Assignment 3 – Applicazione Spring

Repository Link: https://gitlab.com/fam7790642/2023_assignment3_officina.git

Gruppo **FAM**, i cui membri sono:

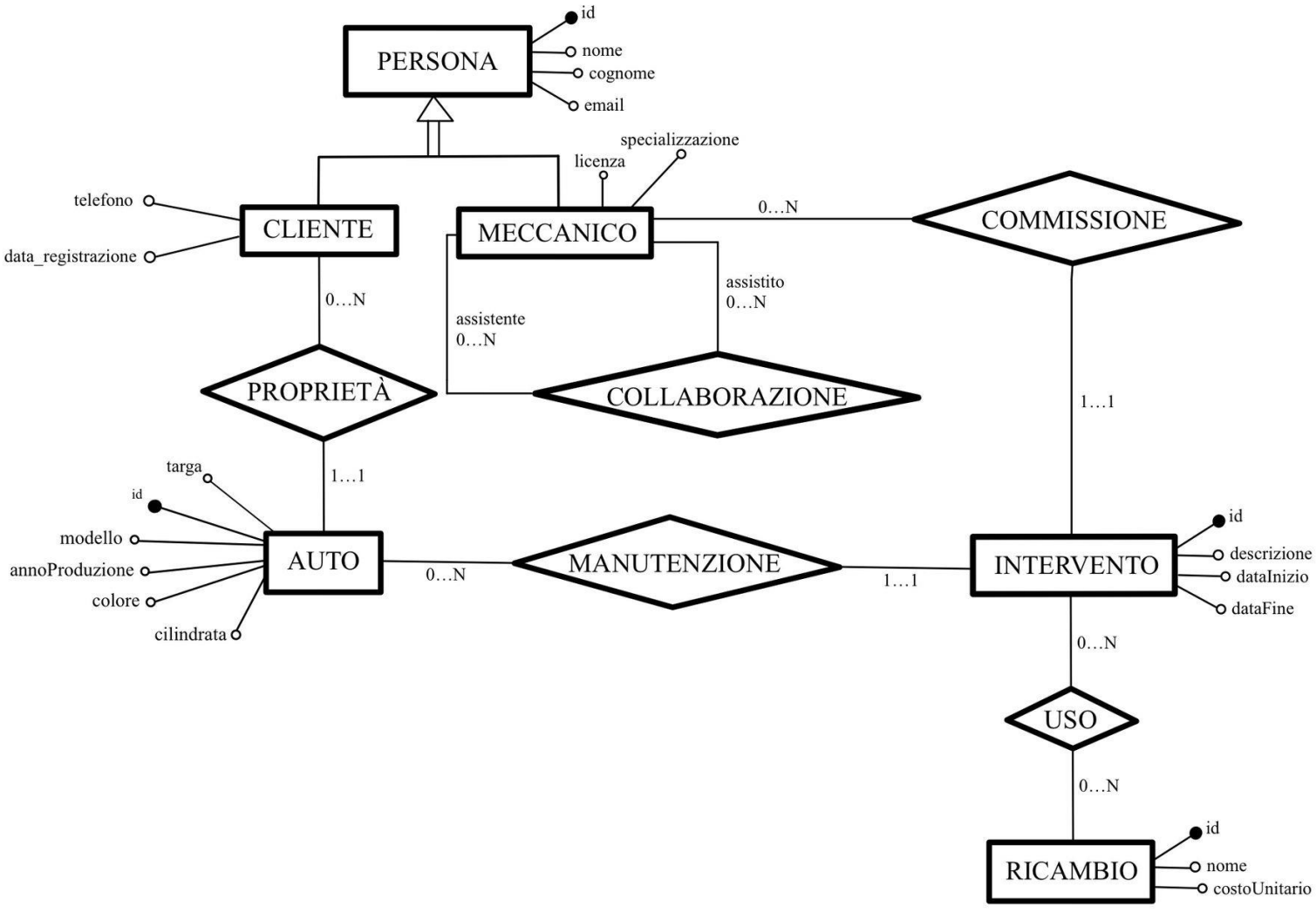
- Biasco Anna Marika 865873
- Pulerà Francesca 870005
- Zarantonello Massimo 866457

Indice

1. Descrizione del Progetto

Per il seguente Assignment si è scelto di implementare un'applicazione Spring che rappresenta un'officina virtuale, con la capacità di registrare nuovi veicoli e di tracciare le diverse attività di riparazione.

Di seguito viene mostrato il diagramma entità-relazione della nostra Web Application:



1.1. Entità

Le entità dello schema E/R sono:

- Persona -> rappresenta un individuo che può interagire con l'officina, assumendo uno dei due ruoli principali: *Cliente* o *Meccanico*. La chiave primaria è l'identificatore (*id*), e ha gli attributi comuni di *nome*, *cognome* ed *email*

    - Cliente -> estensione di *Persona*, con l'aggiunta di ulteriori dettagli specifci per i clienti, come il *telefono* e la *data_registrazione*
    - Meccanico -> estensione di *Persona*, con l'aggiunta degli attributi specifici per i meccanici, come *licenza* e *specializzazione*
- Auto -> rappresenta un'automobile. È caratterizzata dagli attributi *modello, annoProduzione, colore, cilindrata, targa* e dalla chiave primaria *id.*
- Intervento -> rappresenta un'operazione di riparazione eseguita su un'automobile. Gode degli attributi *descrizione*, *dataInizio* e *dataFine* e della chiave primaria *id.*
- Ricambio -> rappresenta un componente di ricambio che può essere utilizzato durante un intervento di riparazione. È caratterizzato dagli attributi *nome*, *costoUnitario* e dalla chiave primaria *id*.

