

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
2.2	Entità	7
2.3	Casi d'uso	9
2.4	Tracciamento dei requisiti	16
3	Progettazione	19
3.1	Architettura del progetto	19
3.1.1	Layered architecture	19
3.1.2	Motivazioni della scelta	20
3.1.3	Struttura software	21
3.1.4	Repository	25
4	Analisi dei requisiti	29
5	Conclusioni	31
A	Appendice A	33
	Bibliografia	37

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

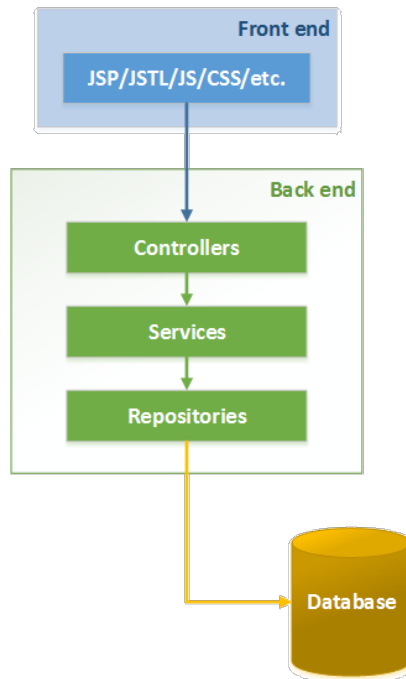
E' stato scelto di sviluppare con un architettura di tipo layered architecture. Questo è uno degli stili architetturali più utilizzati quando si sviluppa un monolite. L'idea dietro a questa architettura è che i moduli con funzionalità simili sono organizzati in livelli orizzontali. Quindi ogni livello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

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.

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.

Ho realizzato quindi un elenco di casi d'uso, in approvazione con la proponente, per avere più chiari i requisiti del progetto.

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.

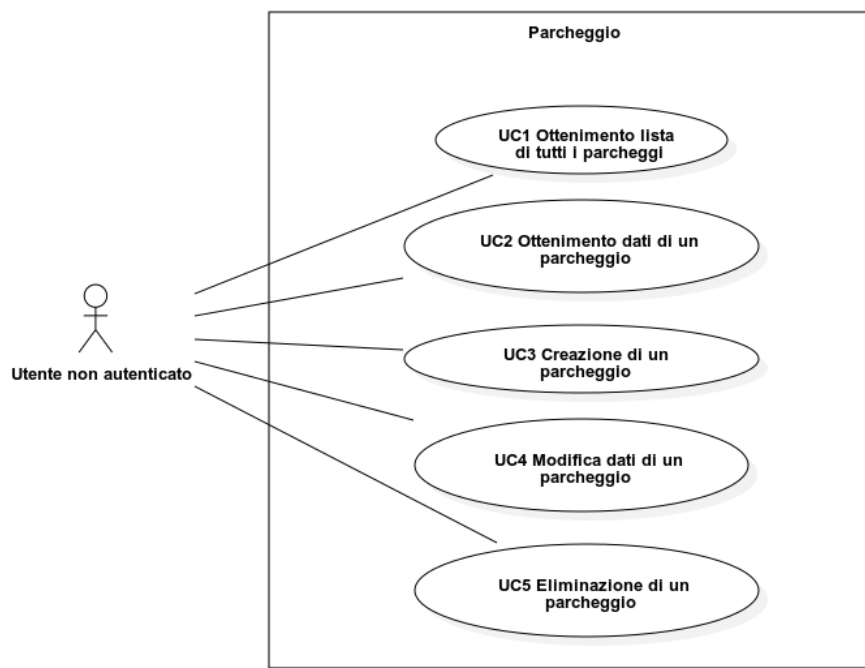
Modella la misurazione effettuata dal sensore di parcheggio. A differenza di un sensore ambientale, un sensore di parcheggio non salva uno storico di misurazioni fatte ma viene inserita solo l'ultima misurazione effettuata, sovrascrivendo la precedente. Ogni misurazione di un sensore di parcheggio è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

