

```
(()),firstplural=  
s ((s),text=(,plural=(s,description=  
,descriptionplural=  
s,symbol=,symbolplural=,user1=,user2=,user3=,user4=,user5=,user6=,long=  
,longplural=  
s,short=(,shortplural=(s,counter=page,parent=,
```


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

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecento ore, del laureando Andrea Volpe presso l'azienda Sync Lab nel periodo che va dal 05/09/2022 al 28/10/2022.

Lo scopo dello stage era la realizzazione della migrazione di un [back-end](#) esistente sviluppato in Spring, in un [back-end](#) scritto in NestJS e la stesura un'analisi comparativa tra le due soluzioni.

In questo documento vengono descritte le varie fasi di lavoro effettuate durante lo stage. In particolare si descrive la fase di analisi e progettazione, ristrutturazione del database, verifica e validazione e infine l'analisi comparativa spiegando quali sono stati i punti di valutazione che hanno portato a preferire una soluzione rispetto all'altra.

“Poiché la disperazione era un eccesso che non gli apparteneva, si chinò su quanto era rimasto della sua vita, e riiniziò a prendersene cura, con l’incrollabile tenacia di un giardiniere al lavoro, il mattino dopo il temporale.”

— Alessandro Baricco

Ringraziamenti

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.

Vorrei inoltre esprimere la mia gratitudine al Prof. Baldan, relatore della mia tesi, per l’aiuto e il sostegno fornitomi durante la stesura del lavoro.

Ringrazio il dott. Daniele Zorzi, tutor aziendale e l’Ingegnere Fabio Pallaro, manager sede Sync Lab, per l’opportunità fornitami e il supporto dato durante l’intera attività di stage.

Padova, Dicembre 2022

Andrea Volpe

Indice

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).

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](#)^[8] 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 [Information Technology \(IT\)](#)^[8], sia dal punto di vista tecnologico, sia nel governo del cambiamento organizzativo.



Figura 1.1: Logo Sync Lab

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 il 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 [Application Programming Interface \(API\)](#)^[g] [Representational State Transfer \(REST\)](#)^[g], lato [back-end](#)^[g], realizzato da un precedente studente tirocinante con il framework Spring, in un servizio di [API REST](#), realizzato con un diverso framework chiamato NestJS. Il proponente ha deciso di fare la migrazione per effettuare un'analisi comparativa tra le due soluzioni, 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 nasce per agevolare un utente che vuole usufruire di un posto auto all'interno di un parcheggio e non vuole perdere tempo in cerca di un posto libero e nemmeno uscire di casa se i posti auto sono tutti occupati; infatti la webapp oltre a mostrare su una mappa le piazzole libere o occupate, segnala anche la disponibilità di posti auto in un parcheggio e il tutto viene fatto in tempo reale.

E' prevista poi la realizzazione di una sezione dedicata ai manutentori, in modo che possano monitorare in tempo reale lo stato dei sensori, facilitando quindi il processo di manutenzione.

Il progetto è formato da una parte di [front-end](#)^[g], realizzata con il framework Angular e una parte di [back-end](#) che consiste di un servizio di [API REST](#).

Questo progetto di stage riguarda la parte di [back-end](#), un altro mio collega stagista si sta occupando della parte di [front-end](#).




	Parceggio libero
	Parceggio occupato
	Sensore ambientale



Figura 1.2: Front-end Smart Parking

1.4 Problematiche riscontrate

Durante lo svolgimento del progetto si sono presentate alcune criticità, alcune dovute alla mancanza di conoscenza delle tecnologie da utilizzare.

Le problematiche riscontrate:

- * architettura a microservizi: avevo solo una conoscenza basilare della tecnologia, grazie al corso d'ingegneria del software ma non sufficiente per fare un'analisi di migrazione futura del progetto in un'architettura a microservizi;
- * 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 ed era di fondamentale importanza conoscerli per poter implementare il servizio di [API REST](#) richiesto;
- * la quantità di [API REST](#) da migrare era troppo elevata per il tempo a disposizione;
- * il modello della base dati ha dovuto subire adeguamenti rispetto alla prima versione per rappresentare nel modo migliore lo scenario funzionale.

1.5 Soluzione scelta

E' stato scelto di sviluppare il progetto con un architettura di tipo layered architecture. Questo è uno degli stili architetturali più utilizzati quando si sviluppa un software

monolitico. 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 adattarsi molto bene al servizio di [API REST](#) che si vuole andare a realizzare. Inoltre molti framework per lo sviluppo di applicativi [back-end](#) si basano su questo tipo di architettura, tra cui Spring e NestJS, che sono fondati sul pattern controller-service-repository; un pattern che sfrutta la layered architecture, creando tre diversi livelli di astrazione:

- * 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.

L'architettura usata da Spring e NestJS è proprio la stessa che si è deciso di usare nell'analisi progettuale fatta e quindi questi due framework sono stati scelti per realizzare la parte di [back-end](#) del progetto.

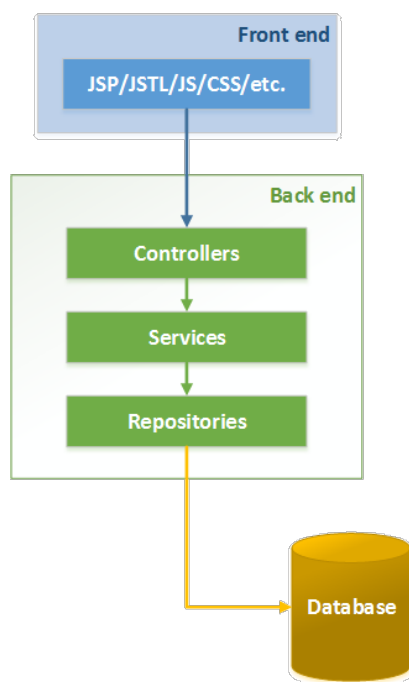


Figura 1.3: Controller-service-repository pattern

Non è prevista la creazione di un sistema di autenticazione per l'uso delle [API REST](#), in quanto un altro studente tirocinante sta realizzando questa parte.

1.6 Descrizione del prodotto ottenuto

Al momento è disponibile un [back-end](#) contenente le [API REST](#), sviluppato in Spring, pronto per poter essere messo in esercizio in ambiente di produzione.

Non potendo migrare l'intero set di [API REST](#), è stato preventivato di sviluppare tutte le [API REST](#) più importanti per effettuare le operazioni [Create Read Update Delete \(CRUD\)](#)^[g] più comuni.

Le [API REST](#) espongono un'interfaccia compatibile con quello che ormai è uno standard per la comunicazione con servizi di tipo [REST](#). Ovvero per comunicare con le [API REST](#) bisogna fare delle richieste [Hypertext Transfer Protocol \(HTTP\)](#)^[g] a degli specifici [endpoint](#) tramite 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](#).

Nel [back-end](#) è presente inoltre un servizio schedulato che ogni due minuti in maniera autonoma va a fare il polling da un file [eXtensible Markup Language \(XML\)](#)^[8] 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 [XML](#) 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.

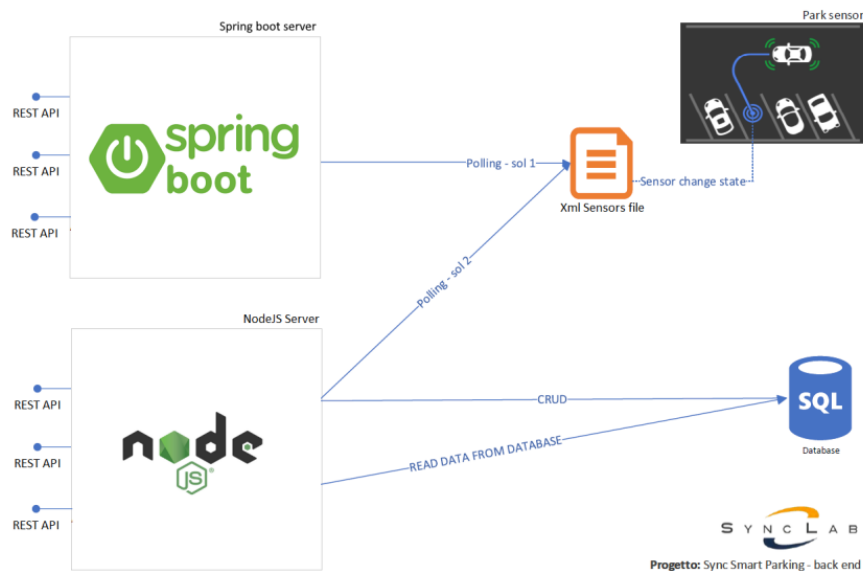


Figura 1.4: Polling sensori

1.7 Tecnologie utilizzate

Git

E' uno degli strumenti di controllo di versionamento più utilizzati. Facilita la collaborazione tra gli 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 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 [Graphical User Interface \(GUI\)](#)^[8], attraverso la quale otteniamo diversi tipi di risposta in base alle [API](#) che andiamo ad interrogare.

Stoplight

E' una piattaforma per progettare [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](#)^[g], 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 generare il [mock](#)^[g] della risposta di un'API, permettendo allo sviluppatore di effettuare le chiamate al [back-end](#) anche senza che questo sia stato ancora realizzato.

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 runtime system open source per eseguire applicazioni scritte 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 generato dalla compilazione del codice TypeScript.

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

Jest è un framework di unit test sviluppato da Facebook. Focalizzato sulla semplicità, è utilizzabile in qualsiasi progetto JavaScript. E' uno dei framework di test JavaScript più popolari in questi giorni e la scelta di default per alcuni framework come NestJS e React.

Winston

Winston è una delle librerie più famose per effettuare il logging su applicazioni Node.js. Permette di effettuare il logging su più livelli di informazione, formattare il logging in modo predefinito, scegliere la destinazione di output del log e molte altre opzioni.

Npm

E' uno dei gestori di pacchetti per il linguaggio JavaScript più popolare. E' il gestore di pacchetti predefinito per Node.js.

1.8 Organizzazione del testo

Il secondo capitolo descrive l'analisi dei requisiti.

Il terzo capitolo approfondisce la fase di progettazione.

Il quarto capitolo descrive la fase di ristrutturazione del database.

Il quinto capitolo descrive la fase di verifica e validazione.

Il sesto capitolo approfondisce l'analisi comparativa tra la soluzione in Spring e quella in NestJS.

Il settimo capitolo presenta le conclusioni finali sul progetto e sull'esperienza di stage.

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].

Capitolo 2

Analisi dei requisiti

In questo capitolo viene descritta la fase di analisi dei requisiti effettuata per la realizzazione del progetto.

2.1 Scopo del progetto

In precedenza è stato realizzato un set di [API REST](#) per interfacciarsi con le funzionalità di un progetto aziendale di Smart Parking che attraverso un servizio di polling raccoglie stato ed informazioni di alcuni sensori di parcheggio e le persiste in una base di dati relazionale.

Lo scopo di questo progetto è migrare le [API REST](#) e il servizio di polling, dalla soluzione realizzata con il framework Spring, in una soluzione realizzata con il framework NestJS.

2.2 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.

2.2.1 Servizio API REST

Nell'incontro sono state prese in considerazione le [API REST](#) esistenti di cui doveva essere fatta la migrazione. E' emerso subito che la quantità di [API REST](#) da migrare era troppo elevata per il tempo di stage a disposizione.

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

2.2.2 Servizio di polling

Abbiamo poi valutato il secondo punto importante per questo progetto, ovvero la necessità di avere le informazioni sullo stato dei sensori sempre aggiornate.

La rete dei sensori cresce in maniera dinamica, in quanto un nuovo sensore viene aggiunto/spostato da un parcheggio senza che un utenza manuale informi il sistema. Ma è necessario che quando un sensore si aggiunge alla rete, il sistema lo rilevi e ne mantenga lo stato aggiornato. 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 a sistema.

Sono stati scelti una tipologia di sensori con [Global Positioning System \(GPS\)](#)^[8] integrato, che ad ogni variazione di stato vanno ad aggiornare un record a loro associato in un file [XML](#) online, dove ogni record contiene le informazioni e lo stato del sensore.

Vediamo un estratto del file [XML](#) online:

```
<markers>
  <marker id="1" name="156A2C71" address="Padova Galleria Spagna" lat="45.389040"
    lng="11.928577" state="0" battery="3,7V" active="1"/>

  <marker id="2" name="156A2A71" address="Padova Galleria Spagna" lat="45.389029"
    lng="11.928598" state="1" battery="3,7V" active="1"/>

  <marker id="3" name="156A2B71" address="Padova Galleria Spagna" lat="45.389028"
    lng="11.928631" state="0" battery="3,7V" active="1"/>

  <marker id="4" name="156A2C72" address="Via Forcellini" lat="45.392648"
    lng="11.904846" state="0" battery="3,7V" active="1"/>
</markers>
```

Non avendo controllo sui sensori e quindi su dove vengano scritti i dati, con la proponente si è deciso di effettuare un polling ogni due minuti al file [XML](#) e aggiornare lo stato del [back-end](#).

Lo stato dei sensori è quindi ridondante in quanto è presente sia nel [back-end](#) del progetto che sul file [XML](#) online. La scelta di replicare lo stato dei sensori nel [back-end](#) è stata fatta per trarne i seguenti 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 è un [real-time system](#)^[8], 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 risultare utili. Mentre un delta troppo piccolo genera un carico di lavoro per l'applicazione troppo elevato.

Infatti effettuare il polling con un piccolo intervallo di tempo tra un polling e il successivo (dell'ordine dei millisecondi ad esempio), genera molte chiamate [HTTP](#) da

parte del [back-end](#) per accedere al contenuto del 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 di chiamate e accessi al database giornalieri al variare dell'intervallo di tempo del polling:

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	$864 * 10^5$	$864 * 10^8$	$1440 * 10^2$

Tabella 2.1: Costi giornalieri di chiamate http e accessi al database

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 (da ricordare che 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 li rende inutilizzabili.

D'altra parte un intervallo di un millisecondo rende i dati aggiornati quasi in tempo reale ma non è sostenibile effettuare un numero di richieste [HTTP](#) 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 che i costi delle chiamate e di accessi al database non siano di sovraccarico per il sistema.

Il servizio per il polling dei sensori deve essere realizzato in modo che sia separato da quello delle [API REST](#).

Sia per una separazione della 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.

2.3 Entità

Per rendere più chiaro il dominio del progetto ed eliminare eventuali ambiguità, si è ritenuto necessario documentare le entità di dominio coinvolte nelle funzionalità fondamentali delle [API REST](#).

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: misura la qualità dell'aria e altri parametri nel parcheggio. Possono ricoprire 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. 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: timestamp, 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 sta 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 delle misurazioni fatte ma viene registrata 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: timestamp, indica la data in cui è stata effettuata la misurazione.

2.4 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.

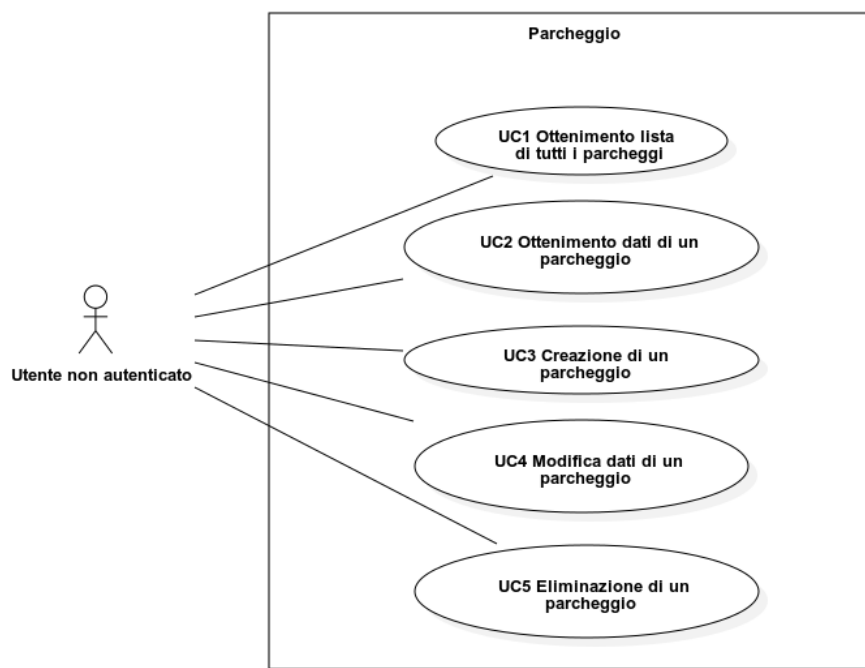


Figura 2.1: Casi d'uso parcheggio

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.

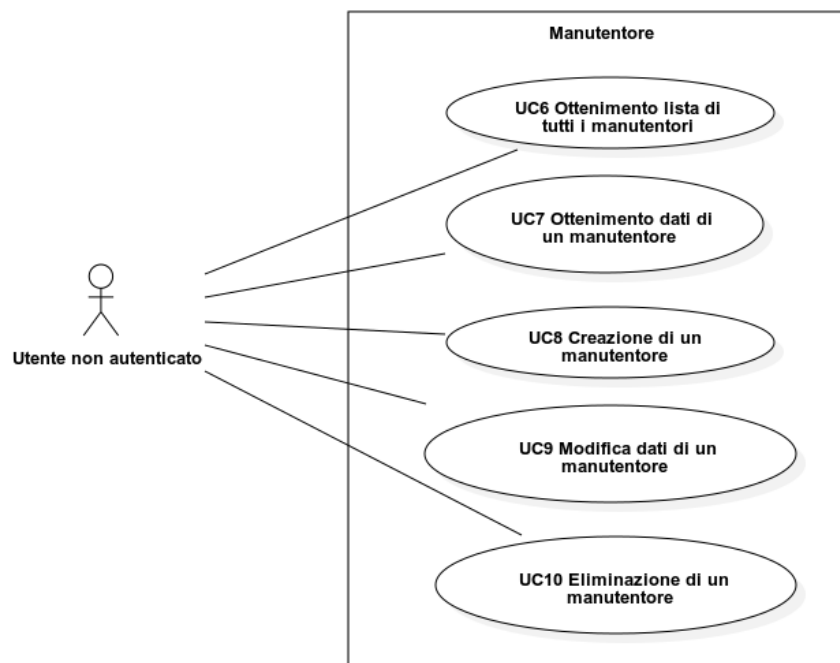


Figura 2.2: Casi d'uso manutentore

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.

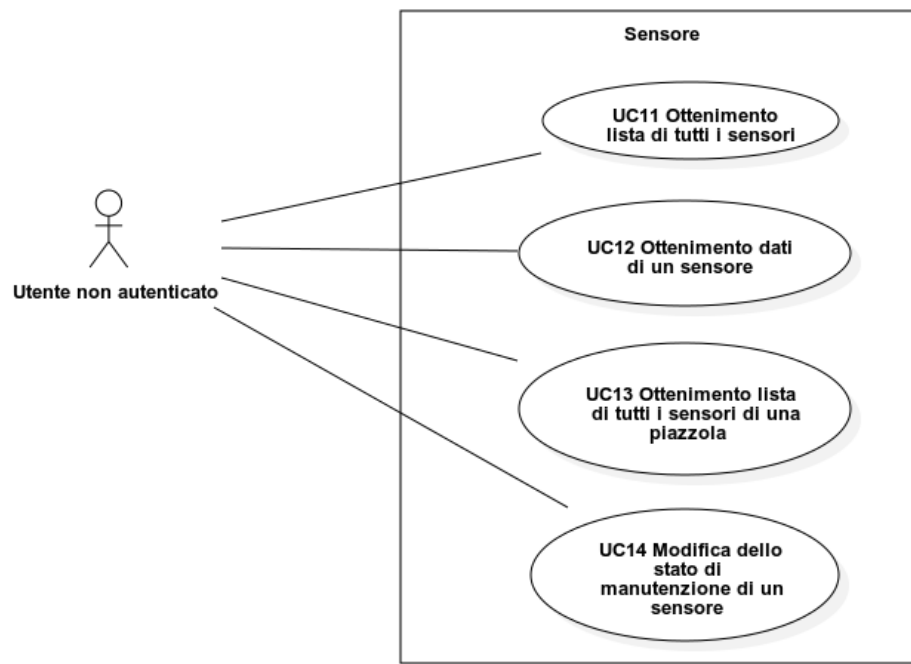


Figura 2.3: Casi d'uso sensore

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.

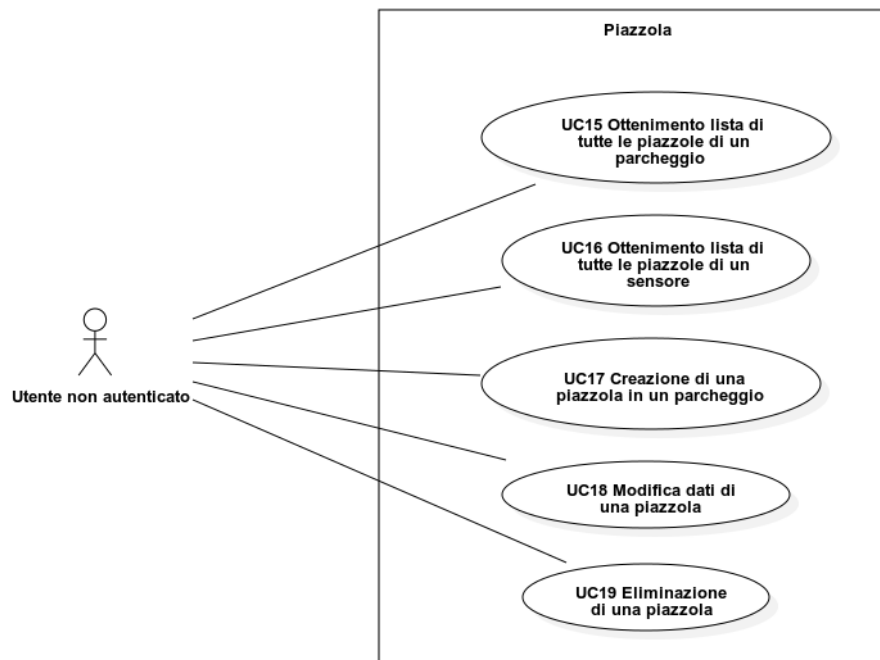


Figura 2.4: Casi d'uso piazzola

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.

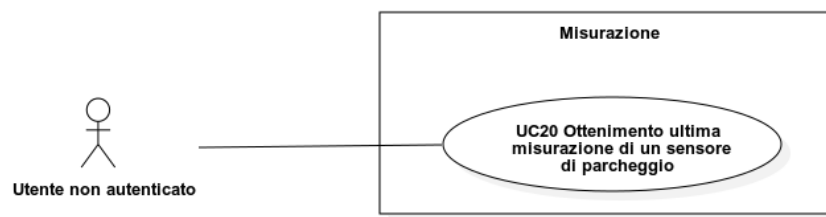


Figura 2.5: Casi d'uso misurazione sensore di parcheggio

UC20 - Ottenimento ultima misurazione di un sensore di 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 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.

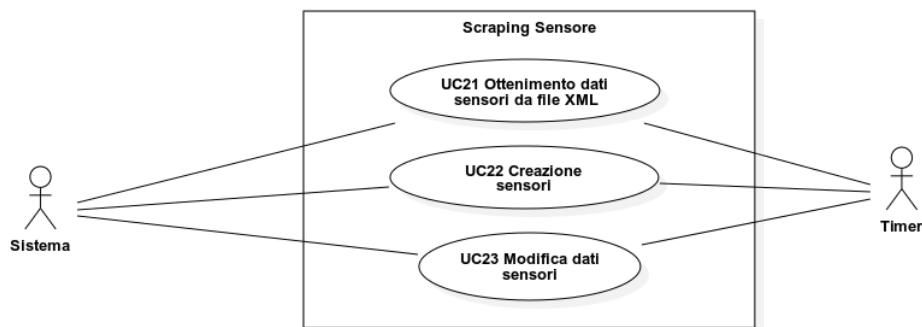


Figura 2.6: Casi d'uso polling sensore

UC21 - Ottenimento dati sensori da file XML

Attori primari: sistema.

Attori secondari: timer.

Precondizioni: il timer è scaduto segnalando di effettuare la richiesta al file [XML](#).

Post-condizioni: il sistema ha ottenuto una lista di tutti i dati dei sensori aggiornati.

Scenario principale:

1. il timer scade e segnala al sistema di effettuare una richiesta al file [XML](#).
2. il sistema effettua la richiesta al file [XML](#).

UC22 - Creazione sensori

Attori primari: sistema.

Attori secondari: timer.

Precondizioni: il sistema ha ottenuto i dati dei sensori aggiornati dal file [XML](#).

Post-condizioni: il sistema ha creato i nuovi sensori.

Scenario principale:

1. il sistema ha ottenuto i dati aggiornati dal file [XML](#).
2. il sistema verifica la presenza di nuovi sensori nei dati ottenuti.
3. il sistema crea i nuovi sensori.

UC22 - Modifica dati sensori

Attori primari: sistema.

Attori secondari: timer.

Precondizioni: il sistema ha ottenuto i dati dei sensori aggiornati dal file [XML](#).

Post-condizioni: il sistema ha modificato sensori.

Scenario principale:

1. il sistema ha ottenuto i dati aggiornati dal file [XML](#).
2. il sistema verifica la presenza di variazioni nei dati ottenuti rispetto ai dati dei sensori esistenti.
3. il sistema modifica i sensori.

2.5 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.

Requisiti funzionali

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

Tabella 2.2: Requisiti funzionali parcheggio

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

Tabella 2.3: Requisiti funzionali manutentore

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

Tabella 2.4: Requisiti funzionali sensore

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

Tabella 2.5: Requisiti funzionali piazzola

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

Tabella 2.6: Requisiti funzionali misurazione sensore di parcheggio

Codice	Descrizione	Rilevanza	Fonti
RF21	Il sistema deve richiedere ogni due minuti i dati dei sensori aggiornati da un file XML online.	Obbligatorio	UC21
RF22	Il sistema deve creare i nuovi sensori ottenuti dal file XML .	Obbligatorio	UC22
RF23	Il sistema deve modificare i sensori esistenti in base ai dati ottenuti dal file XML .	Obbligatorio	UC23

Tabella 2.7: Requisiti funzionali polling sensore**Requisiti di qualità**

Codice	Descrizione	Rilevanza	Fonti
RQ1	Deve essere presente una suite di test automatici per testare la business logic con una copertura a livello branch $\geq 90\%$.	Obbligatorio	Capitolato
RQ2	Deve essere presente una suite di test automatici per testare la business logic con una copertura a livello linee di codice $\geq 60\%$.	Obbligatorio	Capitolato
RQ3	Deve essere presente una documentazione che spieghi le scelte progettuali fatte e i motivi che hanno portato ad effettuare tali scelte.	Obbligatorio	Capitolato

Tabella 2.8: Requisiti di qualità

Requisiti di vincolo

Codice	Descrizione	Rilevanza	Fonti
RV1	Utilizzo del framework NestJS per realizzare l'applicazione.	Obbligatorio	Capitolato
RV2	Utilizzo di database PostgreSQL.	Desiderabile	Capitolato
RV3	Le API REST devono poter essere chiamate tramite protocollo HTTP .	Obbligatorio	Capitolato
RV4	La richiesta di ottenimento delle entità, all'applicazione, deve essere fatta tramite metodo GET.	Obbligatorio	Capitolato
RV5	La richiesta di inserimento delle entità, all'applicazione, deve essere fatta tramite metodo POST.	Obbligatorio	Capitolato
RV6	La richiesta di modifica delle entità, all'applicazione, deve essere fatta tramite metodo PUT.	Obbligatorio	Capitolato
RV7	La richiesta di cancellazione delle entità, all'applicazione, deve essere fatta tramite metodo DELETE.	Obbligatorio	Capitolato

Tabella 2.9: Requisiti di vincolo

Capitolo 3

Progettazione

In questo capitolo viene descritta la fase di progettazione effettuata per la realizzazione del progetto.

3.1 Architettura del progetto

Le dimensioni di questo progetto allo stato attuale sono contenute, inoltre deve essere fatta un'analisi comparativa tra la soluzione finale di questo progetto con un progetto già esistente. Per rendere l'analisi comparativa più efficace, è stato scelto di usare la stessa architettura dell'altro progetto, ovvero la layered architecture.

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

3.1.1 Layered architecture

La layered architecture è uno degli stili architetturali più utilizzati al giorno d'oggi. L'idea che sta dietro a questo tipo di architettura è organizzare i moduli o i componenti con funzionalità simili 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.

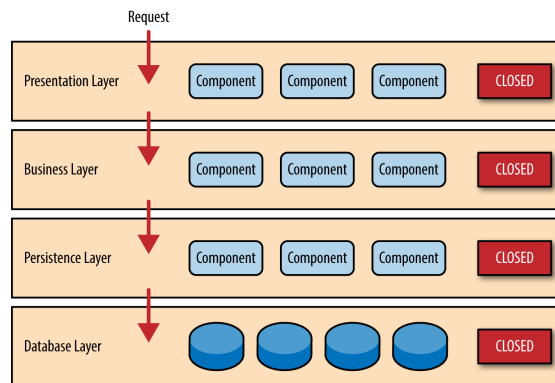


Figura 3.1: Layered architecture

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 sono sempre presenti almeno: un livello di presentazione, un livello di business e un livello fisico.

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 direttamente.

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, della computazione e della richiesta dei dati. Quindi qui è dove viene implementata la logica principale dell'applicazione.

Livello fisico

Nel livello fisico 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 persistiti, 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 del 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 componenti diverse;

- * 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 separate e quindi più facili da testare. Essendo ogni livello isolato dagli altri, è possibile creare casi di test di dimensione ridotte, 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;
- * 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 (lato monolite) anche in un sistema con un'architettura ibrida monolite/microservizi. Questo tipo di architettura ibrida solitamente si viene a formare nel processo di migrazione di un sistema monolitico a un sistema a microservizi.

Di conseguenza, in ottica di una futura migrazione in un sistema a microservizi (molto probabile che avvenga), è bene scegliere un monolite che lavori bene in un'architettura ibrida e quindi che sia facile da migrare in un sistema a microservizi.

Infatti grazie alla separazione delle responsabilità della layered architecture, è 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 utilizza la layered architecture, tramite il pattern controller-service-repository. Vediamo questo design pattern:

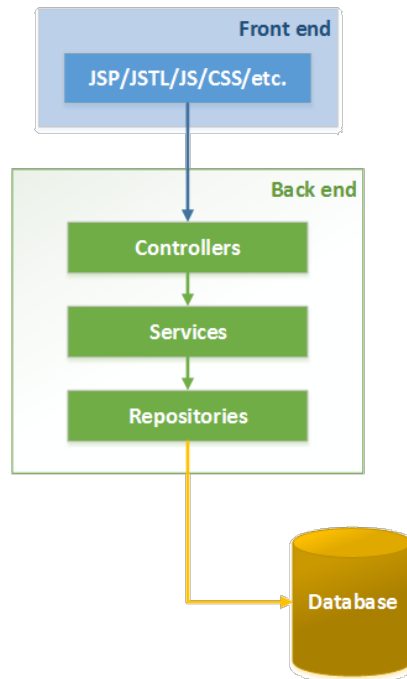


Figura 3.2: Controller-service-repository pattern

Il controller-service-repository pattern suddivide le componenti dell'applicazione su tre livelli fondamentali: il controller, il service e il repository.

Il controller è il livello più alto, mentre il repository quello più basso, in mezzo ai due c'è il service.

Il controller è il livello responsabile di gestire le richieste in arrivo e ritornare le risposte all'utente finale. Esiste un meccanismo di routing built-in nel framework, che gestisce a quale controller inviare le richieste.

Il service è il livello responsabile della business logic.

Il repository è il livello che viene anche chiamato livello di persistenza nella layered architecture e quindi interagisce con il sistema di persistenza dati, come il database.

Analizziamo in dettaglio le componenti che vanno a formare 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.

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 all'interno di 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 dichiara 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 da inserire nell'IoC container.

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

3.2.2 Controller e provider

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){}

  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);
  }
}
```

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('/: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 per fare in modo che NestJS, in maniera automatica ne inietti le dipendenze dichiarate nel costruttore, le istanzi e le inserisca nell'IoC container.

I controller dell'applicazione 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.

I service dell'applicazione individuati sono i seguenti:

- * `MaintainersRegistryService`: gestisce la business logic relativa al dominio dei manutentori;
- * `ParkingAreasService`: gestisce la business logic relativa al dominio dei parcheggi;
- * `ParkingSensorsService`: gestisce la business logic relativa al dominio delle misurazioni dei sensori di parcheggio;
- * `ParkingSpotsService`: gestisce la business logic relativa al dominio delle piazzole;
- * `SensorsService`: gestisce la business logic relativa al dominio dei sensori;
- * `SensorsMaintenanceService`: gestisce la business logic relativa al dominio della manutenzione dei sensori;
- * `SensorsScrapingService`: gestisce la business logic relativa al dominio del polling dei sensori.

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 gestione della persistenza come un [Database Management System \(DBMS\)](#)^[g]. 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 alla base di dati tramite il framework TypeORM. TypeORM implementa una tecnica di programmazione chiamata [Object Relational Mapping \(ORM\)](#)^[g], che converte i dati tra diversi tipi di sistemi usando linguaggi di programmazione [Object-Oriented Programming \(OOP\)](#)^[g].

Uno strumento [ORM](#) incapsula il codice necessario per manipolare i dati, senza

aver bisogno di scrivere manualmente le query al database, interagendo direttamente con un oggetto nello stesso linguaggio di programmazione 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 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 è però necessario dichiarare, all'interno del modulo in cui si vuole che NestJS crei il repository e in particolare nell'array *imports*, con 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 custom che estenda la classe Repository, che è una classe Generic built-in di NestJS, che si aspetta come tipo del Generic il tipo dell'entità di cui vogliamo creare il repository.

Se creiamo un repository custom non è più necessario usare il metodo *forFeature* nella classe modulo ma va importato il repository custom come provider.

Vediamo un esempio di repository custom:

```
@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 repository custom, oltre che ai metodi ereditati dalla classe padre Repository, possiamo creare i nostri metodi personalizzati che eseguono le nostre query custom.

Ci sono 2 modi per scrivere le query:

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

Il Query Builder è uno strumento built-in in TypeORM, per creare query SQL usando una sintassi elegante e conveniente, eseguirle e ottenere il risultato.

E' fortemente consigliato l'utilizzo del Query Builder anziché usare la notazione SQL pura 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 migrazione del database (ad esempio passando da PostgreSQL a Oracle) 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 dell'applicazione individuati sono i seguenti:

- * `ParkingSensorsRepository`: ha un metodo custom per aggiornare i timestamp delle misurazioni dei sensori di parcheggio passate come parametro;
- * `SensorsRepository`: ha un metodo custom per ottenere i sensori che non hanno almeno una manutenzione.

3.2.4 Moduli

Un modulo è una componente fondamentale di 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 molti. I moduli sono utilizzati 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 stabilendo dei confini chiari. 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.

In un modulo possiamo inserire solo componenti controller o provider. Per inserire un componente in un modulo, dobbiamo specificare il nome del componente all'interno del decorator `@Module` della classe modulo (i controller devono essere dichiarati nell'array *controllers* di all'interno di `@Module`, mentre i provider devono essere dichiarati nell'array *providers* all'interno 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 in 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 A abbia bisogno di un componente di un modulo B e quindi lo dichiara come dipendenza. Questa cosa, in NestJS, genera un errore in fase di compilazione per il problema di scope enunciato sopra.

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

In questo caso se un componente 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* all'interno di `@Module`).

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

I moduli dell'applicazione 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 [Data Transfer Object \(DTO\)](#)^[8] chiamata `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 misurazione del sensore di parcheggio 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 una componente built-in di NestJS, che in maniera automatica invia una risposta [HTTP](#) al client, evitando di far interrompere l'esecuzione del programma. Questa componente si chiama exception filter.

La risposta inviata al client dall'exception filter è user-friendly in quanto contiene un messaggio appropriato ma solo se l'eccezione catturata è di tipo `HttpException` o una sua sottoclasse. Altrimenti viene inviata una risposta con il generico messaggio "internal server error" e status code 500.

Possono essere lanciate eccezioni durante l'esecuzione di query tramite TypeORM. Essendo TypeORM una libreria esterna, nessuna eccezione da lei lanciata viene catturata dal exception filter di NestJS.

Per evitare di interrompere l'esecuzione del programma in questi casi, si è deciso di sovrascrivere l'exception filter built-in con uno custom che catturi tutte le eccezioni.

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

`TypeOrmExceptionHandler` simula il comportamento dell'exception filter di NestJS catturando qualsiasi tipo di eccezione, rispondendo con "internal server error" e status code 500 in caso l'eccezione non sia riconosciuta. Inoltre catturando anche le eccezioni di librerie esterne garantisce un buon livello di resilienza del programma.

E' stato deciso di creare un set di eccezioni custom che vengono lanciate per problemi collegati al servizio di persistenza.

In questo modo rendiamo più espliciti gli errori e inviamo in maniera appropriata status code e il messaggio all'utente finale.

Alcune eccezioni di TypeORM di tipo `QueryFailedError`, non vengono sovrascritte da eccezioni custom ma vengono analizzate in base al loro codice di errore.

In base a questo codice, lo status code e il messaggio vengono inviati in maniera appropriata all'utente finale.

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 separato; in questo modo si facilita la manutenzione e si assicura coerenza.

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.

Questo porta alla creazione di controller di grandi dimensioni rendendo il codice meno pulito 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 l'eccezione `QueryFailedError` 23505 (ad esempio usando un diverso messaggio di errore, usando più messaggi di errore, inserendo un numero e tipo di parametri di risposta diversi ecc..)

generando confusione per l'utente 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 Scheduler

Come spiegato nel capitolo di analisi dei requisiti, si è deciso di implementare il servizio di polling dei dati dei sensori con un intervallo di tempo pari a due minuti. Per farlo è stata utilizzata 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 poter usare la libreria bisogna importare, nel modulo root dell'applicazione, il modulo della libreria `schedule` in questo modo:

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

Avendo importato la libreria abbiamo a disposizione un decorator `@Cron`, da specificare sopra al metodo che vogliamo venga schedulato.

Come parametro del `@Cron` dobbiamo inserire una stringa contenente un cron pattern con l'intervallo di tempo con cui vogliamo che NestJS esegua il metodo.

Nel nostro caso volevamo eseguirlo ogni due minuti, quindi gli abbiamo passato la stringa `"*/2 * * * *"`.

In particolare, il metodo da schedulare è il metodo all'interno del service `Sensors-ScrapingService`, che si occupa di effettuare il polling, la persistenza dei dati dei sensori di parcheggio e delle misurazioni dei sensori di parcheggio.

3.2.8 Logging

Il logging in un'applicazione è un processo fondamentale che serve a salvare gli eventi che accadono durante la sua esecuzione.

Grazie al logging, gli sviluppatori possono accorgersi di potenziali attacchi o analizzare gli errori prima che questi interrompano i flussi di lavoro aziendali.

Il logging viene effettuato salvando su un file (solitamente con estensione .log) le informazioni rilevanti accadute in un particolare evento che si è deciso di loggare, come la richiesta da parte dell'utente finale di un servizio dell'applicazione.

Solitamente questi file di log raggiungono dimensioni importanti, a volte nell'ordine dei Gigabyte, quindi possono aver bisogno di un server dedicato per persisterli oppure di meccanismi di compressione file schedulati.

Per questo progetto è stato deciso di fare il logging di tutte le richieste alle [API REST](#) e tutte le volte che viene eseguito il servizio di polling schedulato.

In particolare, per le richieste alle [API REST](#), si vuole effettuare il logging delle seguenti informazioni:

- * metodo della richiesta (GET, POST, PUT, DELETE);
- * [endpoint](#) richiesto;
- * corpo della richiesta;
- * user agent del client;
- * indirizzo ip del client;
- * status code della risposta;
- * lunghezza del messaggio della risposta;
- * timestamp della richiesta.

Per il servizio di polling si vuole effettuare il logging delle seguenti informazioni:

- * timestamp del momento di avvio del polling;
- * timestamp del momento di terminazione del polling.

NestJS ha una classe built-in, chiamata Logger, che è possibile usare per effettuare il logging dell'applicazione.

Questa classe però permette solo di effettuare operazioni basilari e come spiegato nella guida ufficiale, per effettuare operazioni più complesse, come il salvataggio delle informazioni su file, è bene appoggiarsi ad una libreria esterna che si integra molto bene con NestJS, chiamata Winston.

Winston permette di definire la destinazione dei nostri log (nel nostro caso un file .log in una directory logs nel progetto) e la formattazione nel testo (nel nostro caso abbiamo usato una funzionalità di Winston che permette di usare la stessa formattazione dei log di NestJS) e molte altre funzionalità che non sono state utilizzate in questo progetto.

Le informazioni loggabili tramite Winston, hanno diversi livelli. Il livello del log indica la sua importanza:

- * error: log critico, qualcosa nell'applicazione non ha funzionato correttamente e alcune funzionalità potrebbero non rispondere correttamente;
- * warn: log di avviso, si è verificato qualcosa di inaspettato nell'applicazione;
- * info: log di informazione, si è verificato un evento all'interno dell'applicazione.

Esistono poi altri livelli di log che non sono stati usati in questo progetto, quindi si è deciso di non elencarli.

Winston permette di decidere i livelli d'informazione che vogliamo loggare, in particolare per questo progetto è stato effettuato il logging a livello error e info.

Le informazioni riguardanti le richieste alle [API REST](#) e il servizio di polling, sono state loggate a livello info.

Per loggare le richieste alle [API REST](#) è stato implementato un middleware `LoggerMiddleware` che intercetta tutte le richieste prima che vengano passate ai controller e salva in delle variabili interne le informazioni da loggare.

Per avere nel log, anche lo status code della risposta inviata al client dal controller, il `LoggerMiddleware` imposta un listener sull'evento `close` dell'oggetto `response`, prima di passare la richiesta al controller. In questo modo il log viene scritto solamente quando il controller ha terminato di gestire la richiesta (attivando il metodo listener) e abbiamo tutti i dati a disposizione.

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  constructor(
    @Inject(WINSTON_MODULE_PROVIDER)
    private readonly logger: Logger
  ) {}

  use(request: Request, response: Response, next: NextFunction): void {
    const { ip, method, originalUrl: url } = request;
    const userAgent = request.get('user-agent') || '';
    const body = JSON.stringify(request.body);

    response.on('close', () => {
      const { statusCode } = response;
      const contentLength = response.get('content-length');

      this.logger.info(
        `${method} ${url} ${body} ${statusCode} ${contentLength} - ${userAgent}
        } ${ip}`
      );
    });

    next();
  }
}
```

3.3 Servizio di polling

Sono state proposte due varianti per sviluppare il servizio di polling. Queste due varianti riguardano il modo in cui il servizio `SensorsScrapingService`, che si occupa di effettuare il polling dei dati dei sensori da un file [XML](#) online, debba comunicare col servizio di persistenza.

Le due varianti:

1. Far comunicare il `SensorsScrapingService` col servizio di persistenza effettuando delle chiamate [HTTP](#) alle [API REST](#) di cui ha bisogno.
2. Far comunicare il `SensorsScrapingService` direttamente col livello di servizio dei sensori e delle misurazioni dei sensori di parcheggio.

E' stato scelto di implementare il punto 2. Vediamo le motivazioni che hanno portato a effettuare questa scelta, analizzando pro e contro dei due rispettivi punti.

Pro del punto 1

- * far comunicare il `SensorsScrapingService` con un servizio di [API REST](#), separa bene le responsabilità e nel caso di migrazione a un applicazione basata su microservizi, il microservizio dedicato al polling dei sensori non deve cambiare il modo in cui interagisce con il servizio di persistenza, in quanto continua a chiamare le stesse [API REST](#) tramite il protocollo [HTTP](#).

Contro del punto 1

- * come analizzato nella fase di analisi dei requisiti il `SensorsScrapingService` deve effettuare 720 chiamate [HTTP](#) giornaliere per scaricare i dati dei sensori aggiornati. Quindi come minimo deve effettuare 720 chiamate [HTTP](#) giornaliere ulteriori anche alle [API REST](#), per verificare se ci sono sensori da aggiornare. Nella pratica le chiamate [HTTP](#) da fare sono di più, poiché dopo aver verificato se ci sono sensori da aggiornare ed eventualmente aggiornati, bisogna fare un'altra chiamata [HTTP](#) alle [API REST](#) per verificare se ci sono anche misurazioni dei sensori da aggiornare ed eventualmente aggiornarle. Il che porta il numero di chiamate [HTTP](#) da effettuare al servizio di [API REST](#), pari ad un minimo di 1440 giornaliere.

Pro del punto 2

- * far comunicare direttamente il `SensorsScrapingService` con il servizio dei sensori e delle misurazioni dei sensori di parcheggio riduce notevolmente i costi in termini di carico dell'applicazione, in quanto per persistere i dati è sufficiente chiamare un metodo del servizio da cui si dipende, senza dover fare chiamate [HTTP](#) che sono molto costose.

Contro del punto 2

- * nel caso di migrazione a un'applicazione basata su microservizi deve essere modificato il modo in cui il SensorsScrapingService persiste i dati dei sensori e delle misurazioni dei sensori di parcheggio. Probabilmente si dovranno effettuare chiamate [HTTP](#) al servizio di [API REST](#) (come il punto 1), in quanto il servizio di polling diventerà un microservizio indipendente, quindi il servizio dei sensori e delle misurazioni di parcheggio saranno fuori dal suo scope;
- * Solitamente è meglio evitare di creare dipendenze tra servizi, per mantenere il servizio il più isolato possibile, perseguendo il principio di separazione delle responsabilità.

Si è optato per il punto 2, in quanto questo è un progetto nato con l'obiettivo di fare un'analisi comparativa con un altro progetto esistente, di conseguenza, per non avere differenze prestazionali, si è scelta la variante più simile al servizio di polling implementato nell'altro progetto. Inoltre, è vero che è bene evitare le dipendenze tra servizi per ridurre l'accoppiamento tra le componenti e separare le responsabilità ma molto spesso non è possibile farlo e capita frequentemente di avere dipendenze tra loro.

3.4 API REST

Tramite lo strumento Stoplight sono state progettate 17 [API REST](#). Stoplight light è un ottimo strumento realizzato per progettare [API REST](#).

Essendo un servizio in cloud è accessibile a chiunque abbia necessità di avere la documentazione delle [API](#).

Stoplight permette di definire l'[endpoint](#) dell'[API](#), il metodo della richiesta per accedere alla risorsa (GET, POST, PUT, DELETE ecc..), eventuale corpo della richiesta, eventuale corpo della risposta, status code della risposta e altre informazioni utili a chi deve sviluppare le [API](#) e al client che deve effettuare le richieste.

Progettata un'[API](#) è possibile generare il [mock](#) della risposta, in questo modo gli sviluppatori [front-end](#) non hanno bisogno di attendere che il [back-end](#) venga realizzato per sviluppare la parte di [front-end](#).

Le [API REST](#), suddivise per dominio di appartenenza, sono le seguenti:

Parcheggio

Endpoint	Metodo	Codice risposta	Descrizione
/parking-areas	GET	200	Restituisce tutti i parcheggi
/parking-areas	POST	201	Crea un parcheggio
/parking-areas/{id}	GET	200	Restituisce il parcheggio con l'id richiesto
/parking-areas/{id}	PUT	200	Modifica il parcheggio con l'id richiesto
/parking-areas/{id}	DELETE	204	Elimina il parcheggio con l'id richiesto

Tabella 3.1: API REST parcheggio

Manutentore

Endpoint	Metodo	Codice risposta	Descrizione
/maintainers	GET	200	Restituisce tutti i manutentori
/maintainers	POST	201	Crea un manutentore
/maintainers/{id}	GET	200	Restituisce il manutentore con l'id richiesto
/maintainers/{id}	PUT	200	Modifica il manutentore con l'id richiesto
/maintainers/{id}	DELETE	204	Elimina il manutentore con l'id richiesto

Tabella 3.2: API REST manutentore**Sensore**

Endpoint	Metodo	Codice risposta	Descrizione
/sensors	GET	200	Restituisce tutti i sensori
/sensors/{id}	GET	200	Restituisce il sensore con l'id richiesto
/sensors/sensors-maintenance	GET	200	Restituisce tutti i sensori con le loro informazioni sulla manutenzione
/sensors/{id}/sensors-maintenance	GET	200	Restituisce il sensore con l'id richiesto con le sue informazioni sulla manutenzione
/sensors/{id}/sensors-maintenance	PUT	200	Modifica le informazioni di manutenzione del sensore con l'id richiesto
/parking-spots/{id}/sensors	GET	200	Restituisce tutti i sensori della piazzola con l'id richiesto

Tabella 3.3: API REST sensore

Piazzola

Endpoint	Metodo	Codice risposta	Descrizione
/parking-areas/{id}/parking-spots	GET	200	Restituisce tutte le piazzole del parcheggio con l'id richiesto
/parking-areas/{id}/parking-spots	POST	201	Crea una piazzola associata al parcheggio con l'id richiesto
/parking-spots/{id}	PUT	200	Modifica la piazzola con l'id richiesto
/parking-spots/{id}	DELETE	200	Elimina la piazzola con l'id richiesto
/sensors/{id}/parking-spots	GET	200	Restituisce tutte le piazzole del sensore con l'id richiesto

Tabella 3.4: API REST piazzola**Misurazione sensore di parcheggio**

Endpoint	Metodo	Codice risposta	Descrizione
/sensors/id/parking-sensors	GET	200	Restituisce tutte le misurazioni del sensore di parcheggio con l'id richiesto (ogni sensore di parcheggio ha solo una misurazione: l'ultima effettuata)

Tabella 3.5: API REST misurazione sensore di parcheggio

Capitolo 4

Ristrutturazione database

In questo capitolo viene descritta la fase di ristrutturazione del database per la realizzazione del progetto.

Durante la fase di studio del database ho rilevato la presenza di alcune criticità che hanno richiesto una revisione del modello logico.

Attraverso il confronto con un'altro stagista con il quale si condivideva la stessa base di dati è emersa la necessità di una revisione del modello logico.

Una revisione o modifica di un database all'interno di progetti ha impatti importanti, in questo particolare caso sono stati coinvolti tutor aziendale e sviluppatori interessati per condividere l'intervento.

Lavorando su branch indipendenti si sono riuscite a mantenere le modifiche in isolamento senza impatti, riuscendo a testare la correttezza dell'intervento in una base di dati locale.

Successivamente l'azienda valuterà in fase di rilascio le versioni adeguate da installare negli ambienti di sviluppo collaudo ed esercizio.

Il modello logico esistente era il seguente:

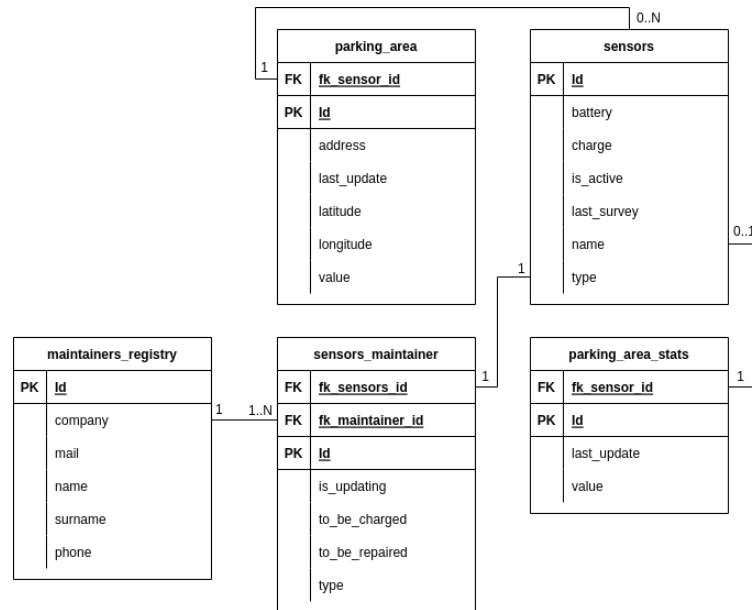


Figura 4.1: Modello logico pre-ristrutturazione

4.1 Anomalie rilevate

Partendo dalla premessa che il software è un prodotto in continua evoluzione, le prime versioni di prodotto possono presentare anomalie o bug.

La stessa progettazione del modello logico iniziale può possedere mancanze, lacune oppure subire variazioni da nuovi requisiti.

I framework di persistenza (es. JPA) riescono a facilitare questo tipo di interventi correttivi.

Seguono una serie di anomalie rilevate e di adeguamenti eseguiti rispetto alla prima versione del modello dati:

Nomi tabelle incoerenti

I nomi di alcune tabelle erano incoerenti con la funzionalità che andavano a svolgere:

`sensors_maintainer` è la tabella che contiene i dati di manutenzione dei sensori, un nome più appropriato potrebbe essere `sensors_maintenance`.

`parking_area` è la tabella che rappresenta la piazzola di parcheggio, un nome più appropriato potrebbe essere `parking_spots`.

Duplicazione di dati

La tabella `parking_area_stats` non serve a nulla, in quanto duplica soltanto dei dati già presenti nella tabella `parking_area`, creando un'inutile ridondanza di dati.

Cardinalità delle relazioni errata

La relazione `sensors -> parking_area` ha cardinalità uno a molti, nel senso che una piazzola può avere più sensori e un sensore può essere associato solo a una piazzola. La cosa è sbagliata, in quanto una piazzola può, sì, avere più sensori associati (un sensore di parcheggio e/o N sensori ambientali) ma anche un sensore può essere associato a più piazzole; in quanto un sensore ambientale può ricoprire un'area di N piazzole.

Mancanza di tabelle fondamentali

Mancava una tabella fondamentale che rappresentasse un parcheggio (un insieme di piazzole).

Tabella molto importante dato che un parcheggio ha un indirizzo e una posizione geografica (latitudine e longitudine), riconosciute come tali dagli strumenti di navigazione più comuni, come Google Maps.

Ogni piazzola di uno stesso parcheggio ha una posizione geografica diversa dalle altre piazzole (discostata di qualche metro) e potenzialmente diversa da quella del parcheggio.

Quindi con la struttura vecchia non era possibile ricercare un parcheggio a sistema passando le coordinate del parcheggio di Google Maps ad esempio e nemmeno identificare un singolo parcheggio.

Database poco modulare

La misurazione del sensore di parcheggio viene salvata all'interno della tabella `parking_area` (libero/occupato). Questa cosa non creava problemi con il corrente modello progettuale dove, dall'analisi fatta, si era deciso di salvare solo l'ultima misurazione di un sensore di parcheggio.

Se però in futuro si decidesse di salvare uno storico di misurazioni dei sensori di parcheggio (cosa molto probabile che avvenga e cosa che viene già fatta con i sensori ambientali), non sarebbe possibile farlo e i costi per modificare la struttura del database con il software in esercizio su un ambiente di produzione, sarebbero molto più alti rispetto a farlo in fase di sviluppo.

Inoltre l'aggiunta di una tabella per lo storico non ha un impatto negativo sulla struttura del database.

4.2 Ristrutturazione

Il modello logico ristrutturato è il seguente:

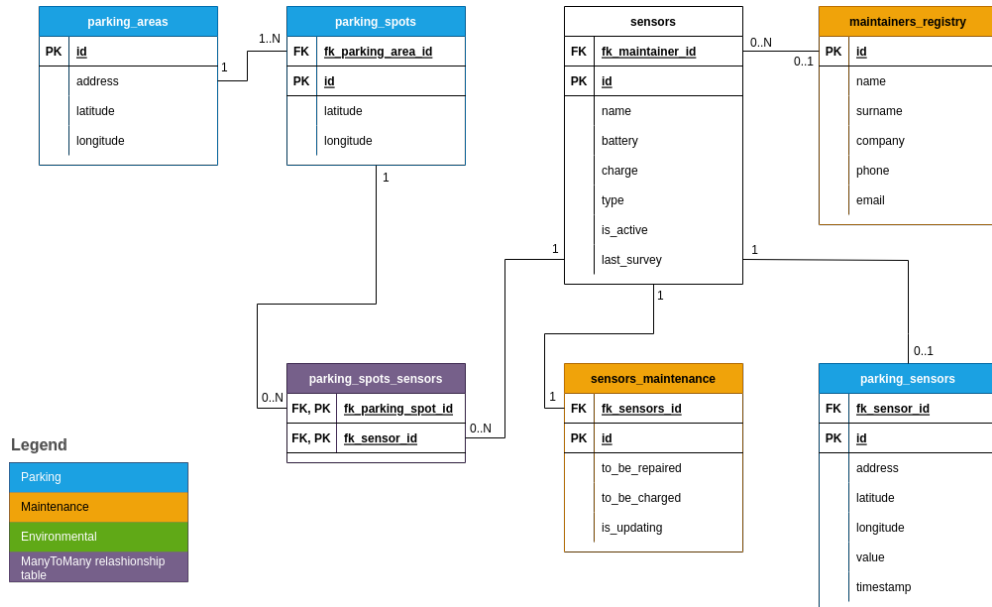


Figura 4.2: Modello logico post-ristrutturazione

Operazioni effettuate:

Nomi tabelle incoerenti

Sono stati modificati i nomi di alcune tabelle per renderle più coerenti alla loro funzionalità:

parking_area è stata modificata in parking_spots.

sensors_maintainer è stata modificata in sensors_maintenance.

Duplicazione di dati

E' stata eliminata la tabella parking_area_stats.

Cardinalità delle relazioni errata

La relazione sensors -> parking_area è diventata una relazione molti a molti.

E' stato reso facoltativo il fatto che il sensore debba essere associato a una piazzola.

E' stato reso facoltativo il fatto che il manutentore debba essere associato a un sensore.

E' stato reso facoltativo il fatto che il sensore debba essere associato a un manutentore.

Mancanza di tabelle fondamentali

E' stata aggiunta la tabella `parking_areas` che rappresenta i parcheggi, dotati di indirizzo, latitudine e longitudine.

Database poco modulare

E' stata creata la tabella `parking_sensors` per salvare la misurazione del sensore di parcheggio ed è stato tolto il campo `value` dalla nuova tabella `parking_spots`; in quanto avendo già la tabella `parking_sensors`, il campo `value` rappresentava una ridondanza inutile.

Per ogni sensore di parcheggio ora viene salvata solo una misurazione nella tabella `parking_sensors`, ovvero l'ultima, che sovrascrive la precedente ma a differenza del modello logico precedente, se in qualsiasi momento si decidesse di salvare lo storico delle misurazioni dei sensori di parcheggio, si può fare semplicemente togliendo il vincolo `unique` dalla foreign key `fk_sensor_id` della tabella `parking_sensors`, senza modificare la struttura della base di dati.

Capitolo 5

Verifica e validazione

In questo capitolo viene descritta la fase di verifica e validazione realizzata durante il progetto.

5.1 Verifica

5.1.1 Criteri di verifica

E' stato stabilito con il proponente di realizzare 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%.

Un metodo si divide in più branch di esecuzione, molti di questi branch sono responsabili della gestione dei casi in cui si verifica un errore e/o i casi in cui il metodo non può seguire il suo comportamento standard ad esempio perché i parametri passati dal client non sono validi.

Il proponente ha definito in maniera chiara e precisa, durante la progettazione del software in Spring, come il software dovesse comportarsi in questi casi ed ha ritenuto di fondamentale importanza che gli sviluppatori seguano queste linee guida.

Questa è la motivazione che ha portato a una richiesta di copertura a livello branch così alta. In questo modo si assicura che, anche in caso di manutenzione del programma, il comportamento nei casi descritti non venga alterato.

Ad esempio se il client richiede le informazioni di uno specifico parcheggio passando un'id inesistente, senza delle linee guida su come comportarsi, possono generarsi delle ambiguità. Uno sviluppatore, ad esempio, potrebbe rispondere con il corpo del messaggio vuoto a indicare che non è stata rilevata alcuna informazione, mentre un'altro potrebbe rispondere con un codice di errore e un messaggio appropriato.

Questo tipo di situazione genera confusione per l'utente finale e rende il programma poco solido. Il proponente ha ritenuto quindi importante garantire coerenza nella gestione di questi casi.

5.1.2 Strumenti utilizzati

NestJS fornisce uno strumento built-in per sviluppare vari tipi di test, tra cui unit test, end to end test, integration test e così via.

Lo strumento usa Jest, un framework per scrivere test automatici ed effettuare il [mock](#) delle componenti. Jest funziona su progetti che includono React, Babel, TypeScript, Node, Angular, Vue.

I file di test 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 ed eseguiti da NestJS.

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

I metodi da testare devono essere racchiusi in un ulteriore *describe*, specificando il nome del metodo e la funzione di callback contenente i test.

Ogni test deve essere racchiuso nella dicitura *it*, specificando la descrizione e la funzione di callback contenente il test.

```
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);
      });
  });
});
```

5.1.3 Progettazione

Per i criteri di verifica stabiliti con il proponente si è deciso di testare tutti i metodi dei service che avessero un numero di branch $>$ di 1. In questo modo viene assicurata la copertura di tutti i casi di errore, dato che i metodi con un solo branch non gestiscono comportamenti inaspettati in maniera custom e in caso di eccezione non la catturano rilanciandola al chiamante.

Scrivendo dei test automatici che vanno a testare i metodi con più di un branch si assicura che:

- * ogni branch può essere raggiunto;
- * ogni branch si comporta nel modo atteso;
- * se uno sviluppatore modifica un metodo, si assicura che i casi di errore gestiti in maniera custom (id non trovato, aggiornamento non riuscito ecc..) non vengano eliminati e siano conformi con quanto stabilito con il proponente.
Ad esempio in caso di errore di aggiornamento di un'entità, si assicura che venga mantenuto il branch di controllo e venga lanciata un eccezione di tipo *UpdateError*.

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.

I [mock](#) sono delle componenti fittizie 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. Tramite il metodo `spyOn` infatti si va a specificare l'oggetto e il metodo di cui si vuole effettuare il [mock](#).

Per evitare di dover caricare componenti reali, quindi molto pesanti ed effettuare successivamente il [mock](#) di tutti i loro metodi, è stata usata una libreria esterna chiamata *@golevelup/ts-jest*, che permette di dichiarare l'intera classe di cui si vuole effettuare il [mock](#) e restituisce un oggetto con gli stessi metodi della classe specificatagli ma con il corpo vuoto, alleggerendo notevolmente il peso della componente.

Viene successivamente fatto il [mock](#) solo dei metodi necessari all'esecuzione dei test tramite il metodo `spyOn`, come descritto in precedenza.

5.1.4 Realizzazione

Come risultato di quanto progettato sono stata creata una suite di 8 test, per un totale di 40 unit test.

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

Tabella 5.1: Copertura degli unit test

5.2 Validazione

Al termine del progetto la copertura dei requisiti è la seguente:

Tipologia	Coperti	Totale	Percentuale
Funzionali	23	23	100%
Qualitativi	3	3	100%
Di vincolo	7	7	100%
Totale	33	33	100%

Tabella 5.2: Copertura dei requisiti

Come si può notare i requisiti funzionali, qualitativi e di vincolo sono stati coperti con una percentuale pari al 100%.

Di conseguenza sono stati coperti anche i requisiti desiderabili oltre a quelli obbligatori. Non erano presenti requisiti facoltativi.

Capitolo 6

Analisi comparativa

In questo capitolo viene descritta la fase di analisi comparativa effettuata al termine della realizzazione del progetto.

Il proponente ha deciso di realizzare questo progetto di stage con lo scopo di effettuare un'analisi comparativa tra la versione del prodotto finale in NestJS e quella in Spring.

Essendo NestJS un framework abbastanza giovane (rilasciato nel 2016), il proponente si è interessato a questa tecnologia nell'ottica di poter sviluppare futuri progetti aziendali con questo framework e sfruttarne i benefici.

Avendo una bassa conoscenza della tecnologia è stato ritenuto troppo rischioso utilizzare NestJS per realizzare un nuovo progetto da zero.

E' stato deciso quindi di migrare un progetto già esistente, nell'ottica che se la soluzione realizzata risultasse fallimentare, è comunque presente un progetto solido che può essere messo in esercizio in ambiente di produzione, riducendo i costi del dover scartare un prodotto inutilizzabile.

Il problema di dover scartare il prodotto realizzato in NestJS può presentarsi nel caso in fase di sviluppo sorgessero problemi di non fattibilità del software, non preventivati in fase di analisi per la scarsa conoscenza del framework. Oppure per problemi legati alla soluzione finale realizzata, per lo stesso motivo di prima.

6.1 Le due soluzioni

Le due soluzioni sviluppano lo stesso prodotto con due framework diversi. Il prodotto realizza un set di [API REST](#), per la gestione delle operazioni [CRUD](#) del progetto Smart Parking.

Il prodotto già esistente è stato realizzato col framework Spring, il nuovo col fra-

mework NestJS:

Spring è un framework che si basa sul linguaggio Java, rilasciato nel 2003. Ci sono molti progetti scritti sopra a Spring, come Spring Boot. Spring Boot è il framework che è stato utilizzato per realizzare il vecchio progetto.

Spring Boot si occupa di gestire e fornire tutte le librerie di Spring o di terze parti in base alla configurazione di cui abbiamo bisogno, quindi utilizzare questo strumento velocizza notevolmente il processo di sviluppo di un'applicazione.

NestJS è un framework basato sul linguaggio JavaScript, in particolare sul linguaggio TypeScript. NestJS è stato rilasciato nel 2016.

Attualmente non sono molti i prodotti realizzati in NestJS, ma questo framework sta prendendo piede negli ultimi anni in quanto usa Node.js per eseguire il codice JavaScript.

Node.js è uno strumento molto utilizzato, nonostante sia fornito di una vastità di librerie la sua lacuna è la mancanza di un'architettura ben strutturata, che invece è ben presente in Spring. Quest'architettura permette la realizzazione di applicazioni altamente testabili, scalabili, con basso grado di accoppiamento e facili da mantenere.

NestJS copre questa mancanza di Node.js, fornendo un livello in più di astrazione sopra a Node.js e creando un'architettura molto simile a quella di Spring.

6.2 Punti di valutazione

Per effettuare l'analisi comparativa tra le due soluzioni, sono stati presi in considerazione i seguenti punti e messi a confronto:

- * facilità di sviluppo;
- * strumenti di supporto allo sviluppo (documentazione, community, video, ecc..);
- * facilità di accesso agli strumenti di supporto;
- * prestazioni;
- * qualità del codice.

6.3 Valutazione

6.3.1 Facilità di sviluppo

Spring

Spring Boot si basa sul concetto di Dependency Injection e usa il pattern controller-service-repository. Questa struttura permette uno sviluppo rapido delle applicazioni, dato che si occupa il framework di iniettare le dipendenze dichiarate nei costruttori dei componenti. Il pattern controller-service-repository permette di creare applicazioni ben strutturate e dotate di una buona separazione delle responsabilità, perseguendo i

principi SOLID.

NestJS

Anche NestJS si basa sul concetto di Dependency Injection e usa il pattern controller-service-repository, per cui ne trae gli stessi benefici di facilità di sviluppo di Spring.

6.3.2 Strumenti di supporto allo sviluppo

Spring

Nel sito ufficiale Spring offre una documentazione molto ricca per imparare in dettaglio il framework.

Essendo nato 19 anni fa, negli anni, si è formata una community di supporto molto vasta.

In questo ventennio sono state pubblicate domande di vario genere alla community, tra cui quelle relative ai problemi più comuni che si presentano agli sviluppatori che affrontano il framework per la prima volta.

Le risposte degli sviluppatori senior sono molte, ben accurate e scritte in modo chiaro. Le discussioni sono reperibili nei blog online dedicati o in alcuni siti come Stack Overflow.

Ci sono anche molti video tutorial, reperibili nelle piattaforme più conosciute come il sito Udemy.

NestJS

Nel sito ufficiale NestJS offre una documentazione molto ricca per imparare in dettaglio il framework.

Questo framework è relativamente giovane, nato 6 anni fa. Di conseguenza le aziende e gli sviluppatori che hanno deciso di adottarlo non sono molti rispetto a quelli che usano Spring.

La community è molto piccola.

Le discussioni sui problemi che si presentano agli sviluppatori che affrontano il framework per la prima volta o sui problemi di carattere generale che si possono presentare sviluppando con questo framework sono molto poche.

Molto spesso se si cerca la soluzione a un problema, non si trova una discussione a riguardo oppure si trova la discussione aperta da uno sviluppatore ma priva di risposte.

A volte invece si viene a scoprire, tramite la discussione ufficiale nel repository GitHub di NestJS, che il problema avuto è causato da un bug di NestJS ancora in fase di fix e previsto in rilascio per le versioni successive.

I video tutorial sono molto pochi e non sufficienti per imparare in dettaglio il framework.

6.3.3 Facilità di accesso agli strumenti di supporto

Spring

La documentazione di Spring è molto pedante e ricca. Imparare l'uso del framework solo dalla documentazione ufficiale richiede molto tempo.

Leggendo interamente la documentazione si diventa dei veri esperti ma con le ridotte tempistiche dello stage non è possibile effettuare questa cosa.

Per iniziare è bene integrare la documentazione ai video tutorial e al materiale online, comprese le discussioni della community.

In questo modo si può essere operativi in tempi più brevi, assicurando di avere una conoscenza del framework alle spalle sufficiente larga da poter fare le cose in modo coerente.

NestJS

Lo strumento principale da usare per imparare NestJS è la documentazione ufficiale.

La documentazione è fatta molto bene e non è eccessivamente ricca come quella di Spring.

Spiega in maniera dettagliata e in modo molto chiaro tutti i concetti di cui si ha bisogno per sviluppare con questo framework.

Ho usato principalmente la documentazione ufficiale per imparare NestJS durante lo stage e sono riuscito a essere operativo in tempi ragionevoli e a fare le cose in modo coerente.

6.3.4 Prestazioni

Per valutare le due soluzioni in termini di prestazioni è stato considerato il tempo medio di risposta delle [API REST](#), caricate sullo stesso server ed eseguite a parità di condizioni.

Per tutte le [API REST](#) la differenza di risposta tra la soluzione in Spring e in NestJS è troppo piccola per rappresentare un punto di valutazione significativo (dell'ordine dei millisecondi).

Le prestazioni a livello di carico non sono state prese in considerazione, dato che per il momento non si prevede che l'applicazione venga utilizzata in modo intensivo.

6.3.5 Qualità del codice

Spring

Spring permette di usare degli zuccheri sintattici che nascondono la complessità del codice e lo rendono più pulito.

Un'esempio sono le annotazioni, che permettono di definire delle proprietà della classe in modo esplicito, chiaro e pulito.

L'annotazione `@Controller` sopra la definizione della classe, permette di dichiararla

come componente Controller.

Inoltre già l'architettura interna del framework aiuta a scrivere codice pulito, in quanto la separazione delle componenti per livello di responsabilità permette di scrivere metodi con poche righe di codice, chiari e facili da mantenere.

NestJS

NestJS è dotato degli stessi zuccheri sintattici di Spring per nascondere la complessità del codice e renderlo pulito.

La differenza è che su NestJS le annotazioni si chiamano decorator, ma a livello implementativo hanno lo stesso significato.

Dato che l'architettura di NestJS è la stessa usata da Spring, anche su NestJS si riflettono gli stessi vantaggi di qualità del codice dovuti all'architettura.

Inoltre l'uso dei moduli in NestJS permette di organizzare meglio la struttura dei file rispetto a Spring.

TypeOrm, lo strumento ORM di NestJS per interfacciarsi con il database, permette di scrivere query leggermente più pulite rispetto ad Hibernate, lo strumento ORM di Spring.

6.4 Esito

Da quanto emerso dall'analisi comparativa risulta più semplice creare un software in Spring, per via dell'elevato materiale di supporto al framework e per la presenza di una community molto attiva e professionale, che negli anni ha risolto problemi comuni agli sviluppatori che hanno usato il framework.

D'altra parte NestJS risulta povero di materiale di supporto, infatti oltre alla documentazione ufficiale (fatta molto bene), ne è praticamente sprovvisto e anche la community si può considerare inesistente.

Molti dei problemi in NestJS che si presentano agli sviluppatori che affrontano il framework per la prima volta non trovano soluzione in una community online e questo è fonte di elevato costo in termini di tempo per lo sviluppatore inesperto che deve cercare di risolvere il problema da solo.

Il fatto di dover risolvere in maniera autonoma il problema, oltre che a portare un costo in termini di tempo, non assicura che la soluzione trovata sia una best practice; creando potenziali inconsistenze nel codice.

A livello di prestazioni non sono state rilevate sostanziali differenze e nemmeno la possibilità di scrivere query leggermente più pulite con TypeOrm rispetto ad Hibernate è stato un punto sufficiente per favorire NestJS rispetto a Spring.

Il proponente ha considerato importante il punto riguardo la facilità di sviluppo

del codice e la facilità/disponibilità di accesso agli strumenti di supporto.

Il costo in termini di tempo per lo sviluppo di un progetto in NestJS è ancora troppo elevato rispetto a Spring, proprio per la difficoltà nel reperire materiale di supporto. Per questo motivo l'analisi comparativa ha portato a preferire l'uso del software scritto in Spring come [back-end](#) del progetto Smart Parking.

La realizzazione del progetto in NestJS è stata comunque molto utile al proponente, che ha maturato competenze riguardo a un nuovo framework; competenza utile in un futuro prossimo se NestJS prenderà piede e il materiale a disposizione e la community cresceranno, rendendo il framework competitivo a livello di Spring.

L'azienda inoltre ha maturato una risorsa interna (me stesso) nel framework NestJS spendibile per progetti futuri.

Capitolo 7

Conclusioni

In questo capitolo vengono tratte le conclusioni finali sulla realizzazione del progetto.

7.1 Analisi del prodotto ottenuto

Il prodotto ottenuto, pronto per il rilascio in ambiente di produzione, come spiegato nell'analisi comparativa, non verrà usato come strumento di produzione.

Questo prodotto verrà comunque conservato dal proponente nel proprio repository GitHub, come strumento da cui prendere spunto per un eventuale progetto futuro in NestJS.

Il progetto realizzato ha seguito tutte le buone pratiche di sviluppo suggerite nella documentazione ufficiale NestJS e vari design pattern.

Come richiesto dalla proponente è stata implementata una suite di unit test automatici per testare tutti i servizi a livello branch.

Il prodotto realizzato è molto facile da avviare, infatti qualsiasi sviluppatore che non conosce NestJS può andare nel repository GitHub del progetto e leggere il file README, che spiega in modo dettagliato come scaricare le dipendenze richieste tramite npm e far partire il servizio.

7.2 Raggiungimento degli obiettivi

Vediamo gli obiettivi pianificati e il loro stato di raggiungimento al termine dello stage:

Notazione utilizzata:

- * O per gli obiettivi obbligatori, vincolanti in quanto obiettivo primario richiesto dal proponente;
- * D per gli obiettivi desiderabili, non vincolanti o strettamente necessari ma dal riconoscibile valore aggiunto;

* F per i vincoli facoltativi, rappresentante valore aggiunto non strettamente competitivo.

Codice	Descrizione	Stato
O01	Acquisizione competenze Java	Soddisfatto
O02	Acquisizione competenze passaggio da monolite a microservizi	Soddisfatto
O03	Acquisizione competenze framework Spring Boot	Soddisfatto
O04	Acquisizione competenze strumento Spring Data JPA	Soddisfatto
O05	Acquisizione competenze realizzazione servizio di API REST	Soddisfatto
O06	Acquisizione competenze strumento Node.js	Soddisfatto
O07	Acquisizione competenze framework Express.js	Soddisfatto
O08	Realizzazione di un servizio REST prototipale con Spring Boot	Soddisfatto
O09	Realizzazione di un servizio REST prototipale con Express.js	Soddisfatto
O10	Realizzazione di un'analisi per il progetto Smart Parking	Soddisfatto
O11	Realizzazione di una progettazione per il progetto Smart Parking	Soddisfatto
O12	Realizzazione di una suite di test automatici con copertura a livello branch $\geq 90\%$	Soddisfatto
O13	Realizzazione di una suite di test automatici con copertura a livello linee di codice $\geq 60\%$	Soddisfatto
O14	Realizzazione di un numero di API REST progettate maggiore dell'80%	Soddisfatto
O15	Realizzazione di un'analisi comparativa tra la soluzione in Spring e in NestJS	Soddisfatto
O16	Raggiungimento degli obiettivi richiesti in autonomia seguendo il cronoprogramma	Soddisfatto

Tabella 7.1: Stato raggiungimento obiettivi obbligatori

Codice	Descrizione	Stato
D01	Realizzazione di un numero di API REST progettate pari al 100%	Soddisfatto

Tabella 7.2: Stato raggiungimento obiettivi desiderabili

Codice	Descrizione	Stato
F01	Analizzazione di come poter implementare le best practice dell'architettura a microservizi con NestJS	Soddisfatto parzialmente

Tabella 7.3: Stato raggiungimento obiettivi facoltativi

7.3 Valutazione personale

L'esperienza di stage è stata molto positiva. La prima parte dello stage è stata quella più difficile da affrontare, in quanto ho dovuto imparare da zero delle tecnologie che non avevo mai visto come Spring e NestJS.

Questo processo di apprendimento è stato per me molto utile, sia perché la conoscenza di questi due framework mi sarà molto utile in ambito lavorativo, sia perché mi ha insegnato come apprendere il funzionamento di un framework in maniera abbastanza dettagliata ed essere operativo in tempo breve.

Ho potuto affacciarmi in un'azienda con molti dipendenti e quindi confrontarmi giornalmente con sviluppatori più esperti di me che mi hanno aiutato molto.

Sono soddisfatto del prodotto realizzato e di come si sono svolti i processi di sviluppo che hanno portato alla sua realizzazione.

Ritengo molto valida questo tipo di esperienza in azienda, anche per tutti i miei colleghi universitari futuri, in quanto mi ha fatto crescere molto come sviluppatore. Anche se due mesi sono pochi ho potuto mettere mano a un progetto reale e sono molto cresciuto anche nella fase di analisi e progettazione.

Sicuramente il background fornito dall'università è stato fondamentale per poter svolgere questo stage; in quanto l'università mi ha fornito tutti i concetti basilari di cui avevo bisogno per poter apprendere le nuove tecnologie e per poterle utilizzare in maniera corretta.

L'università mi ha dato gli strumenti per poter analizzare con occhio critico le scelte progettuali che sono andato a svolgere.

In conclusione sono pienamente soddisfatto dello stage che ho fatto e del prodotto che ho realizzato che, anche se non verrà utilizzato in produzione, ha posto comunque le basi per un futuro progetto realizzabile in NestJS e ha reso appetibile questa nuova tecnologia per i progetti futuri del proponente.

Bibliografia