1.2. Relazioni

Le relazioni dello schema E/R sono definite come segue:

- Proprietà -> Rappresenta la relazione tra un cliente e la sua automobile. Si tratta di una relazione zero a molti, poiché un cliente registrato può possedere più automobili (ma anche zero, nel momento in cui ad esempio si è registrato il cliente ma non ancora la relativa automobile). Allo stesso tempo, è una relazione uno a uno, in quanto un'automobile può essere di proprietà di al più una persona.
- Collaborazione -> Si tratta di una relazione 0 a molti con se stesso rispetto alla figura del *Meccanico*. Un meccanico può assistere nessuno o molti meccanici (nell'ambito del ruolo di assistente), oppure essere assistito da nessuno o molti meccanici (nel ruolo di assistito).
- Manutenzione -> Rappresenta una relazione tra le entità di *Auto* e *Intervento*. Un'auto può essere oggetto di zero a molti interventi (zero quando si registra l'automobile ma non si è ancora registrato l'intervento che ha subito), mentre un intervento è univoco per la macchina su cui viene eseguito.
- Uso -> Indica una relazione molti a molti tra *Intervento* e *Ricambio*. Un intervento può richiedere diversi pezzi di ricambio, e un pezzo di ricambio può essere utilizzato in diversi interventi, in quanto si intende la "categoria" del pezzo di ricambio (ad esempio "ruota", "marmitta", …)
- Commissione -> Un *Meccanico* può essere incaricato di diversi interventi, mentre un *Intervento* è affidato ad un singolo meccanico.

2. Classi e MVC

Le classi della nostra applicazione web sono organizzate in diverse categorie:

- Controllers: Gestiscono le richieste HTTP e coordinano la logica di controllo.
- Models: Rappresentano le entità di dominio e contengono la logica di business.
- Repository: Gestiscono l'accesso e la persistenza dei dati nel database.
- Services: Implementano la logica di business e coordinano le interazioni tra controller e repository.

2.1. Controllers

In questa categoria abbiamo diverse classi, ognuna delle quali implementa le operazioni CRUD e di ricerca:

- AutoController, la cui responsabilità principale è gestire le richieste relative alle operazioni sulle automobili all'interno dell'applicazione web:
    - Costruttore:
      La classe ha un costruttore che accetta due servizi (*AutoService* e *ClienteService*) come argomenti e li assegna alle relative variabili di istanza.
    - Elenco delle Automobili:
        - Metodo *listAutoimobili*: Gestisce la richiesta GET a "/automobili" e restituisce una lista di automobili da visualizzare nella vista "automobili-list". La lista include tutte le automobili o solo quelle che soddisfano i filtri di ricerca.
        - Metodo *listAutoimobili* (POST): Gestisce la richiesta POST a "/automobili" con parametri opzionali per filtrare le automobili. Restituisce la vista "automobili-list" con le automobili risultanti dopo l'applicazione dei filtri.
    - Creazione di una nuova Automobile:
        - Metodo *createAutoForm*: Gestisce la richiesta GET a "/automobili/{clienteId}/new" e restituisce la vista "automobili-create" per creare una nuova automobile associata a un cliente specifico.
        - Metodo *saveAuto* (POST): Gestisce la richiesta POST a "/automobili/{clienteId}/new" per salvare una nuova automobile associata a un cliente. Se la validazione ha successo, la nuova auto viene salvata e l'utente viene reindirizzato alla lista delle automobili.
    - Modifica di un'Automobile esistente:
        - Metodo *editAutoForm*: Gestisce la richiesta PUT a "/automobili/{autoid}" e restituisce la vista "automobili-edit" per modificare un'automobile esistente.
        - Metodo *updateAuto* (POST): Gestisce la richiesta POST a "/automobili/{autoid}" per aggiornare i dettagli di un'automobile esistente.

Se la validazione ha successo, le modifiche vengono salvate e l'utente viene reindirizzato ai dettagli dell'auto.

- Eliminazione di un'Automobile:
  - Metodo *deleteAuto* (DELETE): Gestisce la richiesta DELETE a "/automobili/{autoid}" per eliminare un'automobile esistente. Dopo l'eliminazione, l'utente viene reindirizzato alla lista delle automobili.
- Dettagli dell'Automobile:
  - Metodo *detailAuto*: Gestisce la richiesta GET a "/automobili/{autoid}" e restituisce la vista "automobili-detail" per visualizzare i dettagli di un'automobile specifica. In caso di errore, l'utente viene reindirizzato a una pagina di errore generica.

- ClienteController, che gestisce le operazioni relative ai clienti:
  - Costruttore:
    La classe ha un costruttore che accetta un servizio (*ClienteService*) come argomento e lo assegna alla variabile di istanza *clienteService*.
  - Elenco dei Clienti:
    - Metodo *listClienti*: Gestisce la richiesta GET a "/clienti" e restituisce una lista di clienti da visualizzare nella vista "clienti-list". La lista include tutti i clienti o solo quelli che soddisfano i filtri di ricerca.
    - Metodo *searchClienti* (POST): Gestisce la richiesta POST a "/clienti" con parametri opzionali per filtrare i clienti. Restituisce la vista "clienti-list" con i clienti risultanti dopo l'applicazione dei filtri.
  - Creazione di un nuovo Cliente:
    - Metodo *createClienteForm*: Gestisce la richiesta GET a "/clienti/new" e restituisce la vista "clienti-create" per creare un nuovo cliente.
    - Metodo *saveCliente* (POST): Gestisce la richiesta POST a "/clienti/new" per salvare un nuovo cliente. Se la validazione ha successo, il nuovo cliente viene salvato e l'utente viene reindirizzato alla lista dei clienti.
  - Modifica di un Cliente esistente:
    - Metodo *editClienteForm*: Gestisce la richiesta GET a "/clienti/{clienteId}/edit" e restituisce la vista "clienti-edit" per modificare un cliente esistente.
    - Metodo *updateCliente* (POST): Gestisce la richiesta POST a "/clienti/{clienteId}/edit" per aggiornare i dettagli di un cliente esistente. Se la validazione ha successo, le modifiche vengono salvate e l'utente viene reindirizzato alla lista dei clienti.
  - Visualizzazione dettagli Cliente:
    - Metodo *viewDetailCliente*: Gestisce la richiesta GET a "/clienti/{clienteId}" e restituisce la vista "clienti-detail" per visualizzare i dettagli di un cliente specifico. In caso di errore, l'utente viene reindirizzato a una pagina di errore generica. La visualizzazione del dettaglio Cliente non è accessibile dall'applicazione.
  - Eliminazione di un Cliente:

- Metodo *deleteCliente* (DELETE): Gestisce la richiesta DELETE a "/clienti/{clienteId}" per eliminare un cliente esistente. Dopo l'eliminazione, l'utente viene reindirizzato alla lista dei clienti.

- InterventoController:
  - Costruttore:
    La classe ha un costruttore che accetta quattro servizi (*InterventoService*, *RicambioService*, *AutoService*, e *MeccanicoService*) come argomenti e li assegna alle relative variabili di istanza.
  - Elenco degli Interventi:
    - Metodo *interventoList*: Gestisce la richiesta GET a "/interventi" e restituisce una lista di interventi da visualizzare nella vista "interventi-list". La lista include tutti gli interventi o solo quelli che soddisfano i filtri di ricerca.
    - Metodo *searchInterventi* (POST): Gestisce la richiesta POST a "/interventi" con parametri opzionali per filtrare gli interventi. Restituisce la vista "interventi-list" con gli interventi risultanti dopo l'applicazione dei filtri.
  - Creazione di un nuovo intervento:
    - Metodo *createInterventoFormAuto*: Gestisce la richiesta GET a "/interventi/{autoId}/new" e restituisce la vista "interventi-create" per creare un nuovo intervento associato a un'auto specifica.
    - Metodo *saveIntervento* (POST): Gestisce la richiesta POST a "/interventi/{autoId}" per salvare un nuovo intervento associato a un'auto. Se la validazione ha successo, il nuovo intervento viene salvato e l'utente viene reindirizzato alla lista degli interventi.
  - Modifica di un Intervento esistente:
    - Metodo *editInterventoForm*: Gestisce la richiesta GET a "/interventi/{interventoId}/edit" e restituisce la vista "interventi-edit" per modificare un intervento esistente.
    - Metodo *updateIntervento* (POST): Gestisce la richiesta POST a "/interventi/{interventoId}/edit" per aggiornare i dettagli di un intervento esistente. Se la validazione ha successo, le modifiche vengono salvate e l'utente viene reindirizzato alla lista degli interventi.
  - Visualizzazione dettagli Intervento:
    - Metodo viewDetailIntervento: Gestisce la richiesta GET a "/interventi/{interventoId}" e restituisce la vista "interventi-detail" per visualizzare i dettagli di un intervento specifico. In caso di errore, l'utente viene reindirizzato a una pagina di errore generica.
  - Eliminazione di un Intervento:
    - Metodo *deleteIntervento* (DELETE): Gestisce la richiesta DELETE a "/interventi/{interventoId}" per eliminare un intervento esistente. Dopo l'eliminazione, l'utente viene reindirizzato alla lista degli interventi.
  - Filtro degli Interventi:
    - Metodo filtroInterventi: Gestisce la richiesta GET a "/interventi/filtro" e restituisce la vista "interventi-filter" per visualizzare un filtro di ricerca degli interventi.

- MeccanicoController:
  - Costruttore:
    - La classe ha un costruttore che accetta un servizio (*MeccanicoService*) come argomento e lo assegna alla variabile di istanza *meccanicoService*.
  - Elenco dei Meccanici:
    - Metodo *listMeccanici*: Gestisce la richiesta GET a "/meccanici" e restituisce una lista di meccanici da visualizzare nella vista "meccanici-list". La lista include tutti i meccanici o solo quelli che soddisfano i filtri di ricerca.
    - Metodo *searchMeccanici* (POST): Gestisce la richiesta POST a "/meccanici" con parametri opzionali per filtrare i meccanici. Restituisce la vista "meccanici-list" con i meccanici risultanti dopo l'applicazione dei filtri.
  - Creazione di un nuovo Meccanico:
    - Metodo *createMeccanicoForm*: Gestisce la richiesta GET a "/meccanici/new" e restituisce la vista "meccanici-create" per creare un nuovo meccanico.
    - Metodo *saveMeccanico* (POST): Gestisce la richiesta POST a "/meccanici/new" per salvare un nuovo meccanico. Se la validazione ha successo, il nuovo meccanico viene salvato e l'utente viene reindirizzato alla lista dei meccanici.
  - Modifica di un Meccanico esistente:
    - Metodo *editMeccanicoForm*: Gestisce la richiesta GET a "/meccanici/{meccanicoId}/edit" e restituisce la vista "meccanici-edit" per modificare un meccanico esistente.
    - Metodo *updateMeccanico* (POST): Gestisce la richiesta POST a "/meccanici/{meccanicoId}/edit" per aggiornare i dettagli di un meccanico esistente. Se la validazione ha successo, le modifiche vengono salvate e l'utente viene reindirizzato alla lista dei meccanici.
  - Eliminazione di un Meccanico:
    - Metodo *deleteMeccanico* (DELETE): Gestisce la richiesta DELETE a "/meccanici/{meccanicoId}" per eliminare un meccanico esistente. Dopo l'eliminazione, l'utente viene reindirizzato alla lista dei meccanici.
  - Aggiunta di un Assistente a un Meccanico:
    - Metodo *addAssistente* (POST): Gestisce la richiesta POST a "/meccanici/{meccanicoId}/assistenti" per aggiungere un assistente a un meccanico esistente. Le modifiche vengono salvate e l'utente viene reindirizzato alla lista dei meccanici.
  - Visualizzazione dettagli Meccanico:
    - Metodo *viewDetailMeccanico*: Gestisce la richiesta GET a "/meccanici/{meccanicoId}" e restituisce la vista "meccanici-detail" per visualizzare i dettagli di un meccanico specifico. In caso di errore, l'utente viene reindirizzato a una pagina di errore generica.
- RicambioController:
  - Elenco Ricambi:

    - Metodo *listRicambi*: Gestisce la richiesta GET a "/ricambi" e restituisce la vista "./ricambi/ricambi-list", mostrando l'elenco di tutti i ricambi.
  - Ricerca Ricambi:
    - Metodo *searchRicambi*: Gestisce la richiesta POST a "/ricambi" per la ricerca dei ricambi. Filtra i ricambi in base al nome e mostra i risultati nella vista "./ricambi/ricambi-list".
  - Creazione di un nuovo Ricambio:
    - Metodo *createRicambioForm*: Gestisce la richiesta GET a "/ricambi/new" e restituisce la vista "./ricambi/ricambi-create" per la creazione di un nuovo ricambio.
    - Metodo *saveRicambio*: Gestisce la richiesta POST a "/ricambi/new" per salvare il nuovo ricambio. Se la validazione ha successo, salva il ricambio e reindirizza a "/ricambi".
  - Modifica di un Ricambio esistente:
    - Metodo *editRicambioForm*: Gestisce la richiesta GET a "/ricambi/{ricambioId}/edit" e restituisce la vista "./ricambi/ricambi-edit" per modificare un ricambio esistente.
    - Metodo *updateRicambio*: Gestisce la richiesta POST a "/ricambi/{ricambioId}/edit" per aggiornare un ricambio esistente. Se la validazione ha successo, salva il ricambio e reindirizza a "/ricambi".
  - Dettagli di un Ricambio:
    - Metodo *viewDetailCliente*: Gestisce la richiesta GET a "/ricambi/{ricambioId}" e restituisce la vista "./ricambi/ricambi-detail" per visualizzare i dettagli di un ricambio. La pagina di dettaglio del Ricambio non è accessibile dall'applicazione.
  - Eliminazione di un Ricambio:
    - Metodo *deleteRicambio*: Gestisce la richiesta DELETE a "/ricambi/{ricambioId}" per eliminare un ricambio. Prima di eliminare il ricambio, elimina anche gli interventi associati ad esso. Infine, reindirizza a "/ricambi".

Infine c'è OfficinaController, la quale gestisce le richieste relative alle pagine statiche del sito web, come la pagina principale dell'Officina virtuale, quella relativa ai contatti o alla privacy.

I vari metodi di cui è composta, dunque, non gestiscono operazioni CRUD, ma sono utili per fornire informazioni statiche e a gestire gli errori.