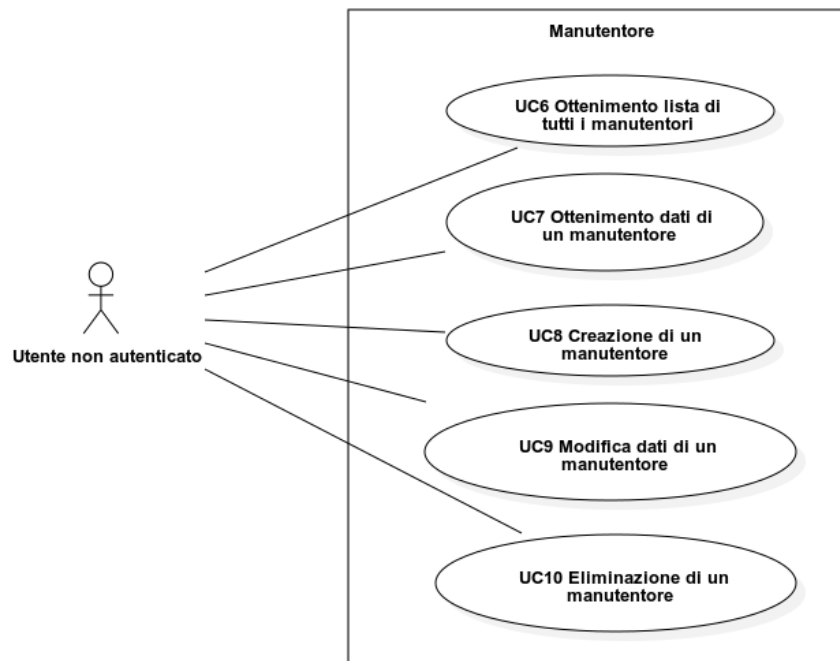
Attori primari: utente non autenticato.

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.

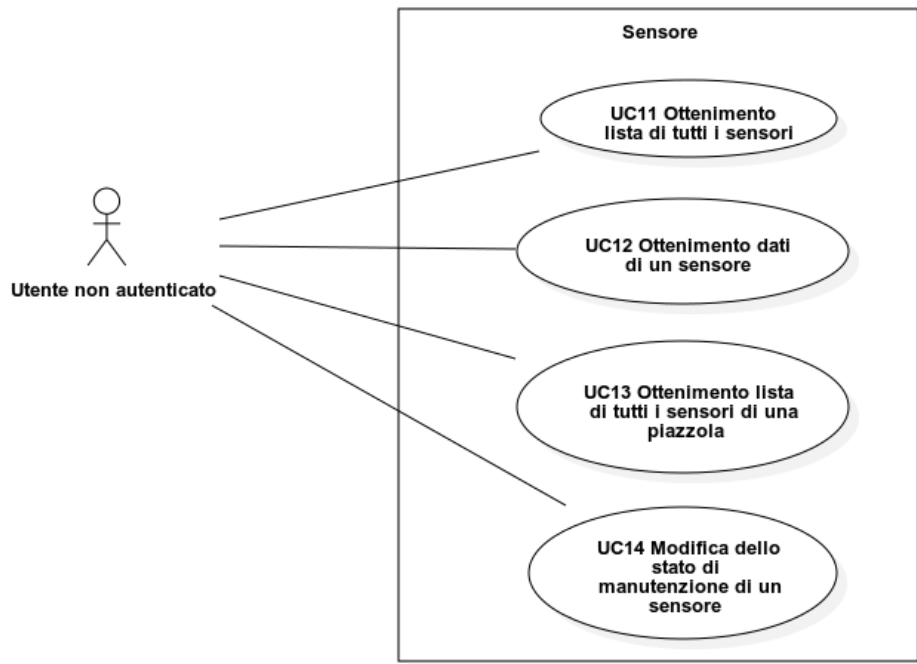
Attori primari: utente non autenticato.

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.

