Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

API REST in Spring Java con architettura a microservizi per un sistema di parcheggio smart

Tesi di laurea triennale

Relatore

Prof.Claudio Enrico Palazzi

Laureando

Luca Busacca

Anno Accademico 2022-2023

Computer science inverts the normal. In normal science, you're given a world, and your job is to find out the rules. In computer science, you give the computer the rules, and it creates the world.

— Allan Kay

Dedicato a mamma e papà, per aver creduto in me in ogni singolo momento e per il sostegno, il grande aiuto e la fiducia riposta in me in ogni occasione.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecentoventi ore, dal laureando Luca Busacca presso l'azienda Sync Lab S.r.l. nel periodo che va dal 04/07/2022 al 02/09/2022.

Lo scopo dello stage era la realizzazione di un backend di un software capace di gestire l'informazione rilevata da alcuni sensori, installati in postazioni di parcheggio, che rilevano la presenza di un veicolo, per poi riuscire a disegnare sulla mappa i parcheggi liberi in "real time" per mezzo di un front end che verrà successivamente sviluppato da un altro stagista.

In questo documento viene descritta l'organizzazione del suddetto stage, la fase di analisi del problema e la conseguente progettazione.
In particolare viene discussa la progettazione delle API REST, la loro realizzazione tramite il framework Spring e l'interazione con esse per mezzo di Postman.

Ringraziamenti

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

Ho desiderio di ringraziare poi l'Ing. Zorzi Daniele, tutor aziendale, per la grande disponibilità dimostrata durante il periodo di stage.

Desidero infine ringraziare con affetto la mia famiglia, la mia ragazza e i miei amici per essermi stati vicini in ogni momento durante gli anni di studio.

Padova, Aprile 2023 Luca Busacca

Indice

Elenco delle figure

Elenco delle tabelle

Premessa

Convenzioni

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*$_{\text{[G]}}$.

Organizzazione del testo

Il capitolo 1 presenta un'introduzione al progetto

Il capitolo 2 presenta gli obbiettivi che lo stage si poneva di raggiungere.

Il capitolo 3 approfondisce la fase di raccolta e analisi dei requisiti.

Il capitolo 4 affronta la fase di progettazione e codifica.

Il capitolo 5 descrive la fase di verifica e validazione del prodotto ottenuto.

Il capitolo 6 trae le conclusioni finali riguardo l'esperienza di stage.

Capitolo 1

Introduzione

In questo capitolo viene brevemente fornita un'introduzione al progetto e alla soluzione realizzata.

1.1 L'azienda

Sync Lab è un'azienda che fornisce da oltre vent'anni soluzioni IT[G] per diversi mercati quali: sanità, industria, energia, telecomunicazioni, finanza, trasporti e logistica.

Figura 1: Logo Sync Lab

Nasce a Napoli nel 2002 ed è continuamente cresciuta da allora: attualmente ha sei sedi in tutta Italia e oltre trecento dipendenti.
Sync Lab ha una lunga storia di collaborazione con diverse università a partire già dal 2003.
L'ambiente di lavoro dell'azienda trasmette armonia, gli strumenti offerti si sono rivelati utili allo svolgimento dello stage. Il tutor aziendale si è dimostrato totalmente disponibile per qualunque necessità presentatasi, facendo in modo che queste fossero comunque risolte in maniera autonoma.

Sito Sync Lab. URL: https://www.synclab.it

1.2 Introduzione al progetto

Lo scopo dello stage è quello di implementare il backend[G] di un software capace di gestire l'informazione rilevata da alcuni sensori, installati in postazioni di parcheggio, che rilevano la presenza di un veicolo, per poi riuscire ad indicare sulla mappa i

parcheggi liberi e quelli occupati avvicinandosi ad una risposta in Real time.[1]

La soluzione software pensata da Sync Lab, oltre al backend, prevede la realizzazione da parte di altri stagisti del frontend[G] al quale saranno esposti gli endpoint[G] prodotti per l'accesso al software.

La realizzazione di tale software prevede l'implementazione tramite architettura a microservizi[G] ed esposizione di API[G] REST[G] per operazioni CRUD[G] sulla base di dati[G] contenente in maniera persistente i dati rilevati dai sensori sotto forma di file XML[G].

La soluzione descritta fa parte di un progetto più ampio, nominato dall'azienda **Smart City Simulator**, volto a realizzare un microservizio per ciascun tipo di sensore, includendone varie tipologie, quali ad esempio sensori di temperatura, di misurazione delle polveri sottili nell'aria, galleggianti per la misurazione della qualità dell'acqua, etc.

