

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA IN INFORMATICA



Migrazione e analisi comparativa di un
back-end per un servizio di smart parking

Tesi di laurea

Relatore

Prof. Paolo Baldan

Laureando

Andrea Volpe

ANNO ACCADEMICO 2021-2022

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Pinco Pallino presso l'azienda Azienda S.p.A. Gli obbiettivi da raggiungere erano molteplici.

In primo luogo era richiesto lo sviluppo di ... In secondo luogo era richiesta l'implementazione di un ... Tale framework permette di registrare gli eventi di un controllore programmabile, quali segnali applicati Terzo ed ultimo obbiettivo era l'integrazione ...

“Life is really simple, but we insist on making it complicated”

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. NomeDelProfessore, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Dicembre 2022

Andrea Volpe

Indice

1	Introduzione	1
1.1	L'azienda	1
1.2	Scelta dell'azienda	1
1.3	Introduzione al progetto	2
1.4	Problematiche riscontrate	3
1.5	Soluzione scelta	3
1.6	Descrizione del prodotto ottenuto	4
1.7	Tecnologie utilizzate	5
1.8	Organizzazione del testo	6
2	Analisi dei requisiti	7
2.1	Confronto con gli stakeholders	7
2.1.1	Servizio REST API	7
2.1.2	Servizio di polling	7
2.2	Entità	9
2.3	Casi d'uso	11
2.4	Tracciamento dei requisiti	17
3	Progettazione	21
3.1	Architettura del progetto	21
3.1.1	Layered architecture	21
3.1.2	Motivazioni della scelta	22
3.2	Struttura software	23
3.2.1	IoC container	23
3.2.2	Controller e provider	24
3.2.3	Repository	27
3.2.4	Moduli	28
3.2.5	DTO	30
3.2.6	Eccezioni	31
3.2.7	Logging	32
4	Verifica e validazione	33
4.1	Verifica	33
4.1.1	Criteri di verifica	33
4.1.2	Strumenti utilizzati	33
4.1.3	Progettazione	34
4.1.4	Realizzazione	35
4.1.5	Scheduler	35

5 Conclusioni	37
A Appendice A	39
Bibliografia	43

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 L'azienda

Sync Lab nasce a Napoli nel 2002 come software house ed è rapidamente cresciuta nel mercato dell'Information and Communications Technology (ICT) G .

A seguito di una maturazione delle competenze tecnologiche, metodologiche ed applicative nel dominio del software, l'azienda è riuscita rapidamente a trasformarsi in System Integrator conquistando significative fette di mercato nei settori mobile, videosorveglianza e sicurezza delle infrastrutture informatiche aziendali.

Attualmente, Sync Lab ha più di 150 clienti diretti e finali, con un organico aziendale di 300 dipendenti distribuiti tra le 6 sedi dislocate in tutta Italia. Sync Lab si pone come obiettivo principale quello di supportare il cliente nella realizzazione, messa in opera e governance di soluzione IT, sia dal punto di vista tecnologico, sia nel governo del cambiamento organizzativo.



1.2 Scelta dell'azienda

Sono venuto a conoscenza dell'azienda Sync Lab grazie al progetto d'ingegneria del software, dove l'azienda è stata la proponente del mio progetto.

Sono venuto a conoscenza del progetto di stage di Sync Lab grazie all'evento stage-it 2022. L'evento promosso da Assindustria Venetocentro in collaborazione con l'Università di Padova per favorire l'incontro tra aziende con progetti innovativi in ambito IT e studenti dei corsi di laurea in Informatica, Ingegneria informatica e Statistica.

1.3 Introduzione al progetto




Lo scopo del progetto di stage consiste nell'effettuare la migrazione di un servizio di API REST lato back-end realizzato da un precedente studente tirocinante con il framework Spring in un servizio di API REST lato back-end realizzato con un diverso framework, chiamato NestJS. La migrazione viene fatta per effettuare un'analisi comparativa tra i due servizi, in modo da valutarne le caratteristiche e decidere quale dei due meglio si adatta alle esigenze del progetto.

Il progetto consiste nella realizzazione di una webapp che si occupa di gestire un sistema di controllo parcheggi auto. Il sistema va ad interrogare una base di dati contenente l'informazione inerente allo stato di alcuni sensori di parcheggio fornendo la visualizzazione dei posti liberi/occupati all'interno di una mappa.

L'idea del progetto consiste nel agevolare il client dell'applicativo, in quanto può venire a conoscenza della disponibilità di un parcheggio prima di entrarci, evitando quindi spostamenti inutili nel caso il parcheggio sia pieno.

E' prevista poi la realizzazione di una sezione dedicata ai manutentori, per verificare lo stato dei sensori, facilitando quindi il processo di manutenzione.

Il progetto è formato da una parte di front-end, realizzata con il framework Angular e una parte di back-end che consiste di un servizio di REST API, realizzato in due versioni: una con il framework Spring e una con il framework NestJS.

	Parcheggio libero
	Parcheggio occupato
	Sensore ambientale



1.4 Problematiche riscontrate

Problematiche dovute alla mancanza di conoscenza delle tecnologie:

- * Architettura a microservizi: avevo solo una conoscenza basilare della tecnologia, grazie al corso d'ingegneria del software ma non sufficiente per sviluppare il progetto.
- * Framework Spring: la conoscenza di questo framework era completamente assente ed era importante conoscerlo per poter comprendere con chiarezza il software esistente di cui doveva essere effettuata la migrazione.
- * Framework Node.js e NestJS: la conoscenza di questi due framework era completamente assente era di fondamentale importanza conoscerli per poter implementare il servizio di REST API lato back-end richiesto.

Problematiche a livello architetturale:

- * La quantità di REST API da migrare era troppo elevata per il tempo a disposizione.
- * Il database in uso si è rivelato non essere in forma normale e di conseguenza dava problemi come la ridondanza dei dati.

1.5 Soluzione scelta

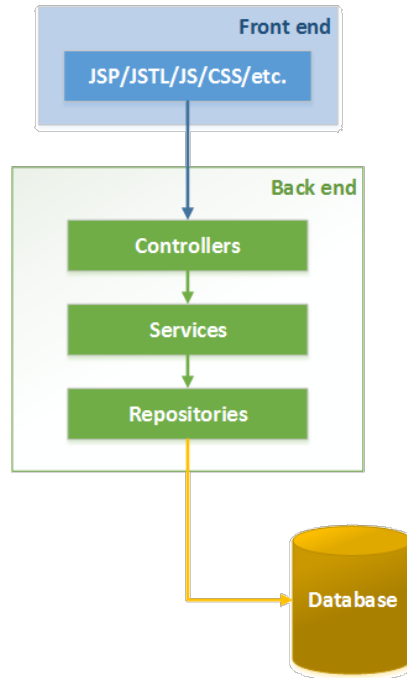
E' stato scelto di sviluppare con un architettura di tipo layered architecture. Questo è uno degli stili architetturali più utilizzati quando si sviluppa un monolite. L'idea dietro a questa architettura è che i moduli con funzionalità simili sono organizzati in livelli orizzontali. Quindi ogni livello svolge uno specifico ruolo nell'applicazione.

La layered architecture astrae la visione del sistema nel suo insieme, fornendo dettagli sufficienti per comprendere ruoli e le responsabilità dei singoli livelli e le relazioni che intercorrono tra loro.

La motivazione che ha portato alla scelta di questo stile architetturale è un'analisi fatta che ha rivelato la layered architecture adattarsi molto bene al servizio di REST API che si voleva andare a realizzare ed inoltre il fatto che molti framework per lo sviluppo di applicativi back-end si basano su esso, tra cui Spring e NestJS, che sono fondati sul pattern controller-service-repository. Un pattern che sfrutta la layered architecture, creando tre diversi livelli:

- * controller: è il livello più alto ed è l'unico responsabile dell'esposizione delle funzionalità in modo che possano essere consumate da entità esterne.
- * service: livello centrale, gestisce tutta la business logic.
- * repository: livello più basso, è responsabile di salvare e recuperare i dati da un sistema di persistenza, come un database.

Questa struttura viene utilizzata per effettuare una buona separazione delle responsabilità. L'architettura usata da Spring e NestJS ha portato a sceglierli come framework per realizzare la parte back-end del progetto.



Non è prevista la creazione di un sistema di autenticazione per l'uso delle API, in quanto un altro studente tirocinante si stava occupando della creazione di questa parte.

