Università degli Studi di Salerno
Corso di Ingegneria del Software

AutoFlow
Object Design Document
Versione 1.2

Data: 08/11/2025

Sommario

1. Introduzione

Questo documento di Object Design descrive l'architettura logica e fisica del sistema AutoFlow, la piattaforma gestionale pensata per digitalizzare le attività delle piccole e medie concessionarie automobilistiche. Vengono presentate la struttura delle principali classi, interfacce e componenti software che costituiscono l'applicazione, con l'obiettivo di fornire una base chiara e coerente per le fasi di implementazione. Il sistema è progettato secondo un approccio orientato agli oggetti, con una netta separazione delle responsabilità tra moduli di autenticazione, gestione utenti, showroom & veicoli, proposte di vendita, fatturazione e statistiche, favorendo riusabilità, manutenibilità e scalabilità nel tempo. Il documento illustra le principali decisioni architetturali e i relativi trade–off, mettendo in relazione le scelte tecnologiche con i requisiti funzionali e non funzionali (sicurezza, prestazioni, tracciabilità dei dati). Sono inoltre definiti vincoli e linee guida implementative per garantire coerenza tra i vari sottosistemi, allineamento con gli standard di settore e conformità alle normative vigenti (es. protezione dei dati personali).

1.1. Object design trade-offs

La progettazione degli oggetti all'interno della piattaforma **AutoFlow** richiede una valutazione accurata dei compromessi tra scalabilità, affidabilità, semplicità di manutenzione, prestazioni e sostenibilità economica. Le decisioni architetturali adottate sono state guidate dai requisiti funzionali e non funzionali, con l'obiettivo di ottenere un equilibrio che permetta al sistema di crescere nel tempo mantenendo comunque costi e complessità sotto controllo. Questo paragrafo illustra i principali trade-off che hanno influenzato la definizione delle entità, dei controller e dei componenti del sistema, evidenziando il modo in cui tali scelte impattano sulla struttura software e sulla futura evoluzione di AutoFlow.

Affidabilità vs Cost-Effectiveness

Per garantire stabilità e continuità operativa, soprattutto nelle fasi critiche come la gestione delle proposte, l'aggiornamento dello stato dei veicoli o la generazione delle fatture PDF, sarebbe ideale adottare infrastrutture cloud scalabili e sistemi avanzati di monitoraggio. Tuttavia, per mantenere il sistema economicamente sostenibile per le piccole e medie concessionarie, AutoFlow adotta inizialmente un'infrastruttura più semplice (server locale o hosting base), che offre buoni livelli di affidabilità senza costi elevati.

Questo compromesso comporta una riduzione della resilienza sotto carico intenso, ma permette un rilascio rapido della piattaforma e lascia aperta la possibilità di scalare verso soluzioni più robuste in futuro, in linea con i requisiti di crescita della concessionaria. Il risultato è un equilibrio efficace tra affidabilità complessiva del sistema e sostenibilità economica.

Scalabilità vs Rapid Development

Una piattaforma gestionale destinata a crescere nel tempo richiederebbe idealmente un'autenticazione completamente stateless basata su JWT, facilitando la distribuzione su più nodi e il bilanciamento del carico. Tuttavia, l'adozione immediata di tale modello aumenterebbe notevolmente la complessità dello sviluppo.

Per questo motivo, AutoFlow utilizza nelle prime versioni un'autenticazione basata su **sessione e cookie**, più semplice da implementare e adeguata alle esigenze iniziali del sistema. Questo approccio consente un *time-to-market* più rapido, senza compromettere la possibilità di migrare in

futuro verso un'autenticazione stateless più scalabile, come già previsto nell'architettura del progetto.

Affidabilità vs Prestazioni

AutoFlow deve garantire affidabilità nelle operazioni di aggiornamento dei veicoli, gestione delle configurazioni e generazione della documentazione. Elementi come la validazione dei dati, la tracciabilità delle operazioni, la struttura delle proposte e la gestione delle fatture introducono inevitabilmente overhead, soprattutto nell'accesso al database o nella generazione dei PDF.

Per mantenere buone prestazioni senza sacrificare l'affidabilità:

- vengono sfruttati i vincoli ACID del database,
- vengono applicati controlli di coerenza lato server,
- si utilizza caching applicativo nelle parti più frequentemente consultate (showroom, lista veicoli),
- si ottimizzano le query sulle entità principali (Veicolo, Proposta, Configurazione).

In questo modo AutoFlow garantisce reattività anche in presenza di molte operazioni concorrenti, senza compromettere la consistenza dei dati.

Riutilizzabilità vs Semplicità

La struttura modulare di AutoFlow — basata su entità riutilizzabili come **Veicolo**, **Proposta**, **Configurazione**, **Cliente** — consente una gestione omogenea dei dati e riduce la duplicazione del codice. Tuttavia, progettare componenti riutilizzabili implica maggiore complessità, poiché:

- i controller devono essere sufficientemente generici,
- le view devono gestire casi d'uso differenti (online, in sede, staff),
- gli oggetti devono supportare diversi stati e transizioni (es. proposta: creata → accettata → finalizzata → fatturata).

Questo livello di astrazione aumenta l'efficienza in fase di manutenzione ma richiede una progettazione accurata per evitare eccessiva complessità nelle prime release. L'architettura adottata mantiene un buon compromesso: i componenti sono riutilizzabili, ma senza sovraccaricare il design con funzionalità non necessarie alla prima versione.

1.2. Linee guida documentazione di interfaccia

Per garantire coerenza, manutenibilità e qualità del codice nell'applicazione AutoFlow, sviluppata con Spring Boot per il back-end e React con TypeScript per il front-end, vengono adottate le seguenti linee guida.

Queste convenzioni traggono ispirazione da standard consolidati e best practices, assicurando uno stile uniforme e favorendo l'integrazione tra i diversi componenti del sistema.

Convenzioni di Nomenclatura

Camel Case

Java / Spring Boot

- Utilizzare camelCase per variabili, metodi e proprietà.
- Utilizzare PascalCase per classi, interfacce, DTO e controller.
- Evitare abbreviazioni non standard.

TypeScript / React

- Usare camelCase per variabili, funzioni e hook.
- Applicare PascalCase ai componenti React e alle interfacce.
- Nomi chiari, significativi e coerenti con il dominio ("Vehicle", "Proposal", "Invoice").

Consistenza tra front-end e back-end

- Mantenere allineati i nomi delle entità (es. Vehicle, Proposal, Configuration, Invoice).
- Allineare i nomi dei campi nelle API REST con le interfacce del front-end per ridurre errori di mapping.

Naming Convention per DTO, Controller e Service

Controller

Java

Ogni controller deve terminare con Controller, esprimendo chiaramente il proprio ruolo.

Esempi: *VehicleController, ProposalController, InvoiceController, AuthenticationController*

React

Anche se React non utilizza controller, i componenti che gestiscono logica complessa dovrebbero terminare con Container.

Esempi: *VehicleListContainer, ProposalDetailContainer, LoginContainer*

Service

Java

Ogni classe di servizio deve terminare con Service.

La logica di business deve essere collocata nei servizi, non nei controller.

Esempi: *VehicleService, ProposalService, InvoiceService, UserService*

TypeScript (front-end)

I moduli che gestiscono chiamate API o logiche cross-component terminano con Service.

Esempi: *VehicleService, AuthService, ProposalService*

Definizione delle Interfacce

Chiarezza e semplicità

- Le interfacce devono definire contratti precisi tra front-end e back-end.
- Evitare over-engineering o responsabilità multiple.

Separazione delle responsabilità

- Ogni interfaccia o DTO deve modellare un'unica entità funzionale (es. *VehicleDTO, ProposalDTO*).
- Evitare DTO "onnicomprensivi".

Stile delle Parentesi

Java

- Aprire { sulla stessa linea della dichiarazione.
- Chiudere } sempre su una nuova linea.

TypeScript / React

- Adottare lo stesso stile del back-end per coerenza.

Spaziatura

- Inserire uno spazio prima di (nei metodi.
- Non inserire spazi superflui all'interno delle parentesi.

Esempio

- Corretto: public VehicleDTO getVehicleById(Long id) {
- Errato: public VehicleDTO getVehicleById(Long id) {

Struttura del Progetto

Organizzazione Modulare:

Java

- Strutturare il codice in pacchetti coerenti, separando le diverse responsabilità (es. controller, service, repository, dto, model).

React

- Organizzare i componenti in cartelle basate sulle funzionalità o sulle pagine, distinguendo tra componenti presentazionali e container.

Coerenza e Standardizzazione

Aderenza agli Style Guide

- Java: Seguire le linee guida del Google Java Style Guide per mantenere uno stile di codice coerente.
- TypeScript: Adottare le pratiche suggerite nel TypeScript Style Guide per garantire una codifica uniforme.
- React: Implementare le raccomandazioni del React Style Guide per sviluppare componenti React chiari e mantenibili.
- Spring Framework: Allinearsi alle Code Style del Spring Framework per assicurare coerenza nelle implementazioni.

Documentazione

- Utilizzare Javadoc per i servizi, DTO e controller Spring.
- Usare JSDoc o tipizzazione esplicita nelle interfacce TS (es. /** Descrizione... */).

Utilizzo di Lombok (Riduzione del boilerplate)

Lombok deve essere utilizzato per:

- getter/setter (@Getter, @Setter)
- DTO e model (@Data)
- pattern Builder (@Builder)
- costruttori (@NoArgsConstructor, @AllArgsConstructor)

Questo migliora leggibilità e manutenzione del codice.

Gestione degli Errori e Validazioni

Back-end (Java / Spring Boot)

- Utilizzare @ControllerAdvice + @ExceptionHandler per centralizzare la gestione eccezioni.
- Impiegare le annotazioni di validazione:
- @NotNull
- @Size
- @Email
- @PositiveOrZero
- Validazioni lato controller con @Valid.

Front-end (React / TypeScript)

- Uso di librerie come Formik + Yup per validazioni robuste dei form.
- Gestione degli errori tramite:
    - useState
    - useEffect
    - Error boundaries (se necessario)

1.3. Design Pattern

Nel progetto Autoflow, sviluppato con Spring Boot, vengono adottati diversi design pattern per organizzare il codice in modo modulare, riutilizzabile e semplice da mantenere. Questi pattern offrono soluzioni consolidate a problematiche ricorrenti nella progettazione software e contribuiscono a gestire in modo più efficace la complessità dell'applicazione. Tra i principali, il Singleton assicura che una risorsa condivisa abbia una sola istanza globale, il Facade fornisce un punto di accesso semplificato a sistemi articolati, mentre l'Adapter permette l'integrazione tra componenti con interfacce non compatibili. Altri pattern come Bridge e Builder introducono maggiore flessibilità sia nella gestione delle implementazioni che nella costruzione di oggetti articolati. Inoltre, soluzioni quali Abstract Factory e Chain of Responsibility supportano rispettivamente la creazione strutturata degli oggetti e la gestione ordinata del flusso delle richieste. Infine, i pattern DTO e DAO consentono di separare la logica applicativa dal livello dati, semplificando le comunicazioni tra le diverse parti dell'architettura.

Singleton

Consiste nella creazione di un oggetto in un'unica istanza condivisa a livello globale, utile quando serve un unico punto di accesso a una risorsa. In Spring i bean sono, per impostazione predefinita, singleton, garantendo che componenti condivisi come i DataSource vengano creati una sola volta.

Facade

Offre un'interfaccia unificata e semplificata per interagire con un insieme complesso di classi o funzionalità, rendendo più semplice l'utilizzo del sistema. In Spring, i servizi spesso svolgono il ruolo di facciata verso repository e logiche interne, fornendo un accesso centralizzato alle operazioni del dominio. In React, una singola componente può raccogliere dati da più API e organizzarli per la visualizzazione.

Adapter

Permette a classi con interfacce non compatibili di collaborare, traducendo l'interfaccia di una classe in quella attesa dal client. In Spring Boot viene impiegato, ad esempio, per integrare servizi esterni che espongono dati in formati diversi, adattandoli a DAO o DTO già presenti nel sistema.

Bridge

Consente di separare un'astrazione dalla sua implementazione, permettendo a entrambe di evolvere in modo indipendente. In Spring è possibile definire un servizio astratto e fornirne diverse implementazioni, configurabili tramite bean, rendendo facile sostituire o estendere il comportamento.

Builder

Offre un metodo graduale e flessibile per creare oggetti complessi, mantenendo il codice del client più leggibile. In Spring Boot può essere utilizzato per costruire entità o DTO articolati partendo da informazioni parziali, evitando costruttori troppo voluminosi. In React, è utile per generare configurazioni articolate o insiemi di proprietà in modo chiaro e intuitivo.

Abstract Factory

Mette a disposizione un'interfaccia per creare famiglie di oggetti correlati senza specificare le classi concrete. In Spring Boot è possibile sfruttarlo definendo configurazioni di bean differenti in base al profilo applicativo attivo.

Chain of Responsibility

Instrada una richiesta attraverso una catena di handler, ognuno dei quali può gestirla o delegarla al successivo. In Spring Boot è spesso utilizzato nei filtri che elaborano le richieste HTTP in sequenza.

DTO (Data Transfer Object)

Sono strutture dati semplici, prive di logica di business, pensate per trasferire informazioni tra diversi livelli o servizi. In Spring Boot vengono impiegati soprattutto nei controller per lo scambio dati con il client React, mantenendo separata la logica dal modello di dominio.

DAO (Data Access Object)

Isola le operazioni di accesso ai dati (query, mapping, persistenza) in classi dedicate, semplificando la logica di business. In Spring, i repository basati su JPA o altri driver implementano di fatto il pattern DAO, offrendo un'interfaccia chiara per la gestione della persistenza.

1.4. Definizioni, acronimi e abbreviazioni

Di seguito è fornito un elenco degli acronimi, abbreviazioni con le relative definizioni utilizzati in questo documento:

Termine	Definizione
CRUD	Create, Read, Update, Delete – Operazioni di base per la gestione dei dati in un'applicazione.
DTO	Data Transfer Object – Struttura dati utilizzata per trasferire informazioni tra diversi livelli o servizi dell'applicazione.
DAO	Data Access Object – Pattern che isola i dettagli di accesso ai dati, come query SQL e mapping, all'interno di classi dedicate.
HTTP	HyperText Transfer Protocol – Protocollo di trasferimento dati utilizzato per le comunicazioni web.
JPA	Java Persistence API – Specifica Java per la gestione della persistenza dei dati tra le applicazioni Java e i database relazionali.
JSDoc	JavaScript Documentation – Strumento di documentazione per il linguaggio JavaScript, simile a Javadoc per Java.
Javadoc	Strumento di documentazione per il linguaggio Java, utilizzato per generare documentazione API a partire dal codice sorgente.
JWT	JSON Web Token – Standard aperto per la trasmissione sicura di informazioni tra le parti come oggetti JSON.
OCL	Object Constraint Language – Linguaggio utilizzato per specificare restrizioni e vincoli nei modelli UML.
REST	Representational State Transfer – Stile architetturale per la progettazione di servizi web che utilizza le operazioni HTTP.
UML	Unified Modeling Language – Linguaggio di modellazione standardizzato utilizzato per specificare, visualizzare, costruire e documentare gli artefatti di sistemi software.
API	Application Programming Interface – Insieme di regole e specifiche che le applicazioni possono seguire per comunicare tra loro.
CSS	Cascading Style Sheets – Linguaggio utilizzato per descrivere la presentazione di documenti HTML o XML.

1.5. Riferimenti

Il presente progetto si basa sull'analisi dei processi tipici delle concessionarie automobilistiche e sul confronto con gestionali del settore già consolidati, i quali hanno dimostrato grande efficacia nella gestione integrata di showroom, trattative e documentazione amministrativa.

Sono state inoltre esaminate piattaforme commerciali note per la gestione dei veicoli, dei clienti e delle vendite, al fine di individuare pratiche ottimali e funzionalità da integrare in AutoFlow.

Di seguito sono elencati i documenti chiave del progetto a cui si fa esplicito riferimento:

- **Problem Statement (PS):** documento che analizza le criticità operative delle concessionarie e descrive gli obiettivi che AutoFlow intende raggiungere per risolverle.
- **Requirement Analysis Document (RAD):** documento che definisce i requisiti funzionali, non funzionali e i vincoli del sistema AutoFlow, descrivendo inoltre scenari d'uso completi e casi d'uso dettagliati.
- **System Design Document (SDD):** documento che illustra l'architettura software proposta, la decomposizione in sottosistemi, la gestione della sicurezza, della persistenza e la mappatura hardware/software del progetto AutoFlow.

Oltre ai documenti progettuali, lo sviluppo metodologico e concettuale del sistema è supportato da opere tecniche e linee guida riconosciute nel settore dello sviluppo software:

- **Object-Oriented Software Engineering Using UML, Patterns, and Java™ — 3rd Edition:** riferimento per la modellazione UML, la progettazione orientata agli oggetti e la gestione dei processi ingegneristici.
- **Google Java Style Guide:** guida alle convenzioni di codifica Java, utile per garantire qualità, coerenza e leggibilità del codice lato back-end.
- **TypeScript Style Guide:** insieme di linee guida e best practice adottate per lo sviluppo del front-end in TypeScript.
- **React Style Guide:** riferimento per la progettazione dei componenti dell'interfaccia cliente/staff, favorendo modularità, performance e pulizia architetturale.
- **Spring Framework Code Style:** linee guida tecniche adottate per garantire uno sviluppo back-end coerente, sicuro e manutenibile all'interno dell'ecosistema Spring Boot.

2. Packages

L'applicazione AutoFlow adotta una struttura di package organizzata per garantire una chiara separazione delle responsabilità e facilitare la manutenzione del codice.

2.1 Back-End

Nel back-end, i package principali includono elementi come i controller, che gestiscono le richieste HTTP esposte tramite endpoint REST e delegano la logica di business ai service, contenuti in package dedicati. I package entity definiscono le tabelle del database utilizzando mappature JPA, rappresentando la struttura dei dati persistenti. Per trasferire dati tra componenti, i dto offrono una rappresentazione sicura e ottimizzata, mentre i repository, anch'essi organizzati in package, forniscono l'accesso ai dati persistenti tramite Spring Data JPA, supportando operazioni CRUD e query personalizzate. I package exception centralizzano la gestione degli errori, mentre i package config includono le configurazioni per aspetti come la sicurezza, il CORS, il mapping tra DTO e entità e l'accesso al database.

Di seguito sono descritti i principali package individuati per AutoFlow:

Authentication

Gestisce login, logout, registrazione cliente e recupero password, garantendo sicurezza tramite controlli sui ruoli e validazione delle credenziali.

Users

Contiene la gestione di amministratori, addetti vendite e clienti, includendo creazione, modifica, disattivazione, ruoli e permessi associati.

Showroom

Offre i servizi per la consultazione dello showroom virtuale, visualizzazione dei veicoli disponibili e accesso del cliente al configuratore.

Vehicle

Gestisce le schede dei veicoli con dati tecnici, prezzo, stato (disponibile/venduto) e optional correlati, permettendo ricerca e aggiornamento.

Configuration

Raccoglie la logica per la creazione e modifica delle configurazioni veicolo, scelta degli optional e calcolo dinamico del prezzo finale.

Proposal

Si occupa della creazione, visualizzazione, modifica e gestione degli stati delle proposte di vendita, sia lato cliente che lato staff.

Invoice

Gestisce la generazione delle fatture PDF, la numerazione progressiva, l'archiviazione dei documenti e l'associazione alla proposta finalizzata.

Statistics

Fornisce KPI, grafici e dati aggregati utili a valutare performance di vendita, trattative attive, veicoli disponibili e andamento mensile.

Struttura package backend

authentication

→ **controller**

→ **dto**

→ **entity**

→ **repository**

→ **service**

→ **utils**

user

→ **entity**

→ **dto**

→ **repository**

→ **service**

→ **controller**

showroom

→ **controller**

→ **dto**

→ **entity**

→ **repository**

→ **service**

vehicle

→ **entity**

→ **dto**

→ **repository**

→ **service**

→ **controller**

configuration

→ **entity**

→ **dto**

→ **repository**

→ **service**

→ **controller**

proposal

→ **entity**

→ **dto**

→ **repository**

→ **service**

→ **controller**

invoice

→ **entity**

→ **dto**

→ **repository**

→ **service**

→ **controller**

statistics

→ **dto**

→ **repository**

→ **service**

→ **controller**

commons

→ **dto**

→ **service**

→ **configuration**

→ **exception**

→ **security**

2.1.1. Elenco delle Classi e Interfacce di tipo Service e Controller

File	Package	Descrizione
CrudService<T, ID>	it.autoflow.commons.service	Servizio generico per la logica delle operazioni CRUD, integrato con i repository specifici dei vari moduli.
FileStorageService	it.autoflow.commons.service	Gestisce il salvataggio e il recupero dei file (es. PDF di proposte e fatture) nel file system locale, organizzati per anno/mese.
PdfDocumentService	it.autoflow.commons.service	Incapsula la logica di generazione dei documenti PDF (preventivi, proposte, fatture) a partire dai dati di dominio.
AuthenticationController	it.autoflow.authentication.controller	Controller REST per login e logout degli utenti (cliente, addetto vendite, amministratore).
RegisterController	it.autoflow.authentication.controller	Gestisce la registrazione del cliente nello showroom virtuale e la validazione dei dati inseriti.
ActivationAccountController	it.autoflow.authentication.controller	Gestisce la verifica del token e l'attivazione dell'account cliente tramite link ricevuto via e-mail.
PasswordRecoveryController	it.autoflow.authentication.controller	Espone le operazioni per l'avvio del recupero credenziali e il reset sicuro della password del cliente o dello staff.
AuthenticationService	it.autoflow.authentication.service	Servizio responsabile della verifica delle credenziali, della gestione della sessione e delle regole di sicurezza per l'accesso.

ClienteController	it.autoflow.user.controller	Controller per la gestione delle operazioni relative ai clienti (anagrafica, area personale, proposte e fatture associate).
ClienteService	it.autoflow.user.service	Gestisce la logica applicativa legata ai clienti, inclusa la consultazione di configurazioni, proposte e documenti dello storico.
AddettoVenditeController	it.autoflow.user.controller	Controller per l'amministrazione degli addetti vendite: creazione, disattivazione/riattivazione e cambio password.
AddettoVenditeService	it.autoflow.user.service	Servizio che gestisce la logica legata agli utenti di staff (permessi, ruoli, reset password, stato account).
ShowroomController	it.autoflow.showroom.controller	Controller per la visualizzazione dello showroom virtuale e la ricerca dei veicoli disponibili secondo filtri dinamici.
ShowroomService	it.autoflow.showroom.service	Fornisce i servizi di business per il caricamento dello showroom, l'applicazione dei filtri e l'esposizione dei veicoli lato pubblico.
VeicoloController	it.autoflow.vehicle.controller	Controller per la gestione delle schede veicolo: creazione, aggiornamento, duplicazione e cambio di stato (disponibile/venduto).
VeicoloService	it.autoflow.vehicle.service	Gestisce la logica dei veicoli, incluse le regole di unicità su targa/VIN e i controlli durante la modifica o duplicazione.

OptionalAccessorioController	it.autoflow.configuration.controller	Controller REST che espone le API per la gestione degli optional (lista, ricerca, creazione, modifica, eliminazione).
OptionalAccessorioService	it.autoflow.configuration.service	Servizio che gestisce gli optional dei veicoli, fornendo operazioni CRUD e ricerca per codice.
ConfigurazioneController	it.autoflow.configuration.controller	Controller per la creazione e modifica delle configurazioni veicolo, sia lato cliente sia lato addetto vendite.
ConfigurazioneService	it.autoflow.configuration.service	Servizio che implementa la logica del configuratore: gestione degli optional, calcolo del prezzo e validazione delle scelte.
PropostaController	it.autoflow.proposal.controller	Controller per la gestione completa del ciclo di vita delle proposte (creazione, aggiornamento, annullamento, finalizzazione).
PropostaService	it.autoflow.proposal.service	Gestisce la logica di business delle trattative, inclusa la transizione tra stati (creata, in attesa, accettata, confermata, completata).
FatturaController	it.autoflow.invoice.controller	Controller per la generazione delle fatture a partire da una proposta confermata e per la loro consultazione lato staff/cliente.
FatturaService	it.autoflow.invoice.service	Servizio che coordina la creazione della fattura, la numerazione progressiva e l'archiviazione del PDF nelle directory corrette.

StatisticsController	it.autoflow.statistics.controller	Controller che espone le API per la dashboard delle statistiche di vendita, KPI principali e grafici filtrabili.
StatisticsService	it.autoflow.statistics.service	Servizio che aggrega dati su veicoli, proposte e vendite per calcolare KPI globali e personali per periodo, veicolo e venditore.

2.1.2 – Elenco Classi e Interfacce di tipo Entity e JpaRepository

Nel presente paragrafo vengono elencate le classi di tipo Entity utilizzate dal sistema AutoFlow e le corrispondenti interfacce JpaRepository. Le repository sono collocate nel package repository, posto a un livello superiore rispetto all'entità di riferimento, al fine di mantenere una struttura coerente e facilmente navigabile. Per uniformità, ogni repository adotta il nome dell'entità con l'aggiunta del suffisso "Repository". Eventuali classi Embeddable legate a chiavi composte vengono inserite nel package entity associato.

File	Package
Cliente	it.autoflow.user.entity
AddettoVendite	it.autoflow.user.entity
Amministratore	it.autoflow.user.entity
Veicolo	it.autoflow.vehicle.entity
OptionalAccessorio	it.autoflow.configuration.entity
Configurazione	it.autoflow.configuration.entity
ConfigurazioneOptional	it.autoflow.configuration.entity
Proposta	it.autoflow.proposal.entity
PropostaOptional	it.autoflow.proposal.entity
Fattura	it.autoflow.invoice.entity
DocumentoPDF	it.autoflow.commons.entity

2.2 Front-End

La struttura del front-end è progettata per garantire modularità e riusabilità all'interno dell'interfaccia utente di AutoFlow. Gli elementi grafici e funzionali sono suddivisi in pacchetti dedicati, separando i componenti generici dell'interfaccia da quelli utilizzati per i form, le viste tematiche e le pagine principali. Le risorse statiche come immagini, icone e fogli di stile sono raccolte nel package assets, mentre la sezione pages contiene le schermate principali del sistema, organizzate per ruolo e funzionalità. Questa organizzazione permette una chiara separazione delle responsabilità e semplifica l'evoluzione dell'interfaccia nelle versioni successive.

src

- → **assets**
- → **components**
  - → **common**
  - → **forms**
  - → **layout**
- → **routes**
- → **entities**
- → **theme**
- → **styles**
- → **pages**
  - → **authentication**
  - → **user**
  - → **showroom**
  - → **vehicle**
  - → **configuration**
  - → **proposal**
  - → **invoice**
  - → **statistics**
- → **services**

3. Invarianti Classi

Di seguito sono riportate le invarianti, dove necessarie, per le entità individuate in fase di analisi dei requisiti. Non sono previste invarianti per le classi service.

	Invariante
Cliente	email contiene "@" AND dataRegistrazione ≤ oggi
AddettoVendite	ruolo ∈ {"Venditore", "ResponsabileVendite"}
Amministratore	privilegi = "FULL"
Veicolo	prezzo ≥ 0 AND stato ∈ {DISPONIBILE, VENDUTO}
OptionalAccessorio	prezzo ≥ 0
Configurazione	prezzoTotale = prezzoBase + optional.sum(prezzo)
Configurazione	modello ≠ null AND colore ≠ null
Proposta	totaleProposta ≥ 0 AND configurazione ≠ null
Proposta	stato ∈ {CREATA, ATTESA, CONFERMATA, ANNULLATA, COMPLETATA}
Fattura	numeroFattura rispetta formato "AF-AAAA-NNN"
Fattura	totaleFattura = configurazione.prezzoTotale
DocumentoPDF	path termina con ".pdf"

4. Interfaccia Classi

4.1. Interfacce

In questo capitolo vengono descritte le principali interfacce che compongono il sottosistema applicativo di AutoFlow. Tali interfacce definiscono i contratti funzionali tra i diversi moduli del sistema, in particolare tra controller, servizi e componenti di accesso ai dati, e costituiscono la base per un'implementazione coerente, modulare e facilmente estendibile.

AutoFlow adotta una separazione netta tra livello di presentazione, logica applicativa e persistenza, garantendo che ogni strato comunichi tramite interfacce chiaramente definite e non dipenda da implementazioni concrete.

I controller saranno implementati come @RestController Spring Boot, integrati con annotazioni specifiche di validazione e sicurezza, mentre la logica di business sarà affidata a classi @Service, progettate per mantenere bassa l'accoppiatura con il livello dei dati. Le entità individuate in fase di analisi dei requisiti vengono mappate tramite JPA, mentre gran parte della comunicazione tra front-end e back-end avverrà tramite DTO, che consentono di ridurre il traffico di rete, controllare le informazioni esposte e applicare diverse viste in base ai ruoli utente (cliente, addetto vendite, amministratore).

Tutti i repository estendono JpaRepository<T, ID>, seguendo lo stile già impiegato nei documenti progettuali. Le query personalizzate sono dichiarate direttamente nell'interfaccia del repository o tramite annotazioni specifiche, così da rendere esplicito il comportamento e facilitare la tracciabilità delle operazioni sui dati.

Per alcuni servizi trasversali, come la generazione dei documenti PDF o la gestione dell'archiviazione mensile, saranno implementate interfacce dedicate, così da isolare la logica tecnica e permettere l'integrazione futura di alternative (es. archiviazione cloud o sistemi di terze parti).

Le componenti TypeScript del front-end non sono qui dettagliate, poiché fortemente legate al layout delle pagine applicative; tuttavia si prevede la realizzazione di un livello client-side service che fornisca un accesso uniforme ai vari endpoint REST, riducendo la duplicazione del codice e semplificando la gestione degli errori e della validazione lato utente. L'integrità dei dati viene inoltre supportata da validazioni front-end basate sulla libreria Yup, in linea con le precondizioni definite nei casi d'uso e nei vincoli modellati nelle OCL presenti nel RAD.

Le sezioni successive presentano in dettaglio ciascuna interfaccia, specificando operazioni, precondizioni, postcondizioni e comportamenti attesi, così da fornire un riferimento chiaro e completo per l'implementazione coerente dell'intero sistema AutoFlow.

4.1.1. CrudService<T, ID>

Interfaccia generica per operazioni CRUD (Create, Read, Update, Delete) su un'entità di tipo T, identificata da un ID di tipo ID. Tutti i servizi di dominio che gestiscono entità persistenti estendono questa interfaccia.

Nome	Descrizione	Precondizione	Post condizione
+ create(entity: T): T	Crea una nuova entità di tipo T.	**context** CrudService::create(entity: T): T **pre** not entity.oclIsUndefined() and T::allInstances()->forAll(e | e.id <> entity.id)	**context** CrudService::create(entity: T): T **post** T::allInstances()->exists(e | e.id = result.id) and result = entity
+ getById(id: ID): T	Restituisce l'entità con identificativo id.	**context** CrudService::getById(id: ID): T **pre** not id.oclIsUndefined()	**context** CrudService::getById(id: ID): T **post** (T::allInstances()->exists(e | e.id = id) implies result.id = id) and (not T::allInstances()->exists(e | e.id = id) implies result.oclIsUndefined())
+ update(id: ID, entity: T): T	Aggiorna i dati di un'entità esistente.	**context** CrudService::update(id: ID, entity: T): T **pre** not id.oclIsUndefined() and not entity.oclIsUndefined() and T::allInstances()->exists(e | e.id = id)	**context** CrudService::update(id: ID, entity: T): T **post** T::allInstances()->exists(e | e.id = id and e = result) and result.id = id

+ delete(id: ID): Boolean	Elimina l'entità con ID id.	**context** CrudService::delete(id: ID): Boolean **pre** not id.oclIsUndefined() and T::allInstances()->exists(e | e.id = id)	**context** CrudService::delete(id: ID): Boolean **post** result = true implies not T::allInstances()->exists(e | e.id = id)
+ findAll(): Bag(T)	Restituisce tutte le entità di tipo T.	**context** CrudService::findAll(): Bag(T) **pre** true	**context** CrudService::findAll(): Bag(T) **post** result->forAll(e | T::allInstances()->exists(x | x.id = e.id))

4.1.2. FileStorageService

Servizio per la gestione dello storage dei file (ad esempio PDF di proposte e fatture) nel file system locale, organizzato per anno/mese.

Nome	Descrizione	Precondizione	Post condizione
+ store(file: File, folderKey: String): String	Salva un file nella directory logica identificata da folderKey (es. anno/mese) e restituisce il percorso logico.	**context** FileStorageService::store (file: File, folderKey: String): String **pre** not file.oclIsUndefined() and file.size <= 16 * 1024 * 1024 and folderKey.size() > 0	**context** FileStorageService::store (file: File, folderKey: String): String **post** result.size() > 0 and FileSystem::exists(result) = true
+ load(path: String): File	Carica un file a partire dal percorso logico path.	**context** FileStorageService::load(path: String): File **pre** path.size() > 0 and FileSystem::exists(path) = true	**context** FileStorageService::load(path: String): File **post** not result.oclIsUndefined()
+ delete(path: String): Boolean	Elimina il file associato al percorso path.	**context** FileStorageService::delet e(path: String): Boolean **pre** path.size() > 0	**context** FileStorageService::delet e(path: String): Boolean **post** result = true implies FileSystem::exists(path) = false

4.1.3. PdfDocumentService

Servizio che incapsula la logica di generazione dei documenti PDF (preventivi, proposte, fatture) a partire dai dati di dominio.

Nome	Descrizione	Precondizione	Post condizione
+ generateProposalPdf(propostaId: Long): DocumentoPDF	Genera il PDF associato a una proposta.	**context** PdfDocumentService::generateProposalPdf(propostaId: Long): DocumentoPDF **pre** Proposta::allInstances()->exists(p | p.id = propostaId)	**context** PdfDocumentService::generateProposalPdf(propostaId: Long): DocumentoPDF **post** DocumentoPDF::allInstances()->exists(d | d.id = result.id and d.proposta.id = propostaId)
+ generateInvoicePdf(fatturaId: Long): DocumentoPDF	Genera il PDF associato a una fattura.	**context** PdfDocumentService::generateInvoicePdf(fatturaId: Long): DocumentoPDF **pre** Fattura::allInstances()->exists(f | f.id = fatturaId)	**context** PdfDocumentService::generateInvoicePdf(fatturaId: Long): DocumentoPDF **post** DocumentoPDF::allInstances()->exists(d | d.id = result.id and d.fattura.id = fatturaId)
+ regeneratePdf(documentoId: Long): DocumentoPDF	Rigenera un PDF esistente (ad esempio dopo modifica dei dati).	**context** PdfDocumentService::regeneratePdf(documentoId: Long): DocumentoPDF **pre** DocumentoPDF::allInstances()->exists(d | d.id = documentoId)	**context** PdfDocumentService::regeneratePdf(documentoId: Long): DocumentoPDF **post** DocumentoPDF::allInstances()->exists(d | d.id = documentoId and d.lastUpdated >= Date::now())

4.1.4. AuthenticationService

Servizio responsabile della verifica delle credenziali, della gestione della sessione e delle regole di sicurezza per l'accesso.

Nome	Descrizione	Precondizione	Post condizione
+ login(request: LoginRequestDTO): LoginResponseDTO	Autentica un utente (cliente / addetto / amministratore) e restituisce le informazioni di login (token, ruolo, dati utente).	**context** AuthenticationService::login(request: LoginRequestDTO): LoginResponseDTO **pre** not request.oclIsUndefined() and request.email.size() > 0 and request.password.size() > 0	**context** AuthenticationService::login(request: LoginRequestDTO): LoginResponseDTO **post** result.oclIsUndefined() = false implies result.token.size() > 0
+ logout(token: String): Boolean	Invalida il token di sessione fornito.	**context** AuthenticationService::logout(token: String): Boolean **pre** token.size() > 0	**context** AuthenticationService::logout(token: String): Boolean **post** result = true implies not Session::allInstances()->exists(s | s.token = token and s.valid = true)
+ registerCliente(dto: RegisterClienteDTO): Void	Registra un nuovo cliente nello showroom virtuale.	**context** AuthenticationService::registerCliente(dto: RegisterClienteDTO): Void **pre** not dto.oclIsUndefined() and dto.email.size() > 0	**context :** AuthenticationService::registerCliente(dto: RegisterClienteDTO): Void **post** Cliente::allInstances()->exists(c | c.email = dto.email)

+ activateAccount(token: String): Boolean	Attiva l'account di un cliente tramite il token ricevuto via e-mail.	**context** AuthenticationService:: activateAccount(token: String): Boolean **pre :** token.size() > 0	**context** AuthenticationService:: activateAccount(token: String): Boolean **post** result = true implies Cliente::allInstances()->exists(c | c.tokenAttivazione = token and c.attivo = true)
+ requestPasswordReset(email: String): Void *(sostituito implementazione futura)*	Avvia la procedura di reset password inviando una mail con token.	**context** AuthenticationService::requestPasswordReset(email: String): Void **pre** email.size() > 0 and Utente::allInstances()->exists(u | u.email = email)	**context** AuthenticationService::requestPasswordReset(email: String): Void **post** PasswordResetToken::allInstances()->exists(t | t.utente.email = email and t.valid = true)
+ requestPasswordReset(email: String): PasswordResetResultDTO	Avvia la procedura di reset password impostando una password di default a differenza dell'utente che lo chiede	**context** AuthenticationService::requestPasswordReset(email: String): PasswordResetResultDTO **pre** email.size() > 0 and Utente::allInstances()->exists(u | u.email = email)	**context** AuthenticationService::requestPasswordReset(email: String): PasswordResetResultDTO **post** result = true implies Utente::allInstances()->exists(u | u.password = defaultPassword and u.attivo = false)

+ confirmPasswordReset (dto: PasswordResetConfirmDTO): Boolean	Conferma il reset della password utilizzando il token e la nuova password.	**context** AuthenticationService::confirmPasswordReset(dto: PasswordResetConfirmDTO): Boolean **pre** not dto.oclIsUndefined() and dto.token.size() > 0 and dto.nuovaPassword.size() > 0	**context** AuthenticationService::confirmPasswordReset(dto: PasswordResetConfirmDTO): Boolean **post** result = true implies Utente::allInstances()->exists(u | u.email = dto.email and u.password = dto.nuovaPasswordHash)
+ changePasswordAfterReset (dto: PasswordFirstChangeDTO): boolean	Metodo per il cambio password dopo il primo accesso dopo aver richiesto il reset	**context** AuthenticationService::changePasswordAfterReset (dto: PasswordFirstChangeDTO): boolean **pre** not dto.oclIsUndefined() and dto.token.size() > 0 and dto.nuovaPassword.size() > 0	**context** AuthenticationService::changePasswordAfterReset (dto: PasswordFirstChangeDTO): boolean **post** result = true implies Utente::allInstances()->exists(u | u.id = dto.id and u.password = dto.nuovaPasswordHash and u.attivo =true)
+ getUserFromToken(token: String): User	Restituisce l'utente associato a un token di autenticazione.	**context** AuthenticationService::getUserFromToken(token: String): User **pre :** token.size() > 0	**context :** AuthenticationService::getUserFromToken(token: String): User **post** result.oclIsUndefined() = false implies result.id > 0

4.1.5. ClienteService extends CrudService<Cliente, Long>

Servizio che gestisce la logica applicativa legata ai clienti, inclusa la consultazione di configurazioni, proposte e documenti dello storico.

Nome	Descrizione	Precondizione	Post condizione
+ getStoricoConfigurazioni(clienteId: Long): Bag(Configurazione)	Restituisce tutte le configurazioni salvate da un cliente.	**context** ClienteService::getStoricoConfigurazioni(clienteId: Long): Bag(Configurazione) **pre** Cliente::allInstances()->exists(c | c.id = clienteId)	**context** ClienteService::getStoricoConfigurazioni(clienteId: Long): Bag(Configurazione) **post** result->forAll(cfg | Configurazione::allInstances()->exists(c | c.id = cfg.id and c.cliente.id = clienteId))
+ getStoricoProposte(clienteId: Long): Bag(Proposta)	Restituisce tutte le proposte associate a un cliente.	**context** ClienteService::getStoricoProposte(clienteId: Long): Bag(Proposta) **pre** Cliente::allInstances()->exists(c | c.id = clienteId)	**context** ClienteService::getStoricoProposte(clienteId: Long): Bag(Proposta) **post** result->forAll(p | Proposta::allInstances()->exists(x | x.id = p.id and x.cliente.id = clienteId))
+ getStoricoFatture(clienteId: Long): Bag(Fattura)	Restituisce le fatture emesse per il cliente.	**context** ClienteService::getStoricoFatture(clienteId: Long): Bag(Fattura) **pre** Cliente::allInstances()->exists(c | c.id = clienteId)	**context** ClienteService::getStoricoFatture(clienteId: Long): Bag(Fattura) **post** result->forAll(f | Fattura::allInstances()->exists(x | x.id = f.id and x.cliente.id = clienteId))

4.1.6. AddettoVenditeService extends CrudService<AddettoVendite, Long>

Servizio che gestisce la logica legata agli addetti alle vendite (creazione, attivazione, gestione delle credenziali e performance).

Nome	Descrizione	Precondizione	Post condizione
+ resetPassword(addettoId: Long): Boolean	Reimposta la password di un addetto vendite e registra l'operazione.	**context** AddettoVenditeService::resetPassword(addettoId: Long): Boolean **pre** AddettoVendite::allInstances()->exists(a | a.id = addettoId)	**context** AddettoVenditeService::resetPassword(addettoId: Long): Boolean **post** result = true implies AddettoVendite::allInstances()->exists(a | a.id = addettoId and a.passwordChangedRecently = true)
+ deactiveAddetto(addettoId: Long): AddettoVendite	Attiva/disattiva un addetto vendite.	**context** AddettoVenditeService::deactiveAddetto(addettoId: Long): AddettoVendite **pre** AddettoVendite::allInstances()->exists(a | a.id = addettoId)	**context** AddettoVenditeService::deactiveAddetto(addettoId: Long): AddettoVendite **post** result.id = addettoId and result.attivo <> AddettoVendite::allInstances()->any(a | a.id = addettoId)@pre.attivo
+ getPerformance(addettoId: Long, periodo: Periodo): VenditeStats	Restituisce KPI di vendita per un addetto su un dato periodo.	**context** AddettoVenditeService::getPerformance(addettoId: Long, periodo: Periodo): VenditeStats **pre** AddettoVendite::allInstances()->exists(a | a.id = addettoId)	**context** AddettoVenditeService::getPerformance(addettoId: Long, periodo: Periodo): VenditeStats **post** result.addettoId = addettoId

4.1.7. ShowroomService

Servizio che fornisce le operazioni di business per il caricamento dello showroom virtuale e l'applicazione dei filtri lato pubblico.

Nome	Descrizione	Precondizione	Post condizione
+ search(filtro: ShowroomFiltro): Bag(Veicolo)	Restituisce l'elenco dei veicoli disponibili che rispettano i filtri impostati (marca, modello, prezzo, alimentazione, ecc.).	**context** ShowroomService::search(filtro: ShowroomFiltro): Bag(Veicolo) **pre** not filtro.oclIsUndefined()	**context** ShowroomService::search(filtro: ShowroomFiltro): Bag(Veicolo) **post** result->forAll(v | v.stato = VeicoloStato::DISPONIBILE)
+ getDettaglioPubblico(veicoloId: Long): Veicolo	Restituisce i dettagli "pubblici" di un veicolo mostrato nello showroom.	**context** ShowroomService::getDettaglioPubblico(veicoloId: Long): Veicolo **pre** Veicolo::allInstances()->exists(v | v.id = veicoloId and v.stato = VeicoloStato::DISPONIBILE)	**context** ShowroomService::getDettaglioPubblico(veicoloId: Long): Veicolo **post** result.id = veicoloId

4.1.8. VeicoloService extends CrudService<Veicolo, Long>

Servizio per la gestione della logica dei veicoli, incluse le regole di unicità e il cambio di stato (disponibile/venduto).

Nome	Descrizione	Precondizione	Post condizione
+ changeState(veicoloId: Long, nuovoStato: VeicoloStato): Veicolo	Cambia lo stato di un veicolo (es. DISPONIBILE → VENDUTO).	**context** VeicoloService::changeState(veicoloId: Long, nuovoStato: VeicoloStato): Veicolo **pre** Veicolo::allInstances()->exists(v | v.id = veicoloId)	**context** VeicoloService::changeState(veicoloId: Long, nuovoStato: VeicoloStato): Veicolo **post** result.id = veicoloId and result.stato = nuovoStato
+ duplicate(veicoloId: Long): Veicolo	Duplica un veicolo per creare una nuova scheda con dati simili.	**context** VeicoloService::duplicate(veicoloId: Long): Veicolo **pre** Veicolo::allInstances()->exists(v | v.id = veicoloId)	**context** **VeicoloService::duplicate(veicoloId: Long): Veicolo** **post** Veicolo::allInstances()->exists(v | v.id = result.id and v.id <> veicoloId) and result.modello = Veicolo::allInstances()->any(v | v.id = veicoloId).modello
+ checkUnicitaTargaVin(targa: String, vin: String): Boolean	Verifica l'unicità di targa e VIN per un nuovo veicolo.	**context** VeicoloService::checkUnicitaTargaVin(targa: String, vin: String): Boolean **pre** targa.size() > 0 and vin.size() > 0	**context** VeicoloService::checkUnicitaTargaVin(targa: String, vin: String): Boolean **post** result = true implies not Veicolo::allInstances()->exists(v | v.targa = targa or v.vin = vin)

4.1.9. ConfigurazioneService extends CrudService<Configurazione, Long>

Servizio che implementa la logica del configuratore (gestione optional, calcolo prezzo, validazioni e associazione al cliente).

Nome	Descrizione	Precondizione	Post condizione
+ creaConfigurazione(dto: ConfigurazioneDTO): Configurazione	Crea una nuova configurazione veicolo a partire dai dati del DTO.	**context** ConfigurazioneService::creaConfigurazione(dto: ConfigurazioneDTO): Configurazione **pre** not dto.oclIsUndefined()	**context** ConfigurazioneService::creaConfigurazione(dto : ConfigurazioneDTO): Configurazione **post** Configurazione::allInstances()->exists(c | c.id = result.id)
+ getById(id: Long): Configurazione	Restituisce la configurazione con l'identificativo indicato.	**context** ConfigurazioneService::getById(id: Long): Configurazione **pre** not id.oclIsUndefined() and Configurazione::allInstances()->exists(c | c.id = id)	**context** ConfigurazioneService::getById(id: Long): Configurazione **post** result.oclIsUndefined() = false and result.id = id
+ getAll(): Bag(Configurazione)	Restituisce tutte le configurazioni presenti a sistema.	**context** ConfigurazioneService::getAll(): Bag(Configurazione) **pre** true	**context** ConfigurazioneService::getAll(): Bag(Configurazione) **post** result->forAll(cfg | Configurazione::allInstances()->exists(c | c.id = cfg.id))

+ update(id: Long, dto: ConfigurazioneDTO): Configurazione	Aggiorna i dati di una configurazione esistente.	**context** ConfigurazioneService::update(id: Long, dto: ConfigurazioneDTO): Configurazione **pre** not id.oclIsUndefined() and not dto.oclIsUndefined() and Configurazione::allInstances()->exists(c | c.id = id)	**context** ConfigurazioneService::update(id: Long, dto: ConfigurazioneDTO): Configurazione **post** result.id = id
+ delete(id: Long): Boolean	Elimina la configurazione con l'id indicato.	**context** ConfigurazioneService::delete(id: Long): Boolean **pre** not id.oclIsUndefined() and Configurazione::allInstances()->exists(c | c.id = id)	**context** ConfigurazioneService::delete(id: Long): Boolean **post** result = true implies not Configurazione::allInstances()->exists(c | c.id = id)

4.1.10. PropostaService extends CrudService<Proposta, Long>

Servizio per la gestione del ciclo di vita delle proposte (creazione, aggiornamento stato, conferma, annullamento, completamento).

Nome	Descrizione	Precondizione	Post condizione
+ findAll(): Bag(Proposta)	Restituisce tutte le proposte presenti a sistema.	**context** PropostaService::findAll(): Bag(Proposta) **pre** true	**context** PropostaService::findAll(): Bag(Proposta) **post** result->forAll(p | Proposta::allInstances()->exists(x | x.id = p.id))
+ findById(id: Long): Proposta	Restituisce la proposta con l'identificativo indicato.	**context** PropostaService::findById(id: Long): Proposta **pre** not id.oclIsUndefined() and Proposta::allInstances()->exists(p | p.id = id)	**context** PropostaService::findById(id: Long): Proposta **post** result.oclIsUndefined() = false and result.id = id
+ findByClienteId(clienteId: Long): Bag(Proposta)	Restituisce le proposte associate a un determinato cliente.	**context** PropostaService::findByClienteId(clienteId: Long): Bag(Proposta) **pre** Cliente::allInstances()->exists(c | c.id = clienteId)	**context** PropostaService::findByClienteId(clienteId: Long): Bag(Proposta) **post** result->forAll(p | Proposta::allInstances()->exists(x | x.id = p.id and x.cliente.id = clienteId))
+ findByAddettoVenditeId(addettoId: Long): Bag(Proposta)	Restituisce le proposte prese in carico da un addetto vendite.	**context** PropostaService::findByAddettoVenditeId(addettoId: Long): Bag(Proposta) **pre** AddettoVendite::allInstances()->exists(a | a.id = addettoId)	**context** PropostaService::findByAddettoVenditeId(addettoId: Long): Bag(Proposta) **post** result->forAll(p | Proposta::allInstances()->exists(x | x.id = p.id and x.addetto.id = addettoId))

+ create(dto: PropostaDTO): Proposta	Crea una nuova proposta a partire dal DTO.	**context** PropostaService::create(dto: PropostaDTO): Proposta **pre** not dto.oclIsUndefined()	**context** PropostaService::create(dto: PropostaDTO): Proposta **post** Proposta::allInstances()->exists(p | p.id = result.id)
+ update(id: Long, dto: PropostaDTO): Proposta	Aggiorna i dati di una proposta esistente (stato, importi, assegnatario…).	**context** PropostaService::update(id: Long, dto: PropostaDTO): Proposta **pre** not id.oclIsUndefined() and not dto.oclIsUndefined() and Proposta::allInstances()->exists(p | p.id = id)	**context** PropostaService::update(id: Long, dto: PropostaDTO): Proposta **post** result.id = id
+ delete(id: Long): Boolean	Elimina una proposta, se consentito dalle regole di business.	**context** PropostaService::delete(id: Long): Boolean **pre** id.oclIsUndefined() and Proposta::allInstances()->exists(p | p.id = id)	**context** PropostaService::delete(id: Long): Boolean **post** result = true implies not Proposta::allInstances()->exists(p | p.id = id)

4.1.11. FatturaService extends CrudService<Fattura, Long>

Servizio per la creazione delle fatture a partire dalle proposte confermate, la gestione della numerazione progressiva e il collegamento con i documenti PDF archiviati.

Nome	Descrizione	Precondizione	Post condizione
+ findAll(): Bag(Fattura)	Restituisce tutte le fatture presenti a sistema.	**context** FatturaService::findAll() : Bag(Fattura) **pre** true	**context** FatturaService::findAll() : Bag(Fattura) **post** result->forAll(f | Fattura::allInstances()->exists(x | x.id = f.id))
+ findById(id: Long): Fattura	Restituisce la fattura con l'identificativo indicato.	**context** FatturaService::findById(id: Long): Fattura **pre** not id.oclIsUndefined() and Fattura::allInstances()->exists(f | f.id = id)	**context** FatturaService::findById(id: Long): Fattura **post** result.oclIsUndefined() = false and result.id = id
+ findByNumeroFattura(numero: String): Fattura	Cerca una fattura in base al suo numero progressivo.	**context** FatturaService::findByNumeroFattura(numero: String): Fattura **pre** numero.size() > 0	**context** FatturaService::findByNumeroFattura(numero: String): Fattura **post** (Fattura::allInstances()->exists(f | f.numero = numero) implies result.numero = numero)
+ findByClienteId(clienteId: Long): Bag(Fattura)	Restituisce tutte le fatture associate a un determinato cliente.	**context** FatturaService::findByClienteId(clienteId: Long): Bag(Fattura) **pre** Cliente::allInstances()->exists(c | c.id = clienteId)	**context** FatturaService::findByClienteId(clienteId: Long): Bag(Fattura) **post** result->forAll(f | Fattura::allInstances()->exists(x | x.id = f.id and x.cliente.id = clienteId))

+ create(dto: FatturaDTO): Fattura	Crea una nuova fattura a partire dai dati presenti nel DTO.	**context** FatturaService::create(dto: FatturaDTO): Fattura **pre** not dto.oclIsUndefined()	**context** FatturaService::create(dto: FatturaDTO): Fattura **post** Fattura::allInstances()->exists(f | f.id = result.id)
+ update(id: Long, dto: FatturaDTO): Fattura	Aggiorna i dati di una fattura esistente.	**context** FatturaService::update(id: Long, dto: FatturaDTO): Fattura **pre** not id.oclIsUndefined() and not dto.oclIsUndefined() and Fattura::allInstances()->exists(f | f.id = id)	**context** FatturaService::update(id: Long, dto: FatturaDTO): Fattura **post** result.id = id
+ delete(id: Long): Boolean	Elimina la fattura con l'id indicato, se esistente.	**context** FatturaService::delete(id: Long): Boolean **pre** not id.oclIsUndefined() and Fattura::allInstances()->exists(f | f.id = id)	**context** FatturaService::delete(id: Long): Boolean **post** result = true implies not Fattura::allInstances()->exists(f | f.id = id)
+ createFromProposta(id: Long) : FatturaDTO	Crea una fattura da una proposta esistente.	**context** FatturaService:: createFromProposta(id: Long) : FatturaDTO **pre** not id.oclIsUndefined() and Proposta::allInstances()->exists(p | p.id = id)	**context** FatturaService:: createFromProposta(id: Long) : FatturaDTO **post** Fattura::allInstances()->exists(f | f.id = result.id)

4.1.12. StatisticsService

Servizio che aggrega dati su veicoli, proposte e vendite per il calcolo di KPI globali e individuali (ad esempio per addetto vendite), e per la visualizzazione della dashboard statistica.

Nome	Descrizione	Precondizione	Post condizione
+ getDashboardStatistics (): DashboardStatisticsDTO	Restituisce le statistiche aggregate per la dashboard principale (KPI globali di veicoli, proposte e fatture).	**context** StatisticsService::getDashboardStatistics(): DashboardStatisticsDTO **pre** true	**context** StatisticsService::getDashboardStatistics(): DashboardStatisticsDTO **post** result.oclIsUndefined() = false

4.1.13. OptionalAccessorioService

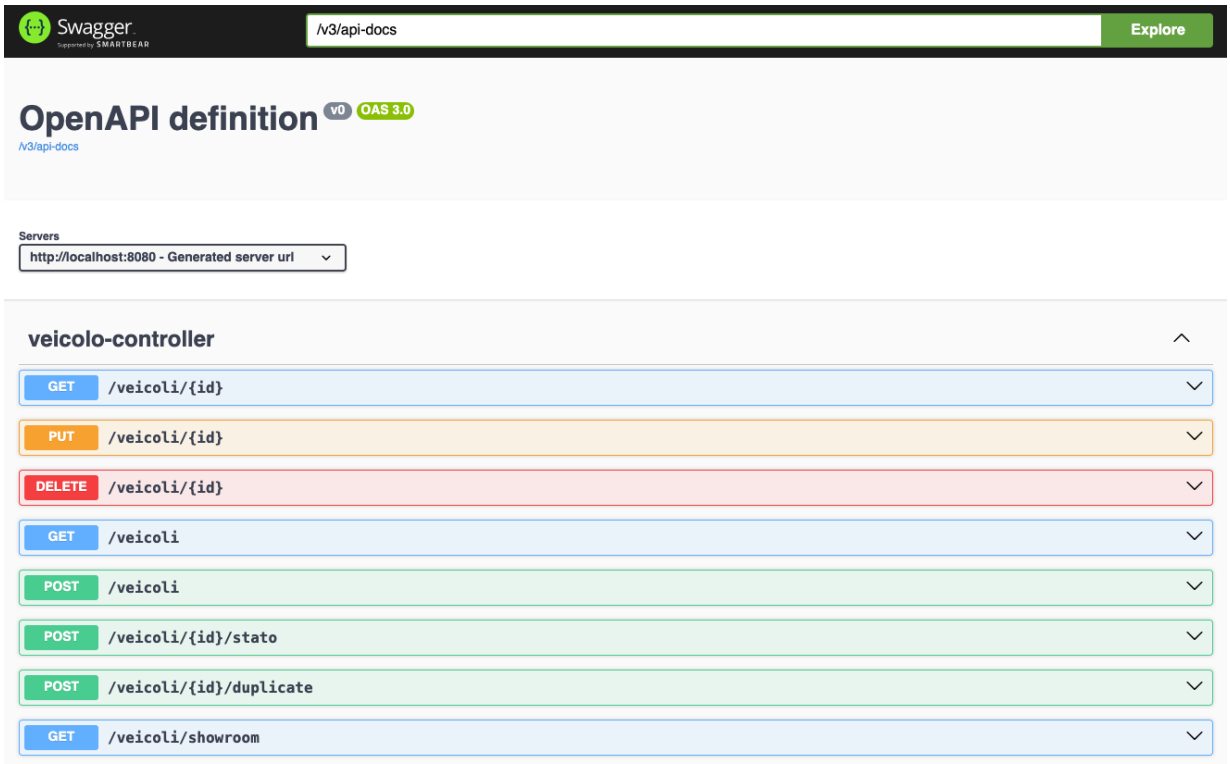
Servizio per la gestione degli optional aggiuntivi associati ai veicoli. Fornisce operazioni CRUD, ricerca tramite ID o codice univoco e validazione dei dati relativi agli optional utilizzati nel configuratore.

Nome	Descrizione	Precondizione	Post condizione
+ findAll() : Bag(OptionalAccessorioDTO)	Restituisce la lista completa degli optional disponibili.	**context** OptionalAccessorioService::findAll() : Bag(OptionalAccessorioDTO) **pre** true	**context** OptionalAccessorioService::findAll() : Bag(OptionalAccessorioDTO) **post** result->forAll(o | OptionalAccessorio::allInstances()->exists(x | x.id = o.id))
+ findById(id : Long) : OptionalAccessorioDTO	Restituisce l'optional corrispondente all'identificativo fornito.	**context** OptionalAccessorioService::findById(id : Long) : OptionalAccessorioDTO **pre** not id.oclIsUndefined()	**context** OptionalAccessorioService::findById(id : Long) : OptionalAccessorioDTO **Post** (OptionalAccessorio::allInstances()->exists(o | o.id = id) implies result.id = id)
+ findByCodice(codice : String) : OptionalAccessorioDTO	Restituisce l'optional tramite codice univoco.	**context** OptionalAccessorioService::findByCodice(codice : String) : OptionalAccessorioDTO **pre** codice.size() > 0	**context** OptionalAccessorioService::findByCodice(codice : String) : OptionalAccessorioDTO **post** (OptionalAccessorio::allInstances()->exists(o | o.codice = codice) implies result.codice = codice)

+ create(dto : OptionalAccessorioDTO) : OptionalAccessorioDTO	Crea un nuovo optional con i dati forniti.	**context** OptionalAccessorioService::create(dto : OptionalAccessorioDTO) : OptionalAccessorioDTO **pre** not dto.oclIsUndefined()and dto.codice.size() > 0and OptionalAccessorio::allInstances()->forAll(o | o.codice <> dto.codice)	**context** OptionalAccessorioService::create(dto : OptionalAccessorioDTO) : OptionalAccessorioDTO **post** OptionalAccessorio::allInstances()->exists(o | o.id = result.id and o.codice = dto.codice)
+ update(id : Long, dto : OptionalAccessorioDTO) : OptionalAccessorioDTO	Aggiorna un optional esistente.	**context** OptionalAccessorioService::update(id : Long, dto : OptionalAccessorioDTO) : OptionalAccessorioDTO **pre** not id.oclIsUndefined()and not dto.oclIsUndefined()and OptionalAccessorio::allInstances()->exists(o | o.id = id)	**context** OptionalAccessorioService::update(id : Long, dto : OptionalAccessorioDTO) : OptionalAccessorioDTO **post** OptionalAccessorio::allInstances()->exists(o | o.id = id and o.codice = dto.codice)
+ delete(id : Long) : Boolean	Elimina l'optional identificato da *id*.	**context** OptionalAccessorioService::delete(id : Long) : Boolean **pre** not id.oclIsUndefined() and OptionalAccessorio::allInstances()->exists(o | o.id = id)	**context** OptionalAccessorioService::delete(id : Long) : Boolean **post** result = true implies not OptionalAccessorio::allInstances()->exists(o | o.id = id

4.2. Specifica DTO e documentazione REST

L'API RESTful del sistema è documentata utilizzando Swagger UI, uno strumento interattivo che consente di esplorare e testare le API direttamente dal browser. La documentazione è accessibile tramite l'interfaccia utente di Swagger al seguente percorso: /swagger-ui/index.html



4.3. Componenti React

L'architettura front-end di AutoFlow sfrutta una struttura modulare basata su componenti React altamente riutilizzabili. I componenti sono progettati per garantire coerenza visiva, separazione delle responsabilità e semplicità di manutenzione, supportando sia l'esperienza pubblica (showroom) sia le aree riservate (clienti e addetti vendite).

Di seguito è riportata una lista strutturata dei componenti React attualmente utilizzati o riusabili all'interno dell'applicazione.

Componenti Comuni

- AppNavbar: barra di navigazione principale dell'applicazione, visualizzata nelle sezioni riservate.
- HomeNavbar: barra di navigazione della homepage pubblica dello showroom.
- HomeFooter: footer comune di tutte le pagine pubbliche.
- Login: pagina di accesso per clienti e addetti alle vendite.
- RegisterCliente: form di registrazione cliente, usato nel flusso.
- AccountSettings: pagina delle impostazioni account (cambio password, preferenze).
- Accordion (Open/Closed): componenti a fisarmonica utilizzati per presentare gruppi di informazioni espandibili.
- GenericTable: tabella generica configurabile per visualizzare dati eterogenei.
- GridList: visualizzazione a griglia per card (veicoli, optional, elementi dello showroom).
- ListGeneric: componente generico per visualizzare liste di elementi.
- FileUploader: componente riusabile per caricare immagini dei veicoli o documenti (PDF).
- Modal: finestra modale configurabile (conferme, dettagli, warning).
- NotificationBanner: componente per mostrare messaggi di sistema (successo/errore).
- LoadingSpinner: componente di caricamento globale.

Componenti Specifici Area Pubblica / Showroom

- ShowroomSearchBar: barra di ricerca con filtri (marca, modello, prezzo, alimentazione).
- ShowroomCard: card di anteprima veicolo (immagine, prezzo, modello, stato).
- ShowroomList: elenco o griglia dei veicoli disponibili.
- VeicoloPublicDetails: pagina con i dettagli pubblici del veicolo, immagini e specifiche.
- VeicoloGallery: galleria immagini del veicolo.

Componenti Configuratore Veicoli

- ConfiguratorWrapper: contenitore principale del configuratore.
- ConfigurazioneStepSelector: barra dei passaggi (sezioni della configurazione).
- OptionalSelector: componente per la selezione degli optional disponibili per un veicolo.
- OptionalCard: card dell'optional singolo.
- RiepilogoConfigurazione: pannello riepilogativo con prezzo base, optional selezionati e totale.
- ConfigSalvataggio: componente per salvare la configurazione nel profilo cliente.
- ConfigConferma: pulsante/box per procedere alla creazione della proposta.

Componenti Area Cliente

- ClienteDashboard: dashboard principale del cliente.
- ClienteSideNav: menu laterale per navigare tra le sezioni cliente.
- StoricoConfigurazioni: pagina che mostra tutte le configurazioni salvate.
- StoricoProposte: pagina per consultare lo storico delle proposte del cliente.
- StoricoFatture: pagina delle fatture emesse e scaricabili.
- PropostaClienteDetails: pagina con il dettaglio di una proposta (stato, importi, optional).
- FatturaViewer: visualizzatore dei PDF generati (fatture e proposte).
- AccountCliente: gestione dati personali e preferenze.

Componenti Area Addetto Vendite

- VenditoreDashboard: pagina principale dell'addetto vendite con KPI sintetici.
- VenditoreNavbar: barra di navigazione superiore dedicata allo staff.
- VenditoreSideNav: menu laterale con le funzionalità dedicate.
- ListaProposteAssegnate: elenco delle proposte in carico all'addetto.
- PropostaEditor: editor per modificare e aggiornare una proposta in sede.
- PropostaInSedeCreation: pagina per creare una proposta in sede.
- ClienteLookup: componente di ricerca cliente.
- VeicoloLookup: componente di ricerca veicolo.
- UploadVeicoloImages: componente per caricare immagini veicolo.
- GestioneVeicoloForm: form per creare/modificare un veicolo.
- DuplicazioneVeicolo: pagina per duplicare una scheda veicolo.
- ImpostazioneTarga: form per aggiungere targa/VIN.
- PropostaStatoManager: componente per aggiornare lo stato della proposta.

Componenti Specifici per la Fatturazione

- FatturaGenerator: pagina o componente che genera la fattura da una proposta.
- FatturaDetails: pagina per visualizzare una fattura emessa.
- DownloadFatturaButton: pulsante per scaricare il PDF.
- ArchivioFattureList: tabella o lista di tutte le fatture.

Componenti per la Dashboard Statistiche

- StatisticsDashboard: dashboard completa delle statistiche.
- KpiCard: card che mostra un singolo KPI (conversion rate, vendite mese…).
- KpiFilterBar: barra dei filtri (periodo, mese, anno).
- GraficoVendite: grafico (linea/barre) delle vendite per periodo.
- GraficoVeicoliPiuVenduti: componente grafico del ranking veicoli.
- StatisticheVenditore: pagina con KPI del singolo venditore.

5. Glossario

Termine	Definizione
Singleton	Modello in cui una classe viene istanziata una sola volta durante il ciclo di vita dell'applicazione. In Spring Boot questo è il comportamento predefinito dei servizi, permettendo di avere componenti condivisi e gestiti dal framework senza creare copie inutili.
Facade	Interfaccia semplificata che raggruppa funzioni complesse nascondendo la struttura interna. Nei sistemi come AutoFlow, i service possono fungere da facciata ai repository e alle logiche applicative, offrendo un unico punto d'accesso alle operazioni sul dominio.
Adapter	Pattern che converte un'interfaccia in un'altra attesa dal chiamante. Utile quando AutoFlow integra servizi esterni (es. sistemi assicurativi o di pagamento) fornendo formati non direttamente compatibili con i DTO interni.
Bridge	Separazione tra astrazione e implementazione, così da poter variare indipendentemente. In AutoFlow può essere usato per gestire differenti strategie di generazione PDF o archiviazione file senza alterare il codice di alto livello.
Builder	Metodo per costruire oggetti complessi passo per passo evitando costruttori enormi. Utile quando si devono generare DTO ricchi (es. dettagli proposta o fattura) partendo da dati parziali provenienti da più sorgenti.
Abstract Factory	Interfaccia che permette di creare gruppi di oggetti correlati senza definirne le classi concrete. Può essere sfruttata in AutoFlow per produrre configurazioni diverse in base al tipo di veicolo o al profilo utente.
Chain of Responsibility	Modello che passa una richiesta lungo una catena di handler fino alla gestione corretta. Applicabile ai filtri delle richieste HTTP (Spring Filter) o alla validazione multilivello dei dati.
DTO (Data Transfer Object)	Oggetti che trasportano dati tra front-end e back-end senza includere logica di business. In AutoFlow sono fondamentali per comunicare dati tra React e i controller Spring tenendo isolato il dominio.
DAO (Data Access Object)	Pattern che incapsula l'accesso ai dati. In AutoFlow è implementato dai repository Spring Data, che gestiscono query e persistenza senza esporre dettagli SQL al livello dei servizi.
CRUD	Acronimo di Create, Read, Update, Delete: rappresenta le operazioni fondamentali per la gestione dei dati in qualunque applicazione software.

HTTP	Protocollo di comunicazione usato per lo scambio dei dati tra client React e server Spring tramite API REST.
JPA	Specifica Java per interagire con database relazionali tramite entità mappate. AutoFlow la utilizza per memorizzare veicoli, proposte, fatture e utenti.
JSDoc	Strumento di documentazione per JavaScript/TypeScript che permette di aggiungere descrizioni e note direttamente nel codice del front-end React.
Javadoc	Sistema di documentazione utilizzato in Java per generare automaticamente reference API dai commenti presenti nel codice dei servizi e controller.
JWT	Standard per codificare informazioni in token firmati utilizzati per autenticazione e autorizzazione. AutoFlow usa JWT per gestire sessioni sicure lato client.
OCL	Linguaggio di vincoli usato in UML per esprimere precondizioni, postcondizioni e invarianti dei metodi, come quelli definiti nelle interfacce di AutoFlow.
REST	Stile architetturale che organizza servizi web tramite risorse raggiungibili con operazioni HTTP standard. AutoFlow espone tutte le funzioni tramite API REST.
UML	Linguaggio di modellazione grafico usato per descrivere struttura e comportamento del software, come nei diagrammi SDD e ODD di AutoFlow.
API	Interfacce che definiscono come diverse parti del software o servizi esterni possono comunicare tra loro. AutoFlow espone API REST consumate dal front-end React.
CSS	Linguaggio di stile per descrivere l'aspetto delle pagine web. In AutoFlow viene utilizzato insieme a React per definire layout e componenti visivi.
Spring Boot	Framework Java che semplifica la creazione di applicazioni back-end grazie a configurazioni automatiche, gestione dei bean e integrazione con Spring Security e JPA.
React	Libreria JavaScript utilizzata per costruire interfacce web dinamiche e component-based, come lo showroom, le dashboard e le pagine di gestione interna di AutoFlow.
TypeScript	Estensione tipizzata di JavaScript che aiuta a ridurre errori nel front-end definendo tipi statici per componenti e strutture dati.

Lombok	Libreria Java che genera automaticamente getter, setter, costruttori e altro boilerplate nelle classi del dominio e dei DTO.
@RestController	Annotazione Spring che definisce controller REST e le relative rotte usate dal front-end React.
@ControllerAdvice	Meccanismo di Spring per gestire globalmente errori ed eccezioni provenienti dai controller.
@ExceptionHandler	Metodo annotato per gestire eccezioni specifiche e restituire risposte strutturate alle API.
Formik	Libreria React utilizzata per gestire form complessi (login, proposte, clienti) mantenendo coerente lo stato dei campi.
Yup	Libreria di validazione dei dati spesso integrata a Formik, utile per form con vincoli strutturati (password, email, dati cliente).
Jakarta EE	Insieme di specifiche per creare applicazioni enterprise Java, oggi alla base di tecnologie utilizzate internamente da molti framework.