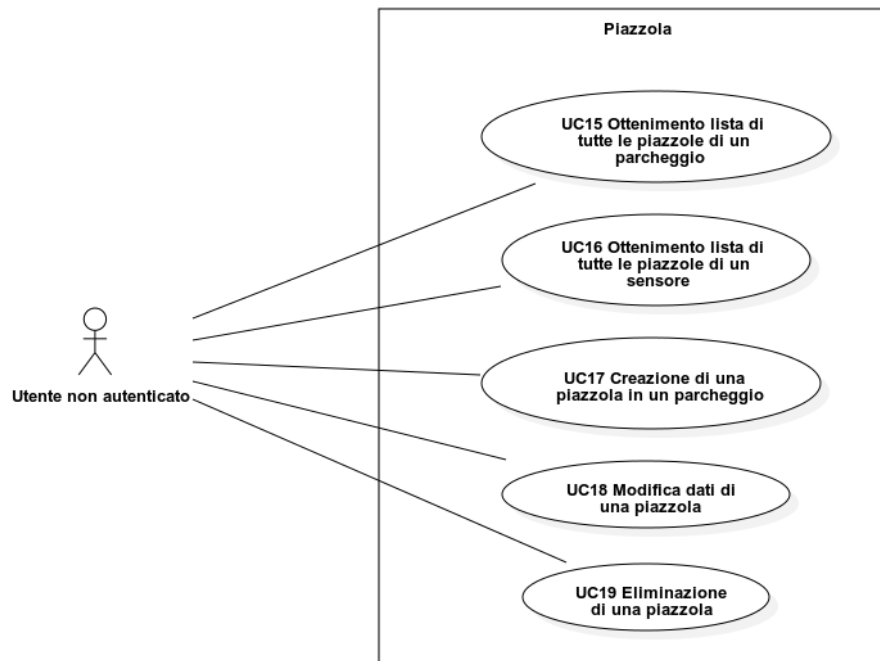
Scenario principale:

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.

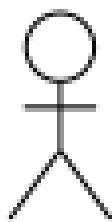
Precondizioni: l'utente è in possesso degli strumenti per poter effettuare la richiesta al sistema.

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.

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

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:

- * 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
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

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

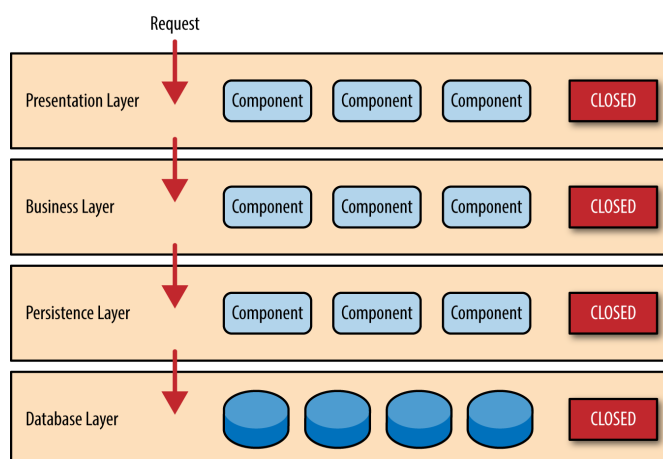
Dato le dimensioni contenute del progetto e per il fatto che deve essere fatta un'analisi comparativa con un'altro progetto, si è deciso di strutturarlo con un'architettura a monolite basata sulla layered architecture.

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

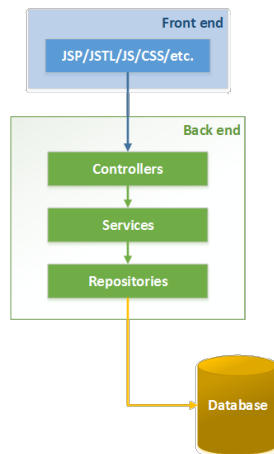
Le motivazioni che hanno portato a scegliere questo stile architetturale sono le seguenti:

- * 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.

3.1.3 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:

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.

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 (numberOfRowsAffected !== 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 (numberOfRowsAffected !== 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.

3.1.4 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.

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.

Capitolo 4

Analisi dei requisiti

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/>.