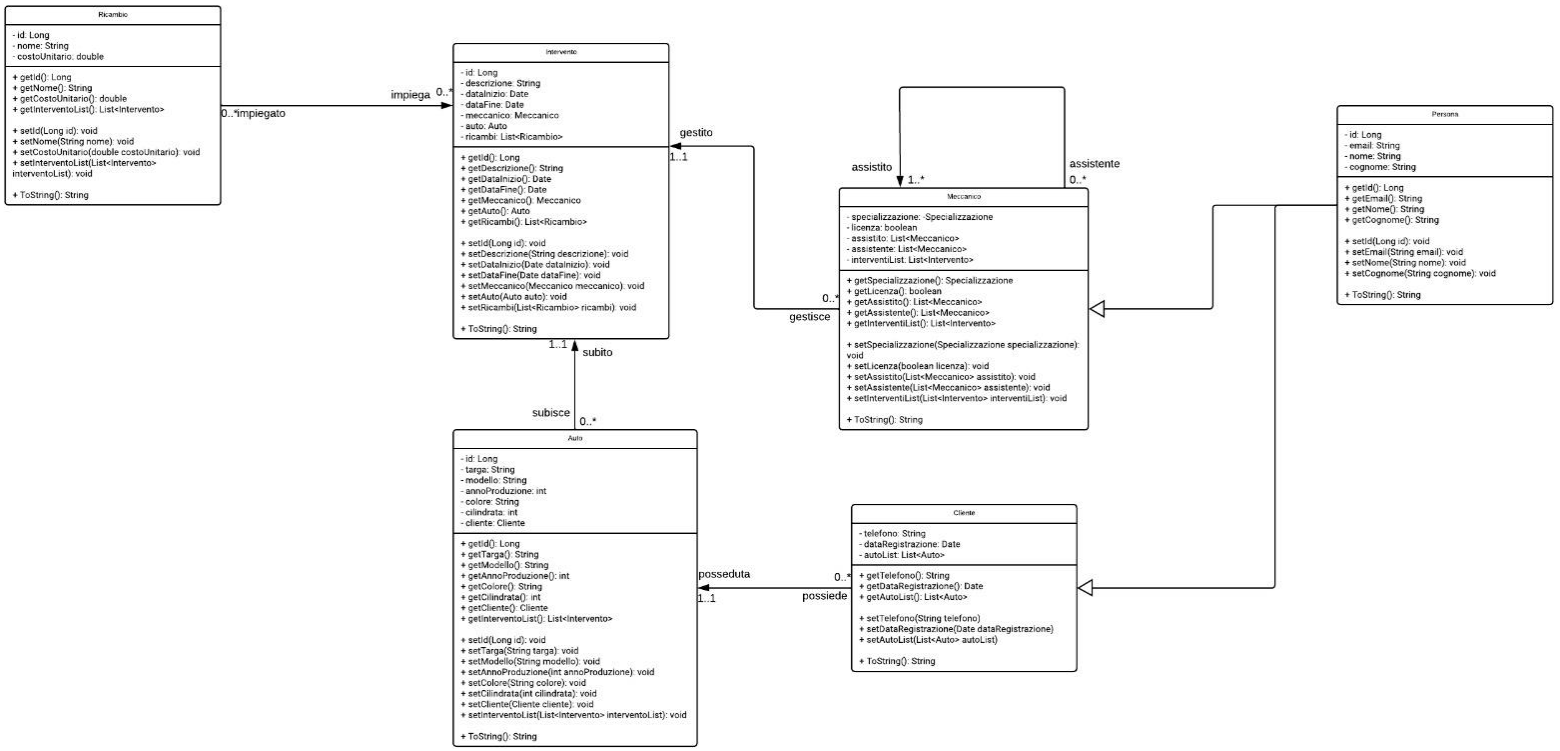
2.2. Models

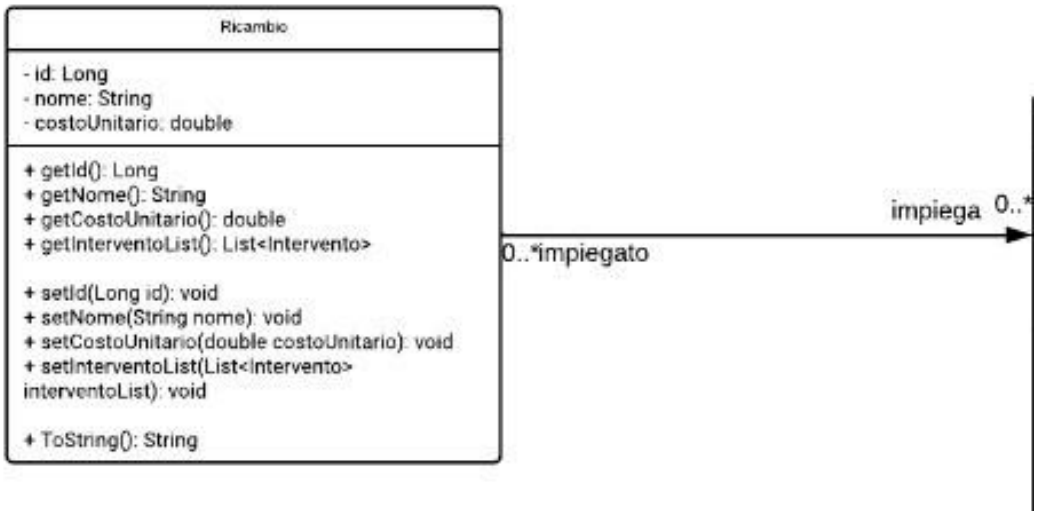
Sono tutte le classi che rappresentano gli oggetti di dominio o le entità all'interno del sistema, e dunque:

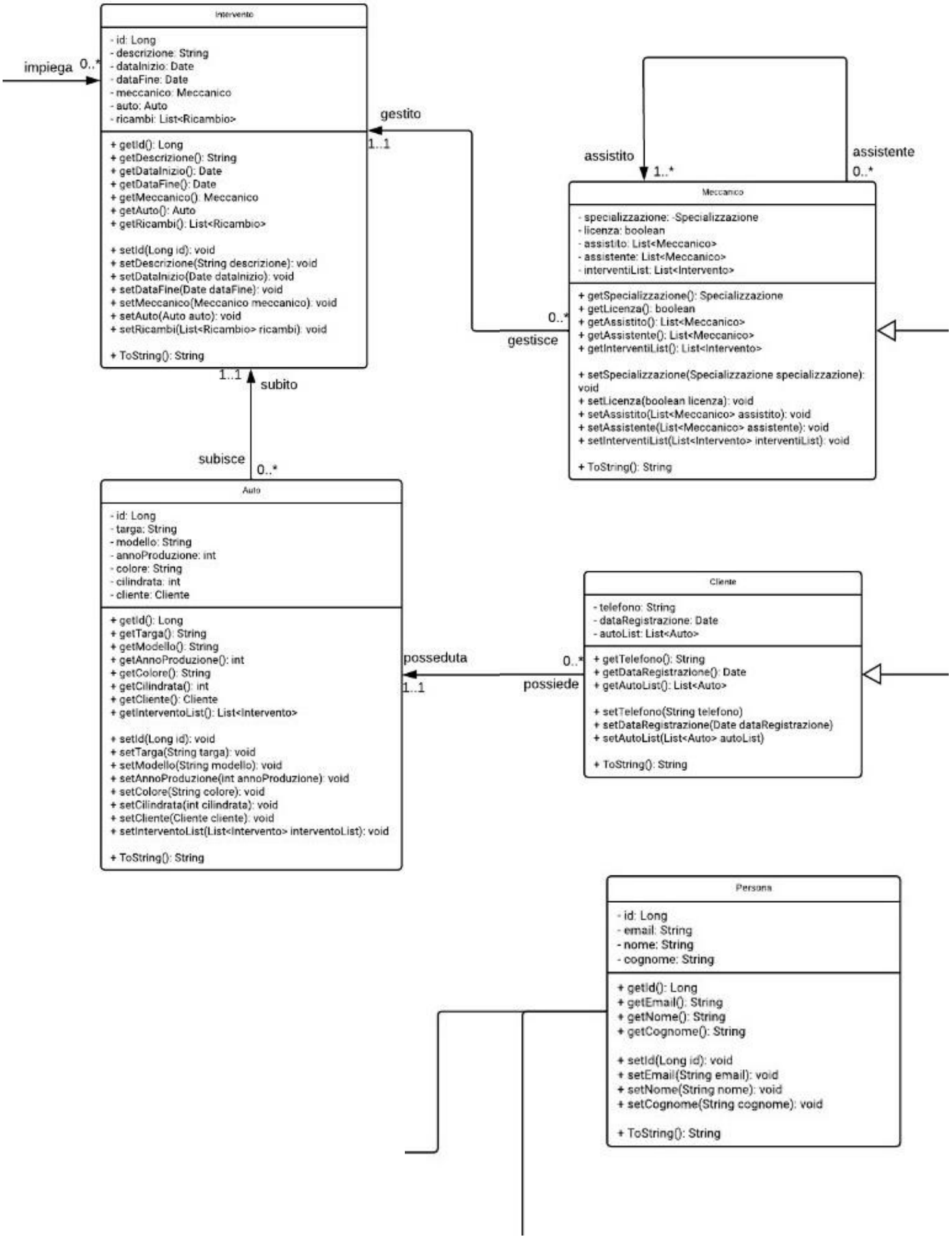
- Auto
- Cliente

- Intervento
- Meccanico
- Persona
- Ricambio

Di seguito viene mostrato il diagramma delle classi UML:






Intervento
- id: Long
- descrizione: String
- dataInizio: Date
- dataFine: Date
- meccanico: Meccanico
- auto: Auto
- ricambi: List<Ricambio>
+ getId(): Long
+ getDescrizione(): String
+ getDataInizio(): Date
+ getDataFine(): Date
+ getMeccanico(): Meccanico
+ getAuto(): Auto
+ getRicambi(): List<Ricambio>
+ setId(Long id): void
+ setDescrizione(String descrizione): void
+ setDataInizio(Date dataInizio): void
+ setDataFine(Date dataFine): void
+ setMeccanico(Meccanico meccanico): void
+ setAuto(Auto auto): void
+ setRicambi(List<Ricambio> ricambi): void
+ ToString(): String
impiega
0..*
gestito
1..1
assistito
1..*
assistente
0..*
Meccanico
- specializzazione: -Specializzazione
- licenza: boolean
- assistito: List<Meccanico>
- assistente: List<Meccanico>
- interventiList: List<Intervento>
+ getSpecializzazione(): Specializzazione
+ getLicenza(): boolean
+ getAssistito(): List<Meccanico>
+ getAssistente(): List<Meccanico>
+ getInterventiList(): List<Intervento>
+ setSpecializzazione(Specializzazione specializzazione): void
+ setLicenza(boolean licenza): void
+ setAssistito(List<Meccanico> assistito): void
+ setAssistente(List<Meccanico> assistente): void
+ setInterventiList(List<Intervento> interventiList): void
+ ToString(): String
0..*
gestisce
1..1
subito
subisce
0..*
Auto
- id: Long
- targa: String
- modello: String
- annoProduzione: int
- colore: String
- cilindrata: int
- cliente: Cliente
+ getId(): Long
+ getTarga(): String
+ getModello(): String
+ getAnnoProduzione(): int
+ getColore(): String
+ getCilindrata(): int
+ getCliente(): Cliente
+ getInterventoList(): List<Intervento>
+ setId(Long id): void
+ setTarga(String targa): void
+ setModello(String modello): void
+ setAnnoProduzione(int annoProduzione): void
+ setColore(String colore): void
+ setCilindrata(int cilindrata): void
+ setCliente(Cliente cliente): void
+ setInterventoList(List<Intervento> interventoList): void
+ ToString(): String
posseduta
1..1
0..*
possiede
Cliente
- telefono: String
- dataRegistrazione: Date
- autoList: List<Auto>
+ getTelefono(): String
+ getDataRegistrazione(): Date
+ getAutoList(): List<Auto>
+ setTelefono(String telefono)
+ setDataRegistrazione(Date dataRegistrazione)
+ setAutoList(List<Auto> autoList)
+ ToString(): String
Persona
- id: Long
- email: String
- nome: String
- cognome: String
+ getId(): Long
+ getEmail(): String
+ getNome(): String
+ getCognome(): String
+ setId(Long id): void
+ setEmail(String email): void
+ setNome(String nome): void
+ setCognome(String cognome): void
+ ToString(): String

2.3. Repository

Ne fanno parte AutoRepository, InterventoRepository, ClienteRepository, RicambioRepository e MeccanicoRepository, che sono le classi responsabili per l'accesso e la manipolazione dei dati nel database. Esse forniscono un'interfaccia dichiarativa per l'accesso ai dati dei clienti, semplificando l'interazione con il database all'interno dell'applicazione Spring.

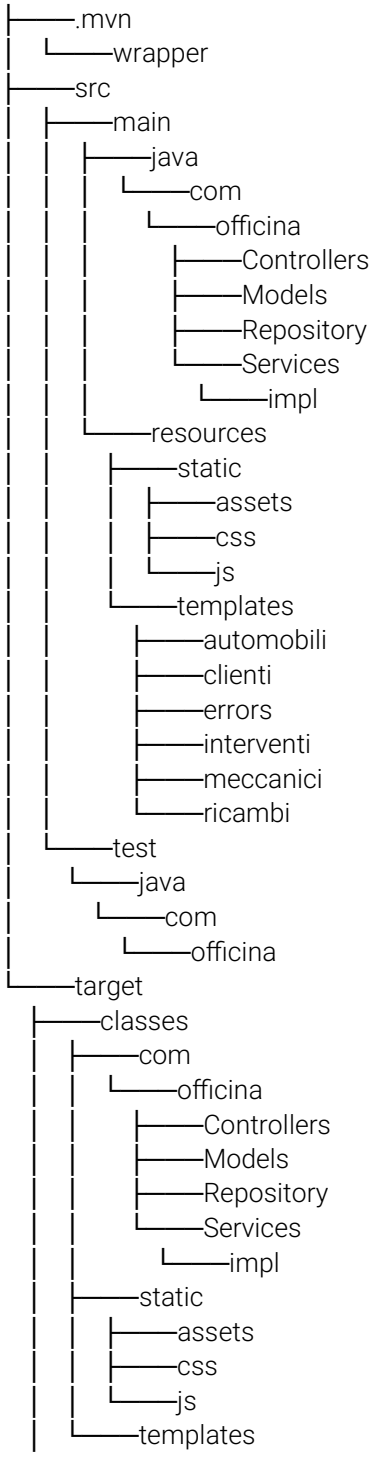
2.4. Services

Ne fanno parte AutoService, ClienteService, InterventoService, RicambioService e MeccanicoService. L'utilizzo di questa interfaccia consente di separare la logica di business dalla logica di accesso ai dati, fornendo un'astrazione che facilita la manutenzione e l'estensione del codice.

3. Architettura Applicazione

3.1. Struttura del codice

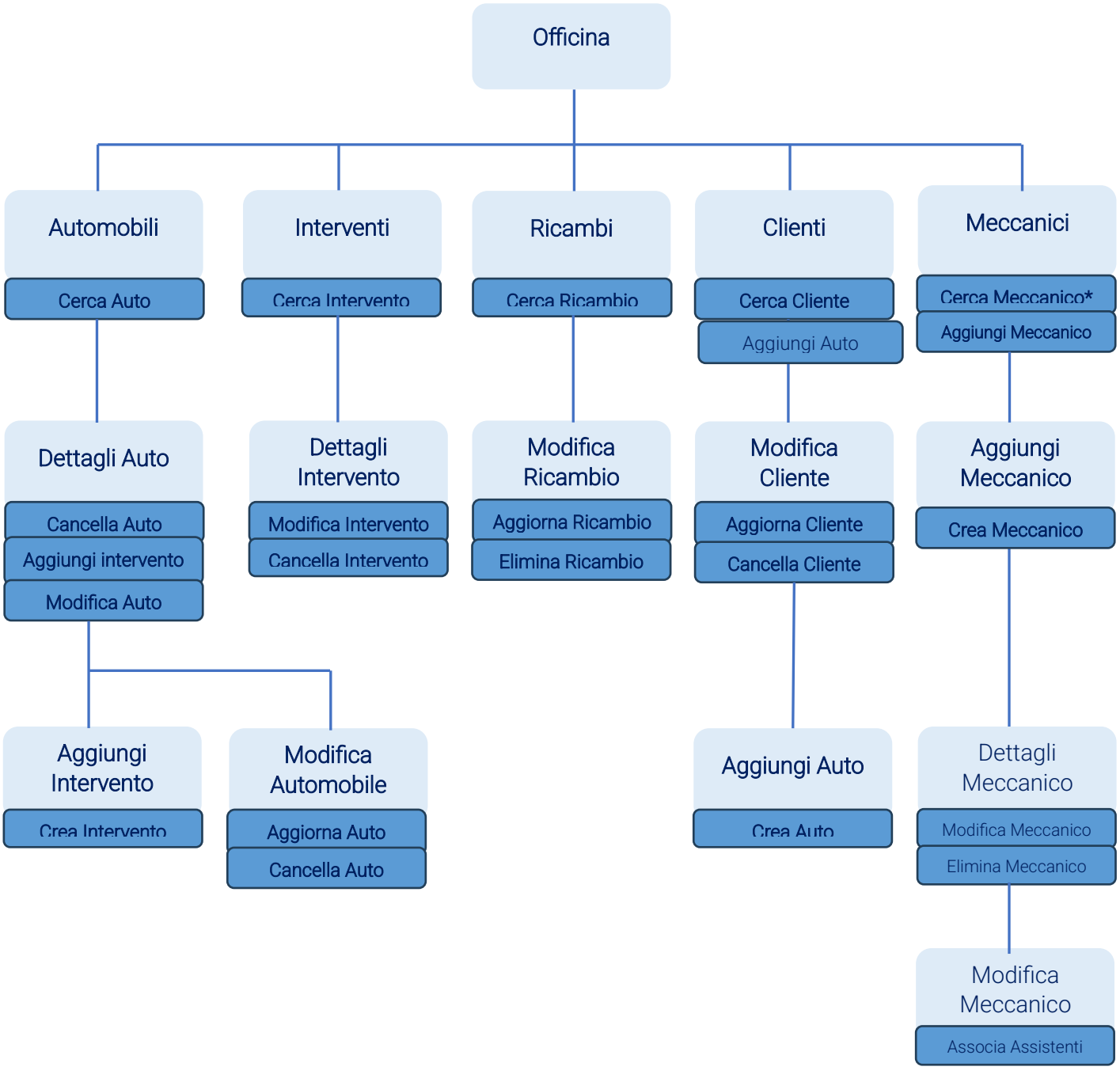
Di seguito la struttura del progetto:

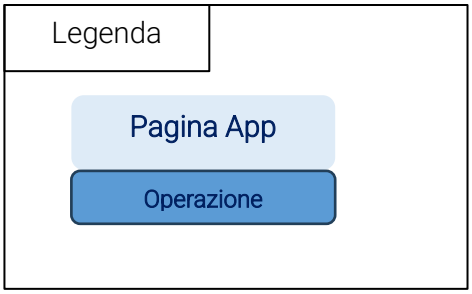


```
│       ├──automobili
│       ├──clienti
│       ├──errors
│       ├──interventi
│       ├──meccanici
│       └──ricambi
├──generated-sources
│   └──annotations
├──generated-test-sources
│   └──test-annotations
└──test-classes
    └──com
        └──officina
```

3.2. Navigazione all'interno della Web App

Il diagramma seguente rappresenta la navigabilità implementata nell'applicazione Officina, evidenziando le operazioni possibili su ciascuna pagina:





*La funzionalità di ricerca per i meccanici è stata implementata in modo che, inserita la specializzazione desiderata, restituisca l'insieme di meccanici che possiedono tale specializzazione. Questo differisce dalle altre entità, in cui la ricerca considera tutti gli attributi.

Considerazioni importanti:

- L'aggiunta di un intervento può avvenire solo a partire da una specifica auto, poiché è ragionevole supporre che un intervento sia associato esclusivamente ad un'auto.
- L'aggiunta di un'auto è possibile solo durante l'inserimento di un nuovo cliente.
- Per creare un nuovo intervento è necessario che almeno un meccanico sia presente nel database.
- La cancellazione di un'auto comporta l'eliminazione di tutti gli interventi ad essa associati.
- La cancellazione di un cliente comporta l'eliminazione di tutte le auto di cui era proprietario.
- La cancellazione di un meccanico comporta l'eliminazione di tutti i suoi interventi associati.
- Tutte le funzioni di ricerca degli elementi sono progettate per restituire una pagina di errore nel caso in cui l'elemento cercato non sia presente nel database.

3.3. Gestione delle dipendenze

Le dipendenze all'interno di questa applicazione non sono state gestite esplicitamente mediante l'annotazione @Autowired. Poiché non sono presenti costruttori personalizzati oltre a quelli di default nelle classi coinvolte, Spring Boot assume automaticamente il controllo della gestione delle dipendenze. Il framework si basa su meccanismi di injection automatica per risolvere e iniettare le dipendenze necessarie attraverso annotazioni come @Repository, @Service e @Controller.

3.4. Transazioni

Metodi Transazionali:

- createIntervento:
  - Il metodo createIntervento è annotato con @Transactional, indicando che una transazione deve essere avviata prima dell'esecuzione del metodo e completata al termine.
  - Questo assicura l'integrità dei dati durante l'inserimento di un nuovo intervento nel sistema.
- findAutoById:
  - Il metodo findAutoById è contrassegnato come un metodo che può essere eseguito durante una transazione. Questo significa che il recupero di informazioni su un'auto specifica avverrà all'interno di una transazione gestita da Spring Boot.

Metodi Non Transazionali:

- updateAuto e deleteAuto:
    - I metodi updateAuto e deleteAuto non possono essere eseguiti all'interno di una transazione.
    - Per garantire che queste operazioni siano eseguite al di fuori di qualsiasi contesto transazionale, l'annotazione @Transactional(propagation = Propagation.NEVER) è stata utilizzata. Ciò significa che se c'è una transazione in corso quando uno di questi metodi viene chiamato, verrà sollevata un'eccezione.

Queste decisioni di progettazione mirano a garantire la coerenza dei dati e a definire chiaramente i contesti in cui le transazioni sono coinvolte o escluse.

Un elevato livello di isolamento, se applicato in modo indiscriminato, potrebbe generare basse prestazioni, poiché due transazioni che accedono simultaneamente agli stessi dati non possono essere eseguite contemporaneamente. In risposta a questo dilemma, è stata presa la decisione strategica di limitare l'utilizzo delle transazioni esclusivamente ai metodi associati alla creazione di nuovi interventi. Questa scelta riflette la natura di una piccola applicazione di software gestionale, dove la creazione di interventi rappresenta un'operazione critica. Limitando le transazioni a questo contesto specifico, si mira a garantire una maggiore efficienza nel contesto di un'applicazione gestionale di dimensioni ridotte.

3.5. REST

Nell'applicazione sono state rispettate le convenzioni di servizio REST, seguendo le seguenti pratiche:

- Per una richiesta in GET a localhost/automobili, verrà restituito l'elenco delle automobili.
- Per una richiesta in POST a localhost/automobili, verrà aggiunto un'automobile all'elenco.
- Per una richiesta in GET a localhost/automobili/{id}, verrà restituita l'automobile con l'ID specificato.
- Per una richiesta in PUT a localhost/automobili/{id}, verrà aggiornata l'automobile con l'ID specificato.
- Per una richiesta in DELETE a localhost/automobili/{id}, verrà eliminata l'automobile con l'ID specificato.

Queste stesse pratiche sono state implementate anche per le entità interventi, ricambi, meccanici, e clienti. In questo modo, l'intera applicazione segue le convenzioni standard dei servizi REST per la gestione delle risorse, rendendo l'interfaccia coerente e intuitiva per gli sviluppatori che interagiscono con il servizio.

4. Setup & Run

Si rimanda al Readme del progetto.