1.6 Descrizione del prodotto ottenuto

Al momento è disponibile un back-end contenente le REST API sviluppate in NestJS, utilizzabile, in quanto non potendo migrare l'intero set di REST API disponibili in Spring, come preventivato, sono state sviluppate tutte le REST API più importanti per effettuare le operazioni CRUD più comuni.

Le REST API espongono un'interfaccia compatibile con quello che ormai è uno standard per la comunicazione con servizi di tipo REST. Ovvero per comunicare con le REST API bisogna fare delle richieste HTTP a degli specifici endpoint con i seguenti metodi HTTP:

- * GET: per ottenere delle risorse dal servizio REST
- * POST: per creare una nuova risorsa nel servizio REST
- * PUT: per modificare una risorsa nel servizio REST
- * DELETE: per eliminare una risorsa dal servizio REST

E' presente poi un servizio schedato che ogni due minuti in maniera autonoma va

a fare il polling da un file XML online, contenente gli stati aggiornati dei sensori. Questo servizio registra poi le variazioni, rispetto al polling precedente, nel servizio di persistenza.

Il file XML viene scritto e gestito dai produttori dei sensori di parcheggio, quindi non è compito di questo progetto gestirne il funzionamento. Il funzionamento di questo file è comunque abbastanza banale, in quanto ad ogni variazione di stato il sensore di parcheggio va semplicemente ad aggiornare il record a lui associato all'interno del file.

1.7 Tecnologie utilizzate

Git

E' uno degli strumenti di controllo di versionamento più utilizzati. Facilita la collaborazione di più sviluppatori nella realizzazione di un progetto e permette con semplicità di spostarsi tra varie versioni del software realizzate. Nel progetto è stato utilizzato con il workflow Gitflow.

Visual Studio Code

E' un editor di codice sorgente sviluppato da Microsoft che aiuta molto lo sviluppatore durante la fase di sviluppo del codice in quanto evidenzia le parole chiave, segnala errori di scrittura, suggerisce snippet di codice. Possiede una grande libreria di estensioni facilmente installabili, per renderlo compatibile con praticamente qualsiasi linguaggio di programmazione.

Postman

E' un'applicazione che viene utilizzata solitamente per testare API. E' un client HTTP che testa richieste HTTP, utilizzando una GUI, attraverso la quale otteniamo diversi tipi di risposta in base alle API che andiamo ad interrogare.

Stoplight

E' una piattaforma per disegnare API. Grazie a questo strumento è possibile documentare in maniera rigorosa e su uno spazio in cloud un set di API. La piattaforma permette di specificare varie informazioni per ogni API, tra cui endpoint, parametri in ingresso attesi, possibili risposte con status code associato. Questo strumento è molto utile per gli sviluppatori front-end che devono chiamare le API di un servizio back-end, soprattutto grazie alla funzionalità che permette di effettuare il mock della risposta di un'API.

TypeScript

E' un superset di JavaScript, che aggiunge tipi, classi, interfacce e moduli opzionali al JavaScript tradizionale. Si tratta sostanzialmente di una estensione di JavaScript. TypeScript è un linguaggio tipizzato, ovvero aggiunge definizioni di tipo statico: i tipi consentono di descrivere la forma di un oggetto, documentandolo meglio e consentendo a TypeScript di verificare che il codice funzioni correttamente.

Node.js

E' un framework per realizzare applicazioni Web in JavaScript, permettendoci di uti-

lizzare questo linguaggio, tipicamente utilizzato nella client-side, anche per la scrittura di applicazioni server-side. La piattaforma è basata sul JavaScript Engine V8, che è il runtime di Google utilizzato anche da Chrome e disponibile sulle principali piattaforme, anche se maggiormente performante su sistemi operativi UNIX-like.

NestJS

E' un framework per la creazione di applicazioni lato server Node.js efficienti e scalabili. Utilizza JavaScript ma è costruito con e supporta completamente TypeScript. Aggiunge un livello di astrazione al framework Express, che a sua volta aggiunge astrazione al framework Node.js. Di conseguenza NestJS utilizza Node.js per eseguire il codice JavaScript prodotto dal codice TypeScript compilato.

Spring

Spring è un framework leggero, basato su Java. Questo framework integra soluzioni a vari problemi tecnici che si presentano con alta frequenza durante lo sviluppo software. Spring si basa su due design pattern fondamentali che sono l'Inversion of Control e Dependency Injection.

PostgreSQL

Chiamato anche Postgres, è un sistema di database relazionale a oggetti (ORDBMS), open source e gratuito. Le principali caratteristiche di Postgres sono affidabilità, integrità dei dati, funzionalità ed estensibilità, oltre alla propria community open source che gestisce, aggiorna e sviluppa soluzioni performanti e innovative.

Jest

1.8 Organizzazione del testo

[Il secondo capitolo](#) describe ...

[Il terzo capitolo](#) approfondisce ...

[Il quarto capitolo](#) approfondisce ...

[Il quinto capitolo](#) approfondisce ...

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- * gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- * per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*^[g];
- * i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Analisi dei requisiti

2.1 Confronto con gli stakeholders

E' stato fatto un incontro iniziale con il proponente, ovvero l'azienda Sync Lab, per definire con chiarezza i requisiti richiesti.

2.1.1 Servizio REST API

Nell'incontro sono state prese in considerazione le REST API scritte col framework Spring già esistenti di cui doveva esserne fatta la migrazione nel framework NestJS. E' emerso subito che la quantità di REST API da realizzare era troppo elevata per la quantità di tempo a disposizione.

E' stato quindi necessario fare una valutazione di quali fossero i servizi fondamentali che il servizio di REST API avrebbe dovuto esporre, per poter essere utilizzato senza che venissero a mancare funzionalità fondamentali per l'utilizzo a livello base del sistema.

2.1.2 Servizio di polling

Abbiamo poi valutato il secondo servizio importante da realizzare per questo progetto, ovvero la necessità di avere una lista sempre aggiornata dei sensori disponibili e del loro stato.

La rete dei sensori cresce in maniera dinamica, in quanto un nuovo sensore viene aggiunto/spostato da un parcheggio senza un'utenza manuale che informi il sistema di ciò che avviene.

Ma è necessario che quando un sensore si aggiunge alla rete, il sistema lo rilevi e ne mantenga lo stato aggiornato e la stessa cosa deve avvenire per quanto riguarda lo spostamento di un sensore, solo che in questo caso il sistema deve aggiornare le coordinate del sensore esistente, anziché aggiungerne uno nuovo.

Sono stati scelti una tipologia di sensori con GPS integrato, che ad ogni variazione di stato vanno ad aggiornare un record a loro associato in un file XML online contenente le loro informazioni compreso il loro stato.

Non avendo controllo sui sensori e quindi su dove vengano scritti i dati, con la

Polling	# chiamate http	# accessi lettura	# accessi scrittura
60 minuti	24	$24 * 10^3$	$24 * 10^3$
30 minuti	48	$48 * 10^3$	$48 * 10^3$
15 minuti	96	$96 * 10^3$	$96 * 10^3$
5 minuti	288	$288 * 10^3$	$1440 * 10^2$
2 minuti	720	$720 * 10^3$	$1440 * 10^2$
1 minuto	1440	$1440 * 10^3$	$1440 * 10^2$
1 millisecondo	$864 * 10^5$	$864 * 10^8$	$1440 * 10^2$

proponente si è deciso di effettuare un polling ogni 2 minuti al file XML e aggiornare lo stato del back-end.

Lo stato dei sensori è stato duplicato nel back-end del progetto oltre che sul file poiché questo dà dei benefici:

- * permette di organizzare i dati in maniera più consona e organizzata.
- * l'accesso ai dati diventa molto più veloce in quanto non si deve interrogare un file XML online in una posizione remota e sconosciuta ma viene interrogato il servizio di persistenza del back-end, di cui abbiamo pieno controllo.

La motivazione che ha portato ad eseguire il polling ogni due minuti è la seguente:

il tipo di servizio offerto non ha bisogno di essere un real-time system in quanto non crea problemi all'utente vedere una piazzola che si libera/occupa con un delta di intervallo di ritardo.

L'importante è che questo delta non sia troppo elevato, in tal caso i dati mostrati agli utenti sarebbero troppo inconsistenti per essere utili, mentre un delta troppo piccolo genera un carico di lavoro per l'applicazione troppo elevato.

Infatti effettuare il polling con un basso intervallo, degli ordini dei millisecondi, genera la produzione di molte chiamate http da parte del back-end per l'accesso al file XML e molti accessi al database in caso ci siano dati da inserire/aggiornare.

Vediamo un esempio pratico con una tabella che mostra il costo delle chiamate e accessi al database giornalieri al variare dell'intervallo di tempo del polling:

Per calcolare il numero medio di accessi al database è stata ipotizzata la presenza di 1000 sensori a sistema e che ogni cinque minuti la metà dei sensori abbiano bisogno di un aggiornamento (quindi 100 sensori devono essere acceduti in scrittura ad ogni minuto).

Quindi si verificano 1000 accessi in lettura ad ogni polling per verificare quali abbiano bisogno di aggiornamento. Gli accessi in scrittura variano in base al tempo trascorso dall'ultimo polling (gli accessi in scrittura sono molto più costosi di quelli in lettura).

Come vediamo dalla tabella e come auspicabile, con il polling ad un intervallo di ogni ora si effettuano solo 24 richieste http al giorno ma il delta di latenza di aggiornamento dei dati ad ogni ora li rende inutilizzabili.

D'altra parte un intervallo di un millisecondo per il polling rende i dati aggiornati quasi in tempo reale ma non è sostenibile effettuare un numero di richieste giornaliere dal back-end pari a $864 * 10^5$ (più di 86 milioni).

Un numero di richieste giornaliere pari a 720 è stato ritenuto accettabile, così come il numero di accessi al database indicati per la colonna dei 2 minuti e si è optato quindi per questa scelta; ritenendo il delta di ritardo di aggiornamento un valore accettabile per l'utente finale e il costo non di sovraccarico per il sistema.

Il servizio per il polling dei sensori deve essere realizzato in modo che sia separato da quello di REST API.

Sia per una separazione di responsabilità, che per una futura migrazione a un'applicazione basata su microservizi, in cui il servizio di polling deve diventare un microservizio a se stante, gestibile in maniera indipendente rispetto agli altri microservizi.

2.2 Entità

Per rendere più chiaro il dominio del progetto ed eliminare eventuali ambiguità è stato necessario documentare le entità di dominio, coinvolte nelle funzionalità fondamentali delle REST API, di cui si è deciso effettuare la migrazione.

Piazzola

Modella il rettangolo bianco dipinto sull'asfalto che delimita la zona in cui l'automobile viene messa in sosta. Ogni piazzola deve essere associata ad un parcheggio. Una piazzola può avere un solo sensore di parcheggio.

Ogni piazzola è caratterizzata da:

- * id: numero incrementale.
- * latitudine: stringa.
- * longitudine: stringa.

Parcheggio

Modella l'insieme di piazzole.

Ogni parcheggio è caratterizzato da:

- * id: numero incrementale.
- * latitudine: stringa.
- * longitudine: stringa.

Sensore

Modella il sensore. Esistono due tipi di sensore:

- * ambientale: misurano la qualità dell'aria e altri parametri nel parcheggio e possono coprire un'area di N piazzole. Sono gestiti da un'altro progetto di tirocinio, quindi non sono facenti parte di questo dominio di progetto.
- * di parcheggio: sensore posizionato sotto l'auto nella piazzola, che rileva la presenza o meno del veicolo. Questo tipo di sensore può essere associato a una sola piazzola.

Ogni sensore può avere una sola azienda manutentrice a lui associata. Ogni sensore è caratterizzato da:

- * id: numero incrementale.
- * nome: stringa.
- * batteria: stringa, indica la tensione della batteria in Volt.
- * carica: stringa, indica il livello di carica della batteria (da 1 a 3).
- * type: stringa, indica il tipo di sensore (ambientale o di parcheggio).
- * attivo: booleano.
- * ultimo sondaggio: data, indica l'ultima volta che è stato aggiornato lo stato del sensore.
- * da riparare: booleano, indica se il sensore deve essere riparato.
- * da caricare: booleano, indica se la batteria del sensore è scarica.
- * in aggiornamento: booleano, indica se il sensore stà aggiornando il suo software.

Manutentore

Modella l'azienda incaricata alla manutenzione dei sensori. Ogni manutentore è caratterizzato da:

- * id: numero incrementale.
- * nome: stringa, indica il nome del titolare dell'azienda.
- * cognome: stringa, indica il cognome del titolare dell'azienda.
- * azienda: stringa.
- * telefono: stringa.
- * email: stringa.

Misurazione sensore parcheggio.

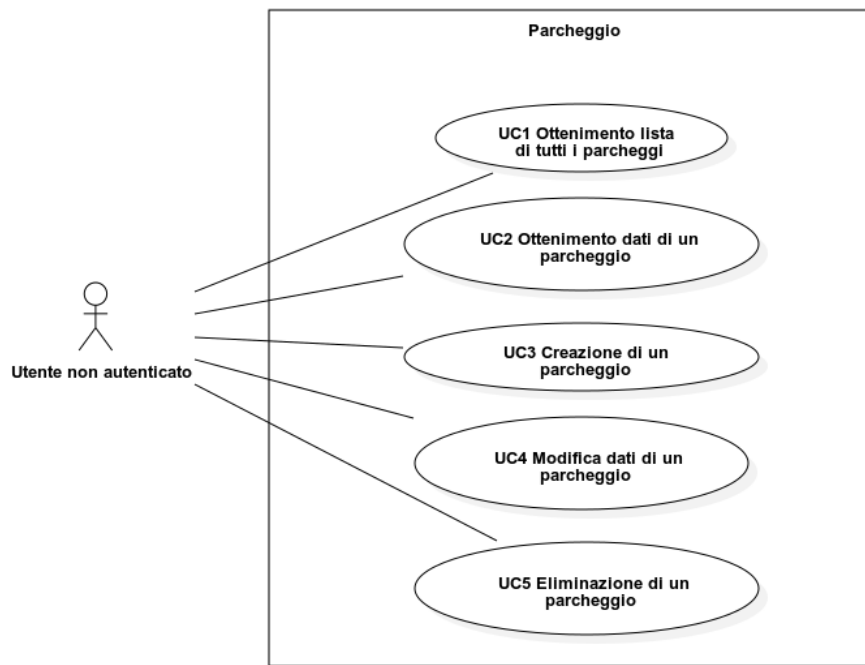
Modella la misurazione effettuata dal sensore di parcheggio. A differenza di un sensore ambientale, un sensore di parcheggio non salva uno storico di misurazioni fatte ma viene inserita solo l'ultima misurazione effettuata, sovrascrivendo la precedente. Ogni misurazione di un sensore di parcheggio è caratterizzata da:

- * id: numero incrementale.
- * indirizzo: stringa.

- * latitudine: stringa.
- * longitudine: stringa.
- * valore: booleano, indica se il veicolo è presente o meno sopra al sensore.
- * marca temporale: data, indica la data in cui è stata effettuata la misurazione.

2.3 Casi d'uso

Definite le entità di dominio si è proceduto con la creazione dei casi d'uso. Per maggior chiarezza i casi d'uso sono stati raggruppati per entità di dominio di appartenenza.



UC1 - Ottenimento lista di tutti i parcheggi.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto una lista di tutti i parcheggi.

Scenario principale:

1. l'utente richiede la lista di tutti i parcheggi.
2. l'utente ottiene una lista di tutti i parcheggi.

UC2 - Ottenimento dati di un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto i dati di un parcheggio.

Scenario principale:

1. l'utente richiede i dati di un parcheggio.
2. l'utente ottiene i dati di un parcheggio.

UC3 - Creazione di un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha creato un parcheggio.

Scenario principale:

1. l'utente richiede la creazione di un parcheggio.
2. l'utente crea un parcheggio.

UC4 - Modifica dati di un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha modificato un parcheggio.

Scenario principale:

1. l'utente richiede la modifica di un parcheggio.
2. l'utente modifica un parcheggio.

UC5 - Eliminazione di un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha eliminato un parcheggio.

Scenario principale:

1. l'utente richiede l'eliminazione di un parcheggio.
2. l'utente elimina un parcheggio.