Il fine di tale progetto è quello di rendere smart una città migliorando la qualità di vita all'interno della stessa, tenendo sotto controllo gli effetti negativi quali il caldo elevato o la quantità di inquinamento nell'aria, ma badando anche a funzioni che tendono a ridurre l'inquinamento.

Si può infatti immaginare come il sensore di parcheggio possa evitare all'utente di continuare a percorrere chilometri in auto cercando un parcheggio. La diffusione dei sensori di parcheggio nelle città permetterebbe alle persone di cercare una zona in cui parcheggiare che sia vicina al luogo in cui dovrà recarsi, ancora prima di partire, evitando emissioni superflue di sostanze inquinanti nell'aria.

Per lo svolgimento dello stage è stata considerata l'informazione generata da sensori installati in postazioni di parcheggio che comunicano con un rilevatore di informazione che a sua volta produce il file in formato XML dal quale vengono poi estratti i dati da salvare nel DB.
Vengono riportati nel seguito un esempio dei dati prelevati dai sensori mediante dei rilevatori installati nelle postazioni e un esempio del file prodotto dal rilevatore e letto dall'applicativo:

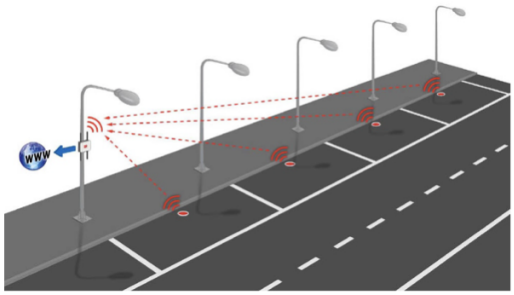


Figura 2: Esempio di funzionamento del sensore di parcheggio

[1] I dati vengono aggiornati ogni 2 minuti in quanto, dopo averne parlato con gli stakeholders[G], questo è stato ritenuto un tempo sufficiente per un ritardo di aggiornamento rispetto al caso d'uso dell'applicativo.

Figura 3: Esempio di file prodotto dai rilevatori

1.3 Soluzione proposta

L'architettura individuata per il progetto è basata sui microservizi il che permette una scalabilità orizzontale. Una parte, il frontend, si occupa di gestire l'interazione con l'utente fornendo un'interfaccia grafica per, ad esempio, visualizzare i parcheggi liberi sulla mappa. La comunicazione con il backend, invece, avviene tramite chiamate REST e questo permette di ridurre la dipendenza tra frontend e backend a una mera interfaccia.

Per la realizzazione del backend, in particolare, si è scelto di utilizzare l'architettura proposta e commentata di seguito:

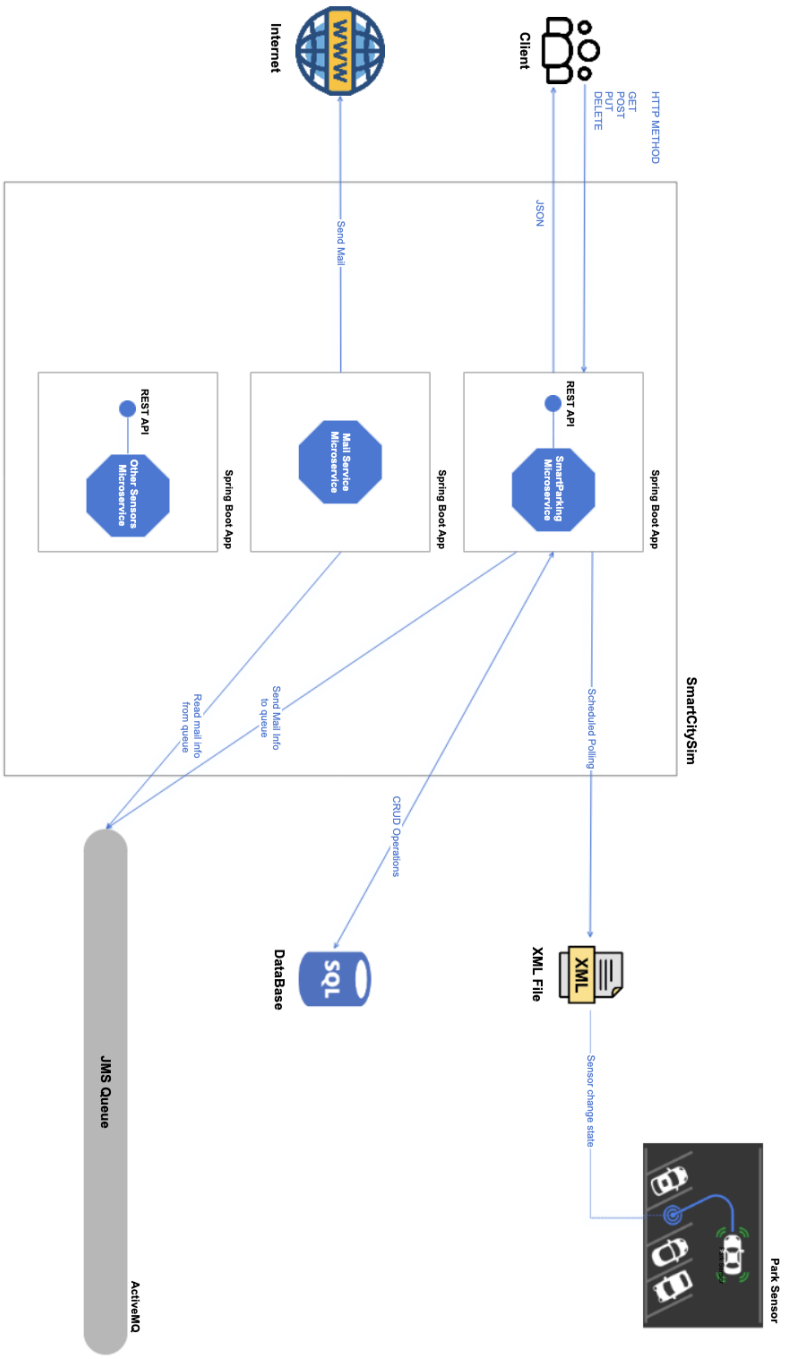


Figura 4: Architettura utilizzata per lo sviluppo del backend di Smart Parking

Dalla figura è facile notare che il client[G] comunica con il microservizio, che espone l'informazione riguardante i sensori, attraverso degli endpoint ai quali, lo stesso client, invia richieste HTTP[G] e riceve in risposta i dati in formato JSON[G].
Lo stesso microservizio comunica poi con il file prodotto dai sensori attraverso una funzione di polling che viene richiamata ogni 2 minuti ed è in grado di effettuare operazioni di tipo CRUD sul DB

Oltre al microservizio per i sensori di parcheggio, l'architettura proposta presenta ancora altri due microservizi:

- **Mail Service Microservice**: si occupa della gestione delle mail degli altri microservizi. In particolare questo legge i dati da una coda JMS[G] alla quale vengono inviati i dati dagli altri microservizi.

- **Other Sensors Microservice**: rappresenta un microservizio generico di sensori che guarda alle estensioni future del progetto in cui verranno implementati, come precedentemente descritto, i microservizi di altre tipologie di sensori.

Si è scelto di realizzare il backend tramite il linguaggio Java[G] e l'utilizzo di framework[G] quali Spring Boot e Spring Data JPA.

La comunicazione tra il microservizio dei sensori di parcheggio e quello della gestione delle mail avviene attraverso una coda JMS implementata con ActiveMQ.

Per quanto riguarda la base di dati, si è scelto di utilizzarne una di tipo PostgreSQL come default, ma è anche possibile utilizzarne una di tipo MySQL in quanto tutte le integrazioni necessarie al funzionamento del software con questo tipo di DB sono pronte.

La scelta di utilizzare tali strumenti è data dal fatto che più persone all'interno dell'azienda li utilizzano quotidianamente, risultando dunque strumenti attuali e di utile apprendimento per il mondo del lavoro e facilitando la comprensione di eventuali dubbi, oltre al fatto che l'interazione tra gli stessi è fornita dallo stesso framework Spring in maniera semplice.

1.4 Strumenti utilizzati

Eclipse

Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma.

ESLint

ESLint è uno strumento di analisi del codice statico per identificare i modelli problematici trovati nel codice.

Git

Git è un Version Control System open source di tipo distribuito.

GitHub

GitHub è un servizio di hosting per progetti software, di proprietà della società GitHub Inc.

JaCoCo

JaCoCo è un'utility per la generazione di report di code coverage.

JUnit

Junit è un framework per la realizzazione dei test di unità.

LaTeX

LaTeXè un linguaggio di marcatura per la preparazione di testi, basato sul programma di composizione tipografica TeX

Maven

Maven è uno strumento di gestione di progetti software basati su Java e build automation, rilasciato in maniera open source da Apache.

MySQL

MySQL è un DBMS relazionale composto da un client a riga di comando e un server, rilasciato in maniera open source da Oracle.

MySql Workbrench

MySql Workbrench è un'interfaccia grafica che consente di amministrare in modo semplificato database di MySQL, rilasciato in maniera open source da Oracle.

PgAdmin4

PgAdmin4 è un'interfaccia grafica che consente di amministrare in modo semplificato database di PostgreSQL, rilasciato in maniera open source da PostgreSQL Global Development Group.

PostgreSQL

PostgreSQL è un DBMