e la gestione delle modifiche richieste dal cliente (UC13).
  - **Dipendenze Principali:** Dipende da 'Gestione Utenti' (per associare PL, TC, Autisti), da 'Gestione Risorse' (per trovare veicoli disponibili) e da 'Notifiche'.

- **Sottosistema di Esecuzione e Monitoraggio:**
  - **Responsabilità:** Gestisce tutte le interazioni in tempo reale con l'applicazione mobile dell'Autista. Implementa la logica per l'esecuzione del trasporto (UC6), la gestione degli imprevisti segnalati dall'autista (UC7) e la conferma delle modifiche in viaggio (UC11).
  - **Dipendenze Principali:** Dipende da 'Gestione Richieste e Viaggi' per aggiornare lo stato del viaggio.

- **Sottosistema di Gestione Risorse e Viabilità:**
  - **Responsabilità:** Si occupa della gestione dei dati anagrafici delle risorse e delle condizioni esterne. Gestisce la flotta di veicoli, le anagrafiche dei partner esterni (scorte, trasportatori) e i vincoli di viabilità inseriti dal TC (UC8).
  - **Dipendenze Principali:** È un modulo di supporto, utilizzato principalmente dal sottosistema di 'Gestione Richieste e Viaggi'.

- **Sottosistema di Notifiche:**
  - **Responsabilità:** È un servizio trasversale responsabile dell'invio di tutte le comunicazioni asincrone, come e-mail (es. credenziali, conferme) e potenzialmente notifiche push per l'app mobile.
  - **Dipendenze Principali:** Viene chiamato da quasi tutti gli altri sottosistemi quando si verifica un evento che richiede una notifica.

3.3. Hardware/software mapping

Questa sezione descrive come i componenti software dell'architettura proposta vengono mappati sull'infrastruttura hardware (fisica o virtualizzata) e come interagiscono tra loro attraverso la rete. Il Diagramma di Deploy in Figura 2 illustra questa distribuzione.