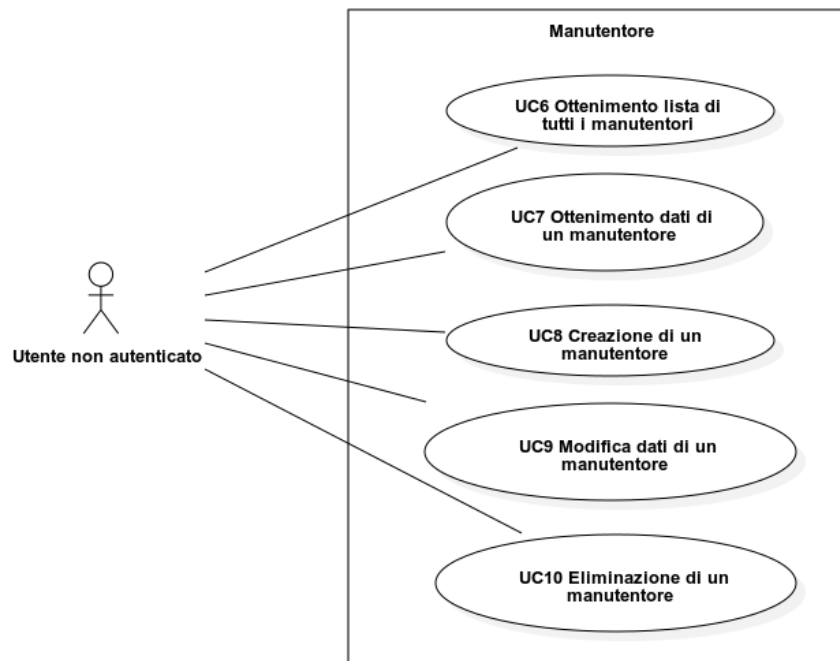
UC6 - Ottenimento lista di tutti i manutentori.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto una lista di tutti i manutentori.

Scenario principale:



1. l'utente richiede la lista di tutti i manutentori.
2. l'utente ottiene una lista di tutti i manutentori.

UC7 - Ottenimento dati di un manutentore.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto i dati di un manutentore.

Scenario principale:

1. l'utente richiede i dati di un manutentore.
2. l'utente ottiene i dati di un manutentore.

UC8 - Creazione di un manutentore.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha creato un manutentore.

Scenario principale:

1. l'utente richiede la creazione di un manutentore.
2. l'utente crea un manutentore.

UC9 - Modifica dati di un manutentore.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha modificato un manutentore.

Scenario principale:

1. l'utente richiede la modifica di un manutentore.
2. l'utente modifica un manutentore.

UC10 - Eliminazione di un manutentore.

Attori primari: utente non autenticato.

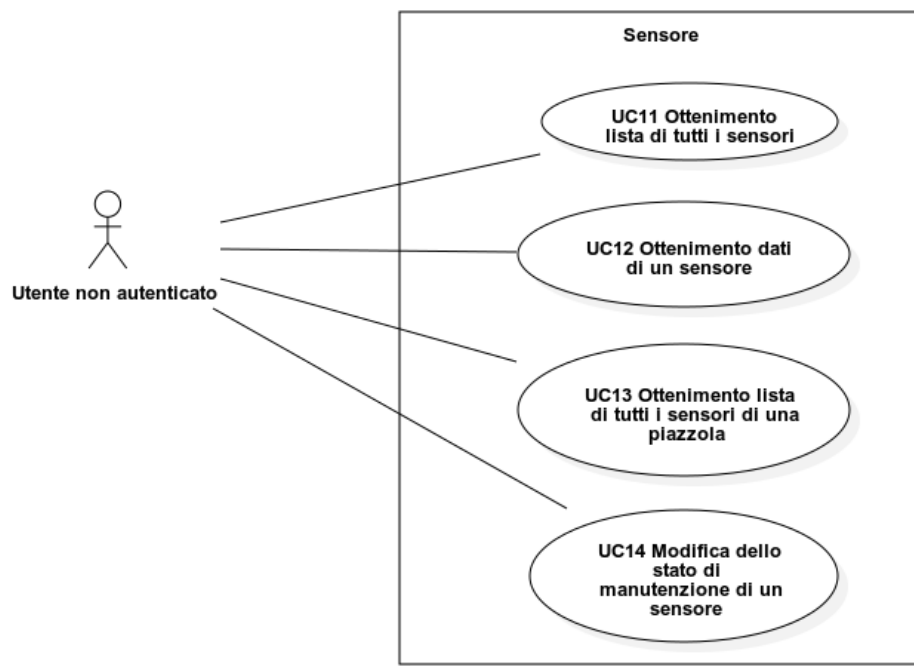
Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha eliminato un manutentore.

Scenario principale:

1. l'utente richiede l'eliminazione di un manutentore.
2. l'utente elimina un manutentore.

UC11 - Ottenimento lista di tutti i sensori.



Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto una lista di tutti i sensori.

Scenario principale:

1. l'utente richiede la lista di tutti i sensori.
2. l'utente ottiene una lista di tutti i sensori.

UC12 - Ottenimento dati di un sensore.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto i dati di un sensore.

Scenario principale:

1. l'utente richiede i dati di un sensore.
2. l'utente ottiene i dati di un sensore.

UC13 - Ottenimento lista di tutti i sensori di una piazzola.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto una lista di tutti i sensori di una piazzola.

Scenario principale:

1. l'utente richiede la lista di tutti i sensori di una piazzola.
2. l'utente ottiene una lista di tutti i sensori di una piazzola.

UC14 - Modifica dello stato di manutenzione di un sensore.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha modificato lo stato di manutenzione di un sensore.

Scenario principale:

1. l'utente richiede la modifica dello stato di manutenzione di un sensore.
2. l'utente modifica lo stato di manutenzione di un sensore.

UC15 - Ottenimento lista di tutte le piazzole di un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

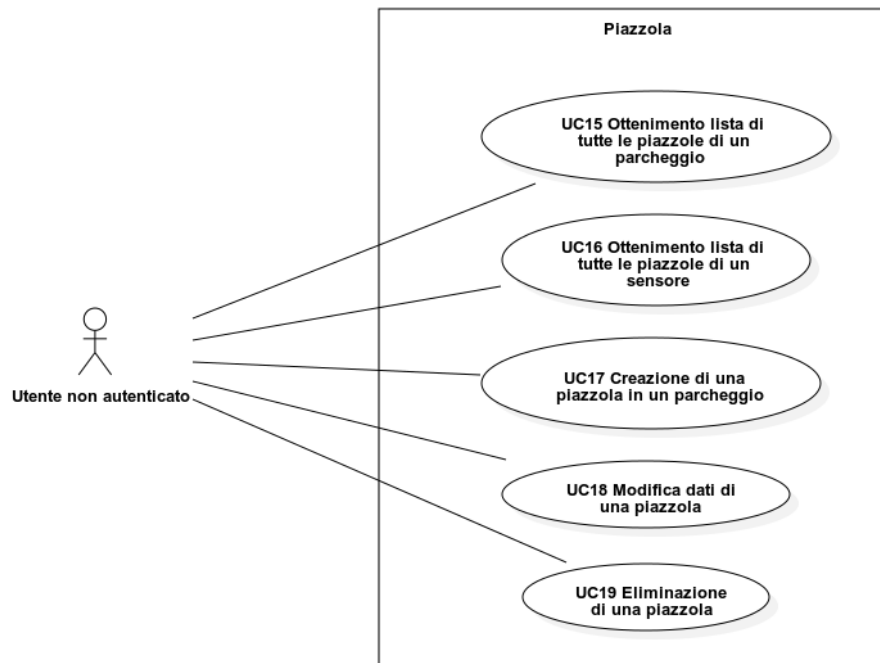
Post-condizioni: l'utente ha ottenuto una lista di tutte le piazzole di un parcheggio.

Scenario principale:

1. l'utente richiede la lista di tutte le piazzole di un parcheggio.
2. l'utente ottiene una lista di tutte le piazzole di un parcheggio.

UC16 - Ottenimento lista di tutte le piazzole di un sensore.

Attori primari: utente non autenticato.



Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto una lista di tutte le piazzole di un sensore.

Scenario principale:

1. l'utente richiede la lista di tutte le piazzole di un sensore.
2. l'utente ottiene una lista di tutte le piazzole di un sensore.

UC17 - Creazione di una piazzola in un parcheggio.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha creato una piazzola in un parcheggio.

Scenario principale:

1. l'utente richiede la creazione di una piazzola in un parcheggio.
2. l'utente crea una piazzola in un parcheggio.

UC18 - Modifica dati di una piazzola.

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha modificato una piazzola.

Scenario principale:

1. l'utente richiede la modifica di una piazzola.

2. l'utente modifica una piazzola.

UC19 - Eliminazione di una piazzola.

Attori primari: utente non autenticato.

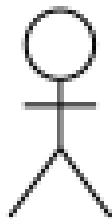
Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha eliminato una piazzola.

Scenario principale:

1. l'utente richiede l'eliminazione di una piazzola.
2. l'utente elimina una piazzola.

UC20 - Ottenimento ultima misurazione di un sensore di parcheggio.



Utente non autenticato

Attori primari: utente non autenticato.

Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

Post-condizioni: l'utente ha ottenuto l'ultima misurazione di un sensore di parcheggio.

Scenario principale:

1. l'utente richiede l'ultima misurazione di un sensore di parcheggio.
2. l'utente ottiene l'ultima misurazione di un sensore di parcheggio.

2.4 Tracciamento dei requisiti

Ogni requisito è identificato da un codice univoco nel seguente formato:

Codice	Descrizione	Rilevanza	Fonti
RF1	L'utente non autenticato deve poter ottenere la lista di tutti i parcheggi.	Obbligatorio	UC1
RF2	L'utente non autenticato deve poter ottenere i dati di un parcheggio.	Obbligatorio	UC2
RF3	L'utente non autenticato deve poter creare un parcheggio.	Obbligatorio	UC3
RF4	L'utente non autenticato deve poter modificare i dati di un parcheggio.	Obbligatorio	UC4
RF5	L'utente non autenticato deve poter eliminare un parcheggio.	Obbligatorio	UC5

Codice	Descrizione	Rilevanza	Fonti
RF6	L'utente non autenticato deve poter ottenere la lista di tutti i manutentori.	Obbligatorio	UC6
RF7	L'utente non autenticato deve poter ottenere i dati di un manutentore.	Obbligatorio	UC7
RF8	L'utente non autenticato deve poter creare un manutentore.	Obbligatorio	UC8
RF9	L'utente non autenticato deve poter modificare i dati di un manutentore.	Obbligatorio	UC9
RF10	L'utente non autenticato deve poter eliminare un manutentore.	Obbligatorio	UC10

* la prima lettera è sempre R, a indicare la parola requisito

* la seconda lettera indica il tipo di requisito:

- F per i requisiti funzionali
- Q per i requisiti qualitativi
- V per i requisiti di vincolo

* un numero progressivo che identifica in modo univoco il requisito.

Per maggior chiarezza i requisiti sono stati raggruppati per entità di dominio di appartenenza.

Codice	Descrizione	Rilevanza	Fonti
RF11	L'utente non autenticato deve poter ottenere la lista di tutti i sensori.	Obbligatorio	UC11
RF12	L'utente non autenticato deve poter ottenere i dati di un sensori.	Obbligatorio	UC12
RF13	L'utente non autenticato deve poter ottenere la lista di tutti i sensori di una piazzola.	Obbligatorio	UC13
RF14	L'utente non autenticato deve poter modificare lo stato di manutenzione di un sensore.	Obbligatorio	UC14

Codice	Descrizione	Rilevanza	Fonti
RF15	L'utente non autenticato deve poter ottenere la lista di tutte le piazzole di un parcheggio.	Obbligatorio	UC15
RF16	L'utente non autenticato deve poter ottenere la lista di tutte le piazzole di un sensore.	Obbligatorio	UC16
RF17	L'utente non autenticato deve poter creare una piazzola in un parcheggio.	Obbligatorio	UC17
RF18	L'utente non autenticato deve poter modificare i dati di una piazzola.	Obbligatorio	UC18
RF19	L'utente non autenticato deve poter eliminare una piazzola.	Obbligatorio	UC19

Codice	Descrizione	Rilevanza	Fonti
RF20	L'utente non autenticato deve poter ottenere l'ultima misurazione di un sensore di parcheggio.	Obbligatorio	UC20

Capitolo 3

Progettazione

3.1 Architettura del progetto

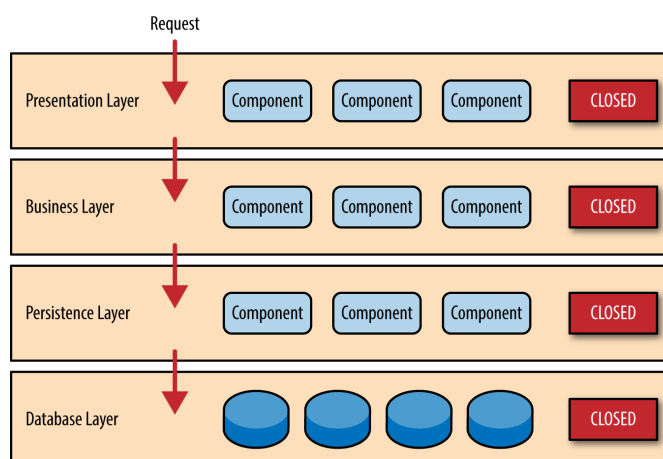
Dato le dimensioni contenute del progetto e per il fatto che deve essere fatta un'analisi comparativa con un'altro progetto, si è deciso di strutturarlo con un'architettura a monolite basata sulla layered architecture.

In questo modo viene velocizzata la realizzazione del progetto, a discapito della facilità di manutenzione ma non è un problema essendo questo progetto di dimensioni contenute ed in caso il progetto dovesse crescere fino al punto in cui risulti difficile mantenerlo è sempre possibile migrarlo in un progetto con un'architettura a microservizi.

3.1.1 Layered architecture

La layered architecture è uno degli stili architetturali più utilizzati. L'idea che sta dietro a questo tipo di architettura è che i moduli o i componenti con funzionalità simili sono organizzati in livelli orizzontali. Di conseguenza ogni livello svolge un ruolo specifico nell'applicazione.

La layered architecture non ha restrizioni sul numero di strati che l'applicazione può avere, in quanto lo scopo è avere livelli che promuovano il concetto di separazione delle responsabilità.



Solitamente ogni livello comunica solo con il livello sottostante. Il connettore tra ogni livello può essere una chiamata di funzione, una richiesta di query, un oggetto dati o qualsiasi connettore che trasmetta richieste o informazioni.

La denominazione dei livelli è abbastanza flessibile ma di solito un livello di presentazione, un livello di business e un livello fisico sono sempre presenti

Livello di presentazione

Il livello di presentazione contiene tutte le classi responsabili di presentare la visualizzazione delle informazioni all'utente finale. Idealmente questo è il solo livello con cui l'utente finale interagisce.

Livello di business

Il livello di business contiene tutta la logica che è richiesta dall'applicazione per poter soddisfare i suoi requisiti funzionali. Solitamente questo livello si occupa dell'aggregazione dei dati, della computazione e della richiesta dei dati. Quindi qui è dove viene implementata la logica principale dell'applicazione.

Livello fisico

Qui è dove sono salvati tutti i dati recuperabili dell'applicazione. Solitamente questo livello è chiamato anche livello di persistenza. Questo livello si occupa di interagire con il sistema in cui i dati sono mantenuti in maniera persistente, come ad esempio un database.

3.1.2 Motivazioni della scelta

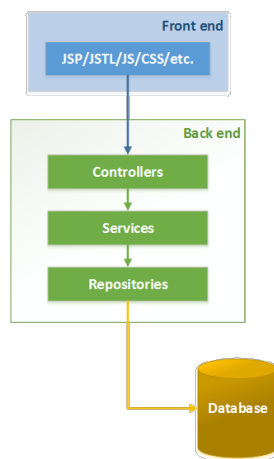
Le motivazioni che hanno portato a scegliere questo stile architetturale sono le seguenti:

- * Dato che la separazione delle responsabilità è la proprietà principale di quest'architettura, ogni livello di software ha la sua specifica funzione. Questo rende facile il dover aggiornare singoli livelli e permette al team di sviluppo di separare bene i carichi di lavoro tra i vari membri, che possono lavorare in maniera contemporanea su livelli diversi.
- * Per la proponente è importante avere una suite di test automatici per testare i vari componenti dell'applicazione. La layered architecture separando bene le responsabilità tra i livelli, permette di suddividere l'applicazione in componenti ben separati e quindi più facili da testare. Essendo ogni livello isolato dagli altri, è possibile creare casi di test di dimensione ridotta, in quanto le componenti di cui fare il mock sono poche.
- * L'isolamento tra i vari livelli permette di modificare un livello senza che la modifica intacchi gli altri livelli.
- * Nel caso l'applicazione diventi molto grande è possibile senza troppo sforzo avviare un processo di migrazione ad un'architettura a microservizi. La layered architecture lavora bene come monolite in un sistema con un'architettura ibrida tra un monolite e un sistema a microservizi. Questa architettura ibrida andrà a formarsi nel mentre che il monolite viene migrato in un sistema a microservizi, quindi è importante avere un'architettura a monolite che lavori bene in questo tipo di sistema.

Inoltre grazie alla separazione delle responsabilità della layered architecture è più facile andare a trasformare i componenti del monolite in microservizi.

3.2 Struttura software

E' stato scelto NestJS come framework di sviluppo del progetto dato che si adatta bene con la layered architecture, dato che usa il pattern controller-service-repository, un pattern basato sulla layered architecture, per permettere allo sviluppatore di sviluppare le proprie applicazioni.



Il controller è il livello responsabile per gestire le richieste in arrivo e ritornare le risposte al client. Esiste un meccanismo di routing che gestisce a quale controller inviare le richieste.

Il service è il livello responsabile della business logic.

Il repository è il livello chiamato livello di persistenza nella layered architecture.

Analizziamo in dettaglio la struttura del software:

3.2.1 IoC container

L'Inversion of Control container è un componente fondamentale di NestJS che permette l'applicabilità del pattern Dependency Injection all'interno di NestJS.

L'IoC container contiene un'istanza di tipo singleton per ogni classe dichiarata come controller o provider.

Il funzionamento dell'IoC container è il seguente:

quando viene avviata un'applicazione NestJS, il sistema runtime ricerca tutti controller e provider che sono stati dichiarati in dei moduli importati dal modulo root. Per ognuna di queste classi crea un'istanza usando il pattern singleton e la inserisce nell'IoC container.

Se però la classe da istanziare dichiarava una dipendenza con un altro controller o provider nel proprio costruttore, il sistema runtime applica in maniera automatica il pattern Dependency Injection; ovvero va a cercare un'istanza della dipendenza dichiarata nel costruttore della classe nell'IoC container, se presente la inietta nella

classe e crea l'istanza della nuova classe da inserire nell'IoC container.

Altrimenti va a creare l'istanza della classe che deve essere iniettata, prima della classe che dichiara la dipendenza (se possibile, in quanto la classe da iniettare potrebbe a sua volta richiedere una dipendenza e in tal caso si segue la successione di dipendenze fino a che non si trova una classe che possa essere istanziata) e la inietta nella classe che dichiara la dipendenza, poi ne crea un'istanza e la inserisce nell'IoC container.

3.2.2 Controller e provider

I due componenti fondamentali di NestJS sono i controller e i provider. Per dichiarare una classe come controller, bisogna applicare il decorator `@Controller`, sopra la definizione della classe, mentre per dichiarare una classe come provider bisogna applicare il decorator `@Injectable` sopra la definizione della classe.

```
@Injectable()
export class MaintainersRegistryService {
  constructor(private readonly maintainersRegistryRepository:
    MaintainersRegistryRepository){}

  getAllMaintainers(){
    return this.maintainersRegistryRepository.find();
  }

  async getMaintainerById(id: string){
    const maintainer =
      await this.maintainersRegistryRepository.findOne({
        where: {
          id: id
        }
      });

    if(isEmpty(maintainer))
      throw new NotFoundError('maintainer id not found');

    return maintainer;
  }

  async createMaintainer(maintainer: MaintainerRegistry){
    const insertResponse =
      await this.maintainersRegistryRepository.insert(
        maintainer);

    if(isEmpty(insertResponse.identifiers))
      throw new InsertError('problem to insert record');

    const maintainerInsertedId = insertResponse.identifiers
      [0].id;

    return this.getMaintainerById(maintainerInsertedId);
  }

  async editMaintainerById(id: string, maintainerRegistry:
    MaintainerRegistry){
```

```

    try{
        await this.getMaintainerById(id);
    }catch(error){
        throw(error);
    }

    const updateResponse =
        await this.maintainersRegistryRepository.update(id,
            maintainerRegistry);

    const numberOfRowsAffected = updateResponse.affected;

    if(numberRowsAffected !== 1)
        throw new UpdateError('problem to update record');

    return this.getMaintainerById(id);
}

async deleteMaintainerById(id: string){
    try{
        await this.getMaintainerById(id);
    }catch(error){
        throw(error);
    }

    const deleteResponse =
        await this.maintainersRegistryRepository.delete(id);

    const numberOfRowsAffected = deleteResponse.affected;

    if(numberRowsAffected !== 1)
        throw new DeleteError('problem to delete record');
}
}

```

I controller sono i componenti dedicati a gestire le richieste in ingresso e a fornire le risposte all'utente finale. NestJS considera come provider tutte le classi istanziabili e marcate con il decorator `@Injectable` che non sono controller; quindi sia classi di tipo service, che repository devono essere marcate con il decorator `@Injectable`.

```

@Controller('maintainers')
export class MaintainersRegistryController {
    constructor(private readonly maintainersRegistryService:
        MaintainersRegistryService){}

    @Get()
    getAllMaintainers(){
        return this.maintainersRegistryService
            .getAllMaintainers();
    }
}

```

```

@Get('/:id')
getMaintainerById(@Param('id') id: string){
    return this.maintainersRegistryService
        .getMaintainerById(id);
}

@Post()
async createMaintainer(@Body() maintainer: MaintainerRegistry
){
    return await this.maintainersRegistryService
        .createMaintainer(maintainer);
}

@Put('/:id')
editMaintainerById(
    @Param('id') id: string,
    @Body() maintainerRegistry: MaintainerRegistry,
){
    return this.maintainersRegistryService
        .editMaintainerById(id, maintainerRegistry);
}

@Delete('/:id')
@HttpCode(204)
deleteMaintainerById(@Param('id') id: string){
    return this.maintainersRegistryService
        .deleteMaintainerById(id);
}
}

```

E' possibile marcare con il decorator `@Injectable` anche classi non service o repository di cui si vuole che NestJS si occupi in maniera automatica di istanziare, iniettare le dipendenze dichiarate nel costruttore e inserire nell'IOC container.

I controller individuati sono i seguenti:

- * `MaintainersRegistryController`: gestisce le richieste/risposte relative al dominio dei manutentori.
- * `ParkingAreasController`: gestisce le richieste/risposte relative al dominio dei parcheggi.
- * `ParkingSensorsController`: gestisce le richieste/risposte relative al dominio delle misurazioni dei sensori di parcheggio.
- * `ParkingSensorsSensorsController`: gestisce le richieste/risposte relative al dominio delle misurazioni dei sensori di parcheggio di un sensore.
- * `ParkingSpotsController`: gestisce le richieste/risposte relative al dominio delle piazzole.
- * `ParkingSpotsParkingAreasController`: gestisce le richieste/risposte relative al dominio delle piazzole di un parcheggio.

- * `ParkingSpotsSensorsController`: gestisce le richieste/risposte relative al dominio delle piazzole di un sensore.
- * `SensorsController`: gestisce le richieste/risposte relative al dominio dei sensori.
- * `SensorsParkingSpotsController`: gestisce le richieste/risposte relative al dominio dei sensori di una piazzola.
- * `SensorsMaintenanceSensorsController`: gestisce le richieste/risposte relative al dominio della manutenzione dei sensori di un sensore.

3.2.3 Repository

I repository sono i componenti dedicati alla gestione della persistenza dei dati. Hanno quindi il compito di comunicare con la componente di archiviazione dati come un database. Nel progetto è stato utilizzato un database relazionale di tipo PostgreSQL.

NestJS è indipendente dal tipo di database scelto (relazionale o non relazionale). Infatti NestJS si interfaccia al database tramite uno strumento che si chiama TypeORM. TypeORM implementa una tecnica di programmazione chiamata ORM che converte i dati tra diversi tipi di sistemi usando linguaggi di programmazione OOP.

Uno strumento ORM incapsula il codice necessario per manipolare i dati, senza aver bisogno di scrivere manualmente le query al database ma si interagisce direttamente con un oggetto nello stesso linguaggio che si sta usando.

In questo modo il database viene astratto e si diventa indipendenti dal tipo di database utilizzato, in quanto è compito dell'ORM tradurre la richiesta fatta in linguaggio di programmazione ad alto livello nella query al database.

Uno strumento come TypeORM offre quindi una grande flessibilità in quanto è possibile decidere di passare da un database relazionale a un database non relazionale in qualsiasi momento senza dover effettuare modifiche al livello di persistenza.

Senza un'ORM la migrazione da un database relazionale a un database non relazionale implica la riscrittura di tutte le query.

Il repository viene fornito e creato in maniera automatica da NestJS. Per fare in modo che ciò avvenga però è necessario dichiarare, all'interno del modulo in cui si vuole che NestJS crei il repository, nell'array di imports tramite il metodo `forFeature` della classe `TypeOrmModule` la lista di entità di cui si vuole creare un repository.

Il repository creato da NestJS include tutti i metodi necessari per le operazioni basilari CRUD (`find()`, `save()`, `update()`, `delete()` ecc.).

Spesso però abbiamo bisogno di effettuare query al database più complesse rispetto a quelle a disposizione nel repository creato da NestJS. Per fare ciò dobbiamo creare una nostra classe repository che estenda la classe `Repository`, che è una classe `Generic` definita all'interno di NestJS e si aspetta come tipo del `Generic` il tipo dell'entità di cui vogliamo creare il repository.

Creare il repository custom non è più necessario usare il metodo `forFeature` nella

classe modulo, ma va importato il repository custom come provider.

```
@Injectable()
export class SensorsRepository extends Repository<Sensor>{
  constructor(private dataSource: DataSource){
    super(Sensor, dataSource.createEntityManager());
  }

  getSensorsWithoutSensorMaintenance(){
    return this.dataSource
      .createQueryBuilder()
      .select('sensor')
      .from(Sensor, 'sensor')
      .leftJoin('sensor.sensorMaintenance', 'sensorMaintenance')
      .where('sensorMaintenance.id IS NULL')
      .getMany();
  }
}
```

Nel nostro repository oltre che ai metodi ereditati dalla classe padre Repository, possiamo creare i nostri metodi personalizzati per poter inserire le nostre query custom.

Ci sono 2 modi per creare le query:

- * tramite notazione pura SQL
- * tramite i metodi del Query Builder

E' fortemente consigliato l'utilizzo del Query Builder anziché usare la notazione SQL per 2 motivi:

1. La concatenazione dei metodi del Query Builder rende molto più chiaro e pulito il codice, quindi più facile da mantenere.
2. Nel caso di cambio di tipo di database le query continuano a funzionare, in quanto grazie al Query Builder, TypeORM le converte adattandole alla sintassi del database che si sta utilizzando. Mentre non viene effettuata alcun tipo di conversione per le query in notazione pura SQL.

I repository custom individuati sono i seguenti:

- * ParkingSensorsRepository: ha un metodo custom per aggiornare i timestamp dei parcheggi passati come parametro.
- * SensorsRepository: ha un metodo custom per ottenere i sensori che non hanno almeno una manutenzione.

3.2.4 Moduli

Un modulo è un concetto fondamentale in NestJS. Ogni applicazione ha almeno un modulo, chiamato modulo root. Avere solo un modulo non è un caso tipico per un'applicazione, solitamente ce ne sono svariati. I moduli sono utilizzati come modo per

organizzare i componenti di un'applicazione.

All'interno di uno stesso modulo devono essere presenti componenti appartenenti allo stesso dominio. Ad esempio il controller, il service e il repository dei sensori di parcheggio sono tre buoni candidati per essere racchiusi all'interno dello stesso modulo.

Grazie ai moduli si riesce a mantenere il codice ben organizzato separando le componenti per dominio di appartenenza e stabiliscono dei confini chiari tra i vari componenti. In questo modo NestJS ci aiuta a gestire la complessità e a sviluppare con principi SOLID, specialmente quando le dimensioni dell'applicazione crescono e/o quando il team cresce.

Per inserire una componente in un modulo deve essere dichiarata come controller o provider, all'interno del decorator `@Module` della classe modulo (i moduli da controller devono essere dichiarati nell'array `controllers` di `@Module`, mentre i moduli provider devono essere dichiarati nell'array `providers` di `@Module`).

Se un controller o un provider non è dichiarato in un modulo che viene incluso dal modulo root, NestJS non istanzierà la classe del componente e non verrà inserito nell'IoC container.

Un concetto fondamentale dei moduli è che le componenti (controller, service, repository, classi varie..) dichiarate come appartenenti ad un modulo hanno uno scope locale al modulo, quindi sono visibili solo tra di loro e non vedono i componenti appartenenti ad altri moduli.

E' un caso comune però che un componente di un modulo abbia bisogno di un componente appartenente ad un altro modulo e quindi lo dichiara come dipendenza. In questo caso NestJS darebbe errore in fase di compilazione, poiché come spiegato sopra un componente di un modulo A, non può vedere un componente di un modulo B.

Per risolvere questo problema NestJS permette di definire nel decorator `@Module` della classe modulo i componenti che quel modulo vuole esportare e quindi che abbiano visibilità pubblica (i componenti da esportare devono essere dichiarati nell'array `exports` di `@Module`).

In questo caso se una classe di un modulo B dichiara una dipendenza da un componente esportato da un modulo A, nel decorator `@Module` della classe modulo B deve essere dichiarato il modulo del componente che si vuole importare (i moduli da importare devono essere dichiarati nell'array `imports` di `@Module`).

```
@Module({
  imports: [
    SensorsModule,
  ],
  controllers: [
    ParkingSpotsController,
    ParkingSpotsParkingAreasController,
    ParkingSpotsSensorsController,
  ],
  providers: [
    ParkingSpotsService,
    ParkingSpotsRepository,
  ],
})
```

```
    ],  
    exports: [  
        ParkingSpotsService,  
    ],  
})  
export class ParkingSpotsModule {}
```

I moduli individuati sono i seguenti:

- * AutomapperCustomModule: contiene i componenti per effettuare il mappaggio da DTO a entità.
- * DtoValidatorModule: contiene i componenti per validare i campi di un DTO.
- * MaintainersRegistryModule: contiene i componenti appartenenti al dominio dei manutentori.
- * ParkingAreasModule: contiene i componenti appartenenti al dominio dei parcheggi.
- * ParkingSensorsModule: contiene i componenti appartenenti al dominio delle misurazioni dei sensori di parcheggio.
- * ParkingSpotsModule: contiene i componenti appartenenti al dominio delle piazzole.
- * SensorsModule: contiene i componenti appartenenti al dominio dei sensori.
- * SensorsMaintenanceModule: contiene i componenti appartenenti al dominio della manutenzione dei sensori.
- * SensorsScrapingModule: contiene i componenti appartenenti al dominio del polling dei sensori.

3.2.5 DTO

E' stata usata una classe di tipo DTO chiama SensorsScrapingDto. Questa classe viene utilizzata per rappresentare il contenuto del file XML online contenente lo stato dei sensori. Da questo oggetto vengono estratte le informazioni utili per rappresentare le entità di tipo sensore e misurazioni sensore con cui si va ad aggiornare il database.

Prima di effettuare la conversione da DTO a entità, i campi del DTO vengono validati, lanciando un'eccezione nel caso la validazione fallisca.

```
export class SensorScrapingDto{  
  
    id: string;  
  
    name: string;  
  
    address: string;  
  
    lat: string;
```

```
    lng: string;

    state: boolean;

    battery: string;

    active: boolean;

    constructor(){
        this.id = '0';
        this.name = '';
        this.address = '';
        this.lat = '0';
        this.lng = '0';
        this.state = false;
        this.battery = '';
        this.active = false;
    }
}
```

3.2.6 Eccezioni

NestJS ha un livello built-in che è responsabile di processare tutte le eccezioni non catturate durante l'esecuzione di un'applicazione.

Quando un'eccezione non viene catturata dal codice dell'applicazione viene catturata da questo livello, che in maniera automatica invia una risposta http al client user-friendly, evitando di far interrompere l'esecuzione del programma. Questa componente si chiama exception filter.

La risposta al client è user-friendly e con un messaggio appropriato se l'eccezione è di tipo `HttpException` o una sua sottoclasse.

Altrimenti viene risposto al client con un messaggio "internal server error" e status code 500.

Possono capitare eccezioni anche durante l'esecuzione di query tramite TypeORM. Essendo TypeORM una libreria esterna nessuna eccezione da lei lanciata viene catturata dal livello descritto sopra di NestJS.

Per evitare di interrompere l'esecuzione del programma in eccezioni non presenti in NestJS si è deciso di sovrascrivere l'exception filter globale fornito da NestJS con uno custom e di creare un set di eccezioni custom che vengono lanciate per problemi di persistenza nella business logic.

Questo exception filter è stato chiamato `TypeOrmExceptionHandler` e implementa l'interfaccia `ExceptionHandler` di NestJS.

`TypeOrmExceptionHandler` simula il comportamento dell'exception filter di NestJS catturando qualsiasi tipo di errore, rispondendo con "internal server error" e status code 500 in caso l'eccezione non sia riconosciuta e in più funziona anche se si verificano

eccezioni da librerie esterne; garantendo un buon livello di resilienza del programma.

Se le eccezioni sono dei tipi custom per TypeORM, lo status code e il messaggio viene vengono inviati in maniera appropriata all'utente finale.

Altri tipi di eccezione di TypeORM di tipo `QueryFailedError`, vengono analizzati in base al codice di errore; lo status code e il messaggio anche in questo caso vengono inviati in maniera appropriata al client. Ad esempio un eccezione di tipo `QueryFailedError` con codice errore 23505 indica un conflitto nel database, quindi viene inviata una risposta al client con status code 409 e messaggio "database error on unique constraint".

Avere un exception filter permette di spostare la responsabilità della gestione delle eccezioni in un livello apposito. In questo modo si facilita la manutenzione e si assicura coerenza nella gestione delle eccezioni.

Senza un exception filter dovrebbero essere i controller a gestire le eccezioni e modificare la loro risposta in base al tipo di eccezione ricevuta dal service (che ha usufruito del metodo del repository che ha lanciato l'eccezione).

In questo modo però si creano controller di grandi dimensioni rendendo meno pulito il codice e più difficile da mantenere. Inoltre questo approccio non garantisce che tutti i controller gestiscano la stessa eccezione allo stesso modo.

Ad esempio sviluppatori diversi potrebbero gestire in maniera diversa la stessa eccezione (messaggio di errore diverso, più messaggi di errore, numero e tipo di parametri di risposta diversi ecc..) generando confusione per il cliente finale.

Le eccezioni custom per TypeORM individuate sono le seguenti:

- * `NotFoundError`: gestisce errori dovuti alla richiesta di dati inesistenti.
- * `InsertError`: gestisce errori dovuti all'inserimento di dati.
- * `UpdateError`: gestisce errori dovuti all'aggiornamento di dati.
- * `DeleteError`: gestisce errori dovuti alla cancellazione di dati.

3.2.7 Logging

Capitolo 4

Verifica e validazione

4.1 Verifica

4.1.1 Criteri di verifica

E' stato stabilito con il proponente che il prodotto finale dovesse avere una suite di test automatici per testare la business logic del progetto con una copertura a livello di branch stabilita \geq al 90% e una copertura a livello linee di codice \geq al 60%.

La motivazione che ha portato a richiedere una copertura così alta a livello branch è dovuta al fatto che si vuole assicurare il corretto funzionamento di tutti i possibili casi di errore anche in caso di futura manutenzione del software.

E' inoltre importante che sia mantenuta la coerenza stabilita per i casi di errore e nel caso alcuni branch non fossero coperti da casi di test, uno sviluppatore potrebbe inconsciamente modificare il comportamento del programma in caso di un errore, generando un comportamento inaspettato del programma per l'utente finale.

E' molto importante per la proponente che questa cosa non succeda e quindi è stata richiesta una copertura a livello branch \geq al 90%.

4.1.2 Strumenti utilizzati

NestJS fornisce al suo interno uno strumento per scrivere vari tipi di test, tra cui unit test, end to end test, integration test e così via.

Lo strumento di test usa Jest, un framework apposito per scrivere test automatici ed effettuare il mock dei componenti. Jest funziona su progetti che includono React, Babel, TypeScript, Node, Angular, Vue.

I file test in NestJS per conformità sono stati nominati con lo stesso nome della classe che vanno a testare e devono terminare in `.spec.ts` se vogliamo che vengano visti da NestJS ed eseguiti.

Ogni caso di test deve essere racchiuso all'interno della dicitura `describe`, specificando nome della classe che si va a testare e una funzione di callback contenente i test

per i vari metodi.

A sua volta ogni specifico metodo che si va a testare deve essere racchiuso in un `describe` che deve contenere il nome del metodo e una funzione callback contenente tutti i test su quel metodo.

```
describe('MaintainersRegistryService', () => {
  describe('getMaintainerById', () => {
    it('should return the maintainer if found', async () => {
      jest.spyOn(maintainersRegistryRepository, 'findOne')
        .mockImplementation(() => Promise.resolve(
          maintainerRegistry));

      const response = maintainersRegistryService.
        getMaintainerById('1');

      await expect(response).resolves.toEqual(maintainerRegistry)
    });

    it('should throw a NotFoundError if the maintainer was not
      found', async () => {
      jest.spyOn(maintainersRegistryRepository, 'findOne')
        .mockImplementation(() => Promise.resolve(null));

      const response = maintainersRegistryService.
        getMaintainerById('1');

      await expect(response).rejects.toThrow(NotFoundError);
    });
  });
});
```

4.1.3 Progettazione

Per i criteri di verifica stabiliti con la proponente si è deciso di testare tutti i metodi dei service che avessero un numero di branch > 1 . In questo modo viene assicurata la copertura di tutti i casi di errore dato che i metodi con un solo branch (solitamente metodi di get delle informazioni di un'entità) in caso di errore non contengono logica di business per lanciare errori custom ma al massimo lanciano eccezioni `TypeORM` o `NestJS` catturate dall'exception filter che conforma la risposta per il client.

Scrivendo dei test che vanno a controllare i metodo con più branch si assicura che:

- * ogni branch può essere raggiunto
- * ogni branch si comporta come aspettato
- * se uno sviluppatore modifica un metodo, si assicura che i casi di errore (id passato non trovato, aggiornamento non riuscito ecc..) non vengano eliminati e si assicura il mantenimento della loro conformità con quanto stabilito con la proponente. Ad esempio si assicura che venga mantenuto il controllo che in caso di errore di aggiornamento di un'entità venga lanciata, dal service, un'eccezione di tipo `UpdateError`.

File	% Stmts	% Branch	% Funcs	% Lines
maintainers-registry.service.ts	91.66	100	83.33	91.17
parking-areas.service.ts	82.14	100	46.15	72.13
parking-sensors.service.ts	94.11	100	94.11	92.3
parking-spots.service.ts	88.88	100	50	87.87
sensors.service.ts	38.46	100	0	27.27
sensors-maintenance.service.ts	86.11	90	71.42	84.84
sensors-scraping.service.ts	65.95	100	36.36	51.78
maintainers-registry.service.ts	91.66	100	83.33	91.17

Per alleggerire i test e quindi velocizzarne l'esecuzione si è deciso di creare dei mock per tutte le dipendenze della classe da testare, repository compresi.

Fornire dei mock per le dipendenze significa fornire ai casi di test delle componenti fittizie che vanno a simulare il comportamento della componente originale.

Le componenti fittizie sono molto più piccole delle componenti originali e implementano solo la parte di codice necessaria a far funzionare il caso di test nel modo in cui ci si aspetta.

Jest fornisce dei metodi molto utili per creare il mock di un metodo di una classe e sovrascriverne il comportamento. Tramite il metodo `spyOn` infatti si va a specificare l'oggetto e il suo metodo di cui si vuole effettuare il mock.

Per evitare di dover caricare componenti reali, quindi molto pesanti ed effettuare il mock dei loro metodi, è stata usata una libreria esterna chiamata `@golevelup/ts-jest`, che permette di dichiarare l'intera classe di cui si vuole fare il mock e restituisce un oggetto con gli stessi metodi della classe che gli abbiamo specificato ma con il contenuto dei metodi vuoto, alleggerendo quindi notevolmente il peso della componente.

Di queste componenti viene poi effettuato il mock dei metodi necessari all'esecuzione dei test tramite il metodo `spyOn`, come descritto in precedenza.

4.1.4 Realizzazione

Come risultato di quanto progettato sono stata creata una suite di 16 casi di test, per un totale di 59 unit test.

4.1.5 Scheduler

Si è deciso di implementare il polling dei dati del sensore dal file XML online tramite la libreria `@nestjs/schedule` che integra il package `cron` di `Node.js`.

Questa libreria mette a disposizione uno strumento per poter eseguire il metodo di una classe ad intervalli di tempo regolari.

Per fare ciò bisogna importare nel modulo root, il modulo della libreria schedule in questo modo:

```
@Module({
  imports: [
    ScheduleModule.forRoot()
  ],
})
export class AppModule {}
```

Successivamente abbiamo a disposizione un decorator @Cron, da indicare sopra al metodo che vogliamo venga schedulato.

Come parametro del @Cron dobbiamo inserire la stringa, rispettando il cron pattern, che indica l'intervallo di tempo con cui vogliamo che NestJS esegua il metodo.

Nel nostro caso vogliamo eseguirlo ogni due minuti quindi gli passiamo la stringa `"*/2 * * * *"`.

Capitolo 5

Conclusioni

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia

Riferimenti bibliografici

James P. Womack, Daniel T. Jones. *Lean Thinking, Second Editon*. Simon & Schuster, Inc., 2010.

Siti web consultati

Manifesto Agile. URL: <http://agilemanifesto.org/iso/it/>.