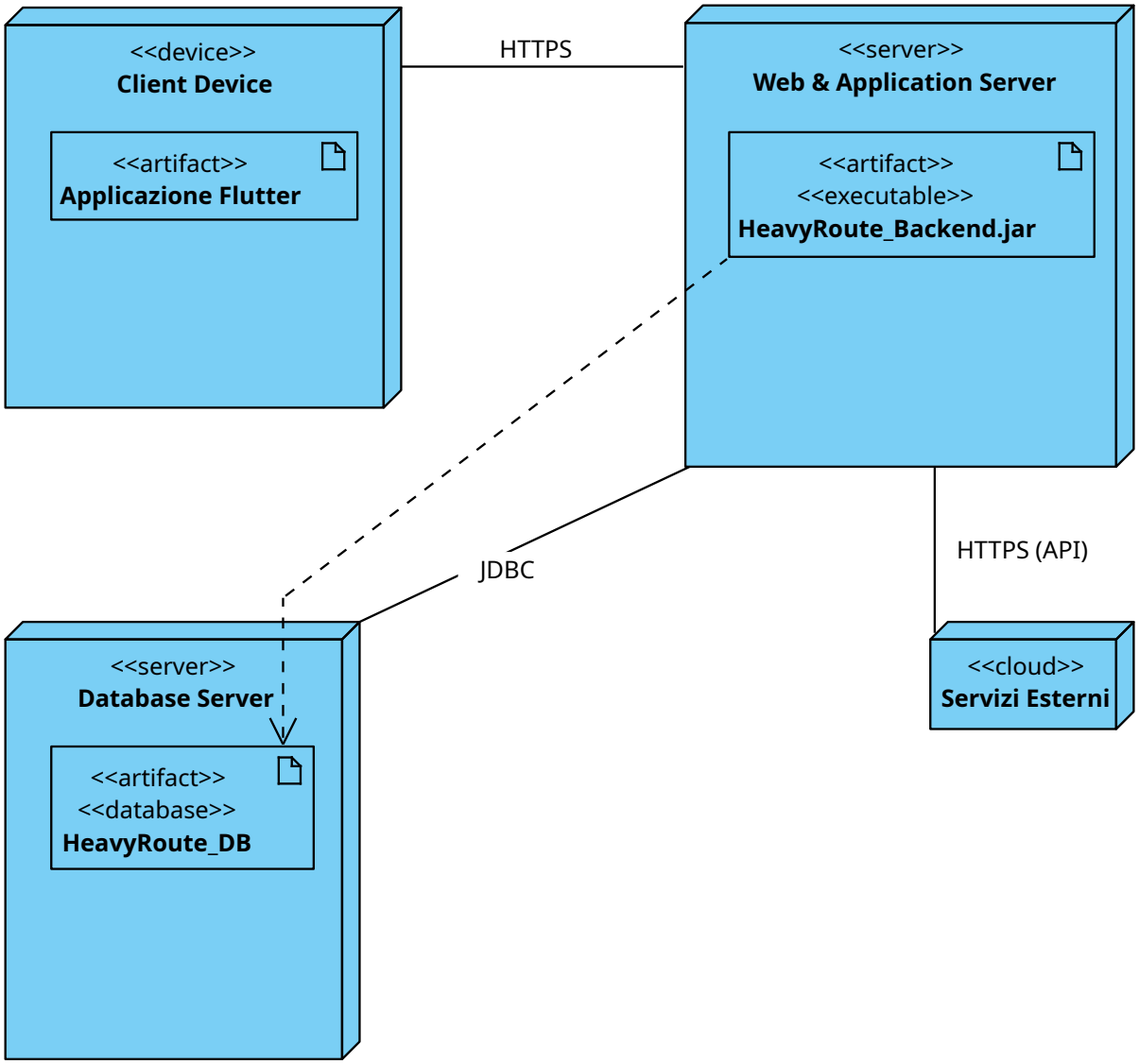

Figura 2: Diagramma di Deploy del sistema HeavyRoute.

L'infrastruttura del sistema è scomposta nei seguenti nodi computazionali:

- **Client Device:** Rappresenta il dispositivo dell'utente finale. Questo nodo non è omogeneo, ma si differenzia in base all'attore:
  - **Per Utenti Web (PL, TC, Committente, Gestore Account):** Il nodo è un computer desktop o laptop su cui è in esecuzione un browser web moderno.

  - **Per Utenti Mobile (Autista):** Il nodo è un dispositivo mobile (smartphone o tablet) con sistema operativo Android o iOS.

  Su questo nodo viene eseguita l'**applicazione client Flutter**, compilata come applicazione web per i primi e come applicazione nativa per i secondi.

- **Web & Application Server:** Questo nodo rappresenta l'ambiente di esecuzione del Logic Tier. Si tratta di un server (fisico o, più probabilmente, una macchina virtuale o un container in un ambiente cloud) con un sistema operativo server (es. Linux) e una Java Virtual Machine (JVM). Su questo nodo viene deployato l'artefatto software principale:
  - `HeavyRoute_Backend.jar`: L'applicazione **Spring Boot** eseguibile, che contiene tutta la logica di business e l'API RESTful.
- **Database Server:** Questo nodo è un server dedicato esclusivamente alla gestione dei dati persistenti (Data Tier). Ospita l'istanza del Database Management System (DBMS) scelto. L'artefatto principale su questo nodo è:
  - `HeavyRoute_DB`: Lo schema del database **MariaDB**, che contiene tutte le tabelle, le relazioni e i dati del sistema.
- **Servizi Esterni in Cloud:** Questo nodo rappresenta i servizi di terze parti a cui il nostro sistema si affida. Questi servizi sono ospitati sull'infrastruttura dei rispettivi provider e vengono invocati tramite API pubbliche. Per il sistema HeavyRoute, includono:
  - Servizi di mappatura (es. Google Maps API) per il calcolo dei percorsi.
  - Servizi di invio e-mail.
  - Sistema VIES per la validazione delle partite IVA.

Flussi di Comunicazione: Le interazioni tra i nodi avvengono tramite protocolli di rete standard:

- La comunicazione tra il **Client Device** e il **Web & Application Server** avviene tramite il protocollo **HTTPS**, garantendo la sicurezza delle chiamate all'API REST.
- La comunicazione tra il **Web & Application Server** e il **Database Server** avviene tramite il protocollo **JDBC** (Java Database Connectivity).
- La comunicazione tra il **Web & Application Server** e i **Servizi Esterni** avviene tramite chiamate API su protocollo **HTTPS**.

3.4. Persistent data management

Questa sezione descrive la strategia per la gestione dei dati persistenti del sistema *HeavyRoute*. Il sistema necessita di memorizzare in modo affidabile tutte le entità di business, tra cui utenti, richieste di trasporto, viaggi, veicoli e la relativa documentazione, per garantirne la disponibilità e la coerenza nel tempo.

Modello dei Dati e Schema Il modello logico dei dati del sistema è una diretta rappresentazione del `Diagramma delle Classi del Dominio` definito nel Requirements Analysis Document (RAD). L'approccio adottato è quello dell'Object-Relational Mapping (ORM), dove:

- Ogni classe persistente del modello (es. `Viaggio`, `Utente`, `Veicolo`) verrà mappata su una tabella corrispondente nello schema del database.
- Gli attributi di ciascuna classe diventeranno le colonne delle rispettive tabelle, con i tipi di dato appropriati (es. `VARCHAR`, `INT`, `DATETIME`).
- Le relazioni tra le classi (Associazioni, Composizioni) verranno implementate tramite l'uso di chiavi esterne (foreign keys) per mantenere l'integrità referenziale.

Database Management System (DBMS) Il DBMS scelto per il sistema è **MariaDB**. Questa scelta è motivata dalle seguenti ragioni:

- **Modello Relazionale:** MariaDB è un DBMS relazionale, ideale per gestire i dati strutturati e le complesse relazioni definite nel nostro Object Model.
- **Affidabilità e Performance:** È una tecnologia matura, open-source, nota per la sua stabilità e le sue ottime performance, adatta a carichi di lavoro transazionali come quelli del nostro sistema.
- **Compatibilità:** Essendo un fork di MySQL, è pienamente compatibile con gli standard SQL e con i connettori standard del mondo Java, come il driver JDBC, garantendo una perfetta integrazione con il nostro backend Spring Boot.
- **Costo e Supporto:** Non prevede costi di licenza e gode del supporto di una vasta community, riducendo il Total Cost of Ownership (TCO).

Strategia di Accesso e Incapsulamento dei Dati L'accesso al database sarà completamente astratto e incapsulato all'interno di un **Data Access Layer (DAL)** nel backend Spring Boot. Questa strategia sarà implementata utilizzando:

- **Jakarta Persistence API (JPA)** con **Hibernate** come implementazione. JPA standardizza il processo di ORM, rendendo il codice della logica di business indipendente dal database sottostante.
- Il **Repository Pattern**, fornito nativamente da **Spring Data JPA**. Per ogni entità (es. `Viaggio`), verrà creato un repository (es. `ViaggioRepository`) che esporrà metodi per le operazioni CRUD (Create, Read, Update, Delete) e per le query più complesse.

Questo approccio garantisce un forte disaccoppiamento: i sottosistemi della logica di business (es. 'Pianificazione') non scriveranno mai query SQL, ma interagiranno con il database attraverso i metodi dei repository (es. `viaggioRepository.save(viaggio)`), rendendo il codice più pulito, testabile e manutenibile.

Integrità e Gestione delle Transazioni Per garantire l'integrità dei dati (proprietà ACID), ogni operazione di business che coinvolge più passaggi di scrittura sul database sarà gestita come una singola `transazione atomica` Il framework Spring Boot, tramite la sua gestione transazionale (es. con l'annotazione `@Transactional`), assicurerà che un'operazione complessa (es. la creazione di un `Viaggio` con il suo `Percorso` e l'aggiornamento della `RichiestaTrasporto`) o abbia successo completamente, o fallisca completamente (rollback), senza lasciare il database in uno stato inconsistente.

3.5. Access control and security

La sicurezza è un obiettivo di progettazione primario per il sistema *HeavyRoute*. L'architettura implementerà una strategia di sicurezza a più livelli ("defense in depth") per proteggere l'integrità, la riservatezza e la disponibilità dei dati. Le politiche di sicurezza coprono l'autenticazione degli utenti, l'autorizzazione basata sui ruoli e la protezione dei dati sia in transito che a riposo.

Autenticazione Il sistema deve verificare in modo sicuro l'identità di ogni utente prima di consentire l'accesso a qualsiasi risorsa protetta.

- **Meccanismo:** L'autenticazione sarà gestita tramite un flusso basato su token. Specificamente, verrà utilizzato lo standard **JSON Web Token (JWT)**.
- **Flusso:**
  1. L'utente invia le proprie credenziali (username e password) a un endpoint di login sicuro (`/api/auth/login`).
  2. Il backend Spring Boot verifica le credenziali confrontando l'hash della password fornita con quello memorizzato nel database.
  3. Se la verifica ha successo, il server genera un JWT firmato digitalmente. Questo token contiene le "claims", ovvero informazioni sull'utente come il suo ID, il suo ruolo e una data di scadenza.
  4. Il token viene restituito al client (l'applicazione Flutter).
  5. Per tutte le richieste successive a endpoint protetti, il client dovrà includere il JWT nell'header `Authorization` (usando lo schema *Bearer*).
- **Gestione delle Password:** Le password degli utenti non saranno mai memorizzate in chiaro. Saranno salvate nel database esclusivamente come hash crittografico, generato tramite l'algoritmo **Bcrypt**, fornito da Spring Security.

Autorizzazione (Access Control) Una volta che un utente è autenticato, il sistema deve garantire che possa accedere solo alle funzionalità e ai dati permessi dal suo ruolo.

- **Modello:** Verrà implementato un modello di `Role-Based Access Control (RBAC)`. Ad ogni utente è associato uno e un solo ruolo (es. *Committente*, *Pianificatore Logistico*, *Autista*, etc.), come definito nel Diagramma delle Classi degli Utenti.
- **Implementazione:** L'autorizzazione sarà gestita a livello di backend da **Spring Security**. Ogni endpoint dell'API REST sarà protetto da regole che specificano quali ruoli sono autorizzati a invocarlo. Ad esempio:

  - L'endpoint `POST /api/viaggi/pianifica` sarà accessibile solo agli utenti con il ruolo `ROLE_PIANIFICATORE_LOGISTICO`.
  - L'endpoint `GET /api/richieste/mie` sarà accessibile solo agli utenti con il ruolo `ROLE_COMMITTENTE`, e la logica di business assicurerà che ogni committente possa vedere solo le proprie richieste.
  - L'endpoint `POST /api/utenti/interni` sarà accessibile solo al `ROLE_GESTORE_ACCOUNT`.

Sicurezza dei Dati Oltre all'accesso, i dati stessi devono essere protetti.

- **Dati in Transito:** Tutta la comunicazione tra i client Flutter (web e mobile) e il backend Spring Boot avverrà obbligatoriamente tramite il protocollo **HTTPS**. Questo garantisce che tutto il traffico API, inclusi i token di autenticazione e i dati di business, sia crittografato e protetto da attacchi *man-in-the-middle*.
- **Validazione degli Input:** Per prevenire vulnerabilità come SQL Injection e Cross-Site Scripting (XSS), ogni dato proveniente dal client sarà rigorosamente validato a livello di backend prima di essere processato o salvato. Verranno utilizzate le annotazioni di validazione di Spring Boot (Bean Validation).

3.6. Global software control

Il controllo globale del software nel sistema *HeavyRoute* è progettato secondo un modello **guidato dagli eventi (Event-Driven)**, dove il flusso delle operazioni è quasi sempre avviato da un'azione esplicita di un attore. Il sistema non esegue processi autonomi complessi, ma reagisce alle richieste degli utenti.
Il meccanismo di controllo può essere scomposto in due modalità principali di comunicazione e sincronizzazione.

Controllo Sincrono (Request-Response) La stragrande maggioranza delle interazioni tra il Presentation Tier (l'applicazione Flutter) e il Logic Tier (il backend Spring Boot) segue un modello di comunicazione **sincrono Request-Response**.

- **Flusso:**
  1. Un utente esegue un'azione sull'interfaccia (es. clicca su "Salva Richiesta").
  2. Il client Flutter invia una richiesta HTTP (es. `POST /api/richieste`) al backend.
  3. Il client si mette in uno stato di attesa, bloccando (o mostrando un indicatore di caricamento) ulteriori interazioni su quella specifica funzionalità.
  4. Il backend Spring Boot riceve la richiesta, la processa in un unico thread (per quella richiesta), esegue le operazioni di business necessarie (validazione, interazione con il database) e, al termine, formula una risposta.
  5. La risposta HTTP (con i dati richiesti o un messaggio di successo/errore) viene inviata al client.
  6. Il client riceve la risposta, aggiorna l'interfaccia utente di conseguenza e torna a essere pienamente interattivo.
- **Gestione della Concorrenza:** A livello di backend, l'Application Server gestirà un pool di thread per processare più richieste concorrenti provenienti da utenti diversi. La concorrenza sull'accesso ai dati sarà gestita dal DBMS (MariaDB) tramite i suoi meccanismi di locking e dal layer di persistenza (JPA) tramite il controllo della concorrenza ottimistica (dove applicabile), per prevenire conflitti e garantire la coerenza dei dati.

Controllo Asincrono (Notifiche) Per le operazioni che non richiedono una risposta immediata o che devono informare attori non direttamente coinvolti nell'azione corrente, il sistema adotta un modello di comunicazione **asincrono**.

- **Flusso:**

1. Durante l'elaborazione di una richiesta sincrona, un sottosistema (es. 'Pianificazione') rileva che si è verificato un evento che richiede una notifica (es. un viaggio è stato validato).
2. Il sottosistema invoca il servizio del sottosistema di 'Notifiche', passandogli i dettagli dell'evento. Questa chiamata è "fire-and-forget": il chiamante non attende il completamento dell'invio.
3. Il sottosistema di 'Notifiche' prende in carico la richiesta e la gestisce in background (potenzialmente tramite una coda di messaggi interna), inviando l'e-mail o la notifica push all'utente destinatario.

- **Casi d'uso tipici:** Questo modello viene utilizzato per tutte le notifiche tra i ruoli, come notificare al PL una nuova richiesta, al TC una richiesta di validazione, o all'Autista un nuovo incarico. Questo garantisce che l'esperienza utente dell'attore che ha avviato l'azione non venga rallentata dai tempi di invio delle notifiche.

3.7. Boundary conditions

Questa sezione descrive il comportamento atteso del sistema *HeavyRoute* durante le sue fasi di avvio, arresto e in risposta a condizioni di errore eccezionali.

Start-up (Avvio del Sistema) La procedura di avvio riguarda principalmente il Logic Tier (l'applicazione Spring Boot).

1. All'avvio dell'applicazione (es. tramite l'esecuzione del file `.jar`), il container Spring inizializza tutti i suoi componenti (beans).
2. Viene stabilita la connessione al database MariaDB. Il sistema di gestione del pool di connessioni viene configurato e popolato.
3. Se configurato per farlo, un meccanismo di migrazione del database verifica la coerenza dello schema del database e applica eventuali aggiornamenti necessari.
4. Vengono inizializzati i sottosistemi principali, come il framework di sicurezza (Spring Security) e il server web embedded.
5. Una volta che tutti i componenti sono attivi e pronti, l'applicazione si mette in ascolto sulla porta configurata per le richieste HTTP in arrivo.
6. L'applicazione client Flutter (web e mobile) può iniziare a comunicare con il backend non appena l'endpoint di "health check" del server risponde positivamente.

Shutdown (Arresto del Sistema) L'arresto del sistema deve avvenire in modo controllato ("graceful shutdown") per prevenire la perdita di dati o stati inconsistenti.

1. Quando l'applicazione riceve un segnale di arresto (es. da un amministratore di sistema o da un orchestratore di container), il server web smette di accettare nuove richieste HTTP in ingresso.
2. L'applicazione attende il completamento delle richieste attualmente in elaborazione, entro un timeout predefinito.
3. Vengono rilasciate le risorse principali: il pool di connessioni al database viene chiuso in modo pulito, terminando tutte le connessioni attive.
4. I componenti e i thread in background (es. quelli per le notifiche asincrone) vengono terminati in modo controllato.
5. L'applicazione termina il suo processo.

Error Handling (Gestione degli Errori Eccezionali) Questa sezione descrive la gestione di errori imprevisti a livello di sistema, non degli errori di validazione gestiti dai casi d'uso.

- **Perdita di Connessione al Database:** Se il Logic Tier perde la connessione al Database Server, ogni nuova richiesta che richiede un accesso ai dati fallirà. Il sistema risponderà al client con un codice di stato HTTP `503 Service Unavailable`. L'applicazione tenterà periodicamente di ristabilire la connessione al database.
- **Errore Interno Inatteso:** In caso di un'eccezione non gestita all'interno della logica di business (es. `NullPointerException`), un gestore di eccezioni globale ("Global Exception Handler") intercetterà l'errore. Per evitare di esporre dettagli sensibili dell'implementazione, il sistema non restituirà lo stack trace al client, ma risponderà con un codice di stato HTTP `500 Internal Server Error` e un messaggio generico (es. "Si è verificato un errore interno. Riprovare più tardi."). L'errore dettagliato, completo di stack trace, verrà invece registrato su file di log sul server per successive analisi da parte degli sviluppatori.
- **Servizio Esterno non Raggiungibile:** Se una chiamata a un servizio esterno (es. VIES, Google Maps API) fallisce per problemi di rete o perché il servizio è offline, il sottosistema corrispondente gestirà l'eccezione in modo controllato. A seconda della criticità, l'operazione potrebbe essere interrotta (informando l'utente con un messaggio specifico, es. "Impossibile verificare la Partita IVA in questo momento") o ritentata in background.

4. Subsystem services

Questa sezione descrive i servizi pubblici offerti da ciascun sottosistema del backend. Seguendo il design pattern **Facade**, ogni sottosistema espone una singola interfaccia di servizio che funge da unico punto di ingresso per i client (i controller dell'API o altri sottosistemi). Questo approccio riduce l'accoppiamento tra i componenti e nasconde la complessità interna di ciascun modulo. Le operazioni qui elencate rappresentano i metodi pubblici principali di queste classi Facade.

Sottosistema: `Sicurezza e Autenticazione`

- **Class:** `AuthenticationService`
- **Responsabilità Principale:** Gestire l'autenticazione, l'autorizzazione e la sicurezza degli accessi. Implementa **UC1** e **UC18**.
- **Servizi Offerti:**
  - `login(username, password)`: Verifica le credenziali e, in caso di successo, genera un token JWT.
  - `validateToken(token)`: Verifica la firma e la scadenza di un token per autorizzare le richieste API.
  - `initiatePasswordReset(email)`: Avvia il processo di recupero password.
  - `completePasswordReset(token, nuovaPassword)`: Finalizza il reset della password.

Sottosistema: `Gestione Utenti`

- **Class:** `UserService`
- **Responsabilità Principale:** Gestire il ciclo di vita e i dati anagrafici di tutte le entità Utente. Implementa **UC2, UC14, UC15, UC16, UC17**.
- **Servizi Offerti:**
  - `registraNuovoCommittente(datiRegistrazione)`: Crea una richiesta di registrazione per un nuovo cliente.
  - `approvaCommittente(idRichiesta)`: Approva una richiesta e crea l'account `Committente`.
  - `creaUtenteInterno(datiUtente)`: Crea un nuovo account per un membro del personale.

  - `modificaUtenteInterno(idUtente, datiDaModificare)`: Aggiorna i dati di un utente interno.
  - `disattivaUtenteInterno(idUtente)`: Disabilita l'accesso per un utente interno.
  - `riattivaUtenteInterno(idUtente)`: Riabilita un account disattivato.
  - `findUtenteByUsername(username)`: Servizio offerto al sottosistema di Sicurezza.
  - `findUtenteById(idUtente)`: Servizio generico per recuperare i dati di un utente.

Sottosistema: `Gestione Richieste e Viaggi (Core Business)`

- **Class:** `ViaggioService`
- **Responsabilità Principale:** Orchestrare l'intero ciclo di vita di una richiesta e di un viaggio, dalla creazione alla pianificazione e validazione. Implementa **UC3, UC4, UC5, UC12, UC13**.
- **Servizi Offerti:**
  - `creaRichiestaTrasporto(dati, idCommittente)`: Crea una nuova richiesta o una bozza.
  - `approvaRichiesta(idRichiesta)`: Approva una richiesta e genera un `Viaggio` in stato "In Pianificazione".
  - `assegnaRisorseAViaggio(idViaggio, idAutista, idVeicolo)`: Associa le risorse a un viaggio.
  - `calcolaEAssociaPercorso(idViaggio)`: Genera e associa un percorso a un viaggio.
  - `sottomettiPerValidazione(idViaggio)`: Inoltra un viaggio al TC.
  - `validaViaggio(idViaggio, documenti)`: Il TC approva un piano di viaggio.
  - `confermaEAttivaViaggio(idViaggio)`: Il PL rende un viaggio operativo.
  - `creaRichiestaPreventivo(dati, idCommittente)`: Gestisce la creazione di una richiesta di preventivo.
  - `accettaPreventivo(idPreventivo)`: Converte un preventivo in una richiesta approvata.
  - `richiediModificaViaggio(idViaggio, modifiche)`: Gestisce una richiesta di modifica da parte del cliente.
  - `getViaggioDettagliato(idViaggio)`: Recupera tutti i dati di un viaggio.

Sottosistema: `Esecuzione e Monitoraggio`

- **Class:** `EsecuzioneViaggioService`
- **Responsabilità Principale:** Gestire gli eventi in tempo reale provenienti dall'app dell'autista. Implementa **UC6, UC7, UC11**.
- **Servizi Offerti:**
  - `accettaIncarico(idViaggio, idAutista)`: L'autista accetta formalmente un viaggio.
  - `aggiornaStatoOperativo(idViaggio, nuovoStato, timestamp)`: L'autista aggiorna lo stato del viaggio (es. "In Transito").
  - `segnalaImprevistoGrave(idViaggio, dettagli)`: Attiva il flusso di gestione di un'emergenza (UC7).
  - `segnalaRitardoSemplice(idViaggio, nuovoETA)`: Comunica un ritardo non bloccante.
  - `confermaRicezioneModifica(idViaggio, idAutista)`: L'autista conferma di aver ricevuto una modifica in viaggio (UC11).

Sottosistema: `Gestione Risorse e Viabilità`

- **Class:** `RisorseService`
- **Responsabilità Principale:** Gestire le anagrafiche delle risorse fisiche (flotta) e dei dati esterni (viabilità). Implementa parte di **UC4** e **UC8**.
- **Servizi Offerti:**
  - `getRisorseDisponibili(criteri)`: Restituisce la lista di autisti e veicoli liberi e idonei.
  - `aggiungiNuovoVeicolo(datiVeicolo)`: Aggiunge un veicolo alla flotta.
  - `registraEventoViabilita(datiEvento)`: Inserisce un nuovo vincolo di viabilità.
  - `getEventiViabilitaAttivi()`: Restituisce la lista degli eventi di viabilità correnti.

Sottosistema: `Notifiche`

- **Class:** `NotificationService`
- **Responsabilità Principale:** Fornire un servizio trasversale per l'invio di comunicazioni asincrone. È utilizzato da quasi tutti gli altri sottosistemi.

- **Servizi Offerti:**
  - `inviaEmail(destinatario, oggetto, corpo)`: Invia una comunicazione via e-mail.
  - `inviaNotificaPush(idUtente, messaggio)`: Invia una notifica all'app mobile dell'autista.

Glossary

Questa sezione fornisce la definizione dei principali termini tecnici, architetturali e tecnologici utilizzati in questo documento per garantire una comprensione univoca delle scelte di progettazione.

3-Tier Architecture (Architettura a 3 Livelli) Stile architetturale che scompone l'applicazione in tre livelli logici e fisici separati: il *Presentation Tier* (interfaccia utente), il *Logic Tier* (logica di business) e il *Data Tier* (database).

API RESTful (Representational State Transfer) Stile architetturale per la progettazione di interfacce di programmazione (API) basate sul protocollo HTTP. È il metodo con cui il frontend (Flutter) comunicherà con il backend (Spring Boot).

Artifact In un Diagramma di Deploy UML, rappresenta un'unità di software concreta e distribuibile, come un file `.jar` o uno schema di database.

Asynchronous Communication (Comunicazione Asincrona) Modello di comunicazione in cui il mittente di un messaggio non attende una risposta immediata (modello "fire-and-forget"). Utilizzato nel sistema per l'invio di notifiche.

Authentication (Autenticazione) Il processo di verifica dell'identità di un utente, rispondendo alla domanda "Chi sei?". Nel nostro sistema, è gestito tramite username/password e JWT.

Authorization (Autorizzazione) Il processo di determinare se un utente autenticato ha i permessi per eseguire una certa azione o accedere a una certa risorsa. Risponde alla domanda "Cosa puoi fare?".

Bcrypt Un algoritmo di hashing delle password progettato per essere lento e computazionalmente intensivo, al fine di resistere ad attacchi di tipo brute-force. È la tecnologia scelta per la memorizzazione sicura delle password.

Cross-Platform La capacità di un'applicazione software di essere eseguita su più piattaforme (es. iOS, Android, Web) a partire da un'unica codebase. È il vantaggio principale offerto da Flutter.

Dart Il linguaggio di programmazione, sviluppato da Google, utilizzato per scrivere applicazioni con il framework Flutter.

Deployment Diagram Un diagramma UML che modella la topologia fisica del sistema, mostrando come gli artefatti software sono mappati sui nodi hardware.

Endpoint Un URL specifico esposto da un'API REST che corrisponde a una singola funzionalità o risorsa (es. `/api/viaggi/id`).

Facade Pattern Un design pattern strutturale che fornisce un'interfaccia unificata e semplificata a un insieme di interfacce più complesse in un sottosistema. Nel nostro design, ogni sottosistema espone una classe di servizio che agisce da Facade.

Flutter Il UI toolkit open-source di Google scelto per lo sviluppo del Presentation Tier, sia per l'applicazione web che per quella mobile.

JPA (Jakarta Persistence API) Una specifica standard Java che descrive come gestire i dati relazionali nelle applicazioni Java. Semplifica l'interazione con il database astraendo le query SQL tramite l'Object-Relational Mapping (ORM).

JWT (JSON Web Token) Lo standard scelto per l'autenticazione basata su token. È un "gettone" digitale firmato che contiene informazioni sull'utente e viene scambiato tra client e server per autorizzare le richieste.

MariaDB Il Database Management System (DBMS) relazionale e open-source scelto per il Data Tier del sistema.

Node In un Diagramma di Deploy UML, rappresenta una risorsa computazionale (hardware o virtuale) su cui vengono eseguiti gli artefatti, come un server, un container o il dispositivo di un utente.

RBAC (Role-Based Access Control) Il modello di autorizzazione scelto per il sistema, in cui i permessi di accesso alle risorse sono associati a ruoli anziché a singoli utenti.

Repository Pattern Un design pattern che media tra il dominio e i layer di mappatura dei dati, agendo come una collezione in-memory di oggetti del dominio. È implementato da Spring Data JPA per astrarre l'accesso al database.

Spring Boot Il framework scelto per lo sviluppo del Logic Tier (backend). Semplifica la creazione di applicazioni basate su Spring, stand-alone e di livello production-grade.

Stateless Un principio di progettazione (in particolare per le architetture REST) in cui il server non memorizza alcuno stato relativo alla sessione del client. Ogni richiesta dal client deve contenere tutte le informazioni necessarie.

Subsystem (Sottosistema) Un modulo logico del sistema con un insieme ben definito di responsabilità e un'interfaccia pubblica (Facade).

Synchronous Communication (Comunicazione Sincrona) Il modello di comunicazione standard Request-Response, in cui il client invia una richiesta e si blocca in attesa di una risposta dal server.

Widget L'elemento costruttivo fondamentale di un'interfaccia utente in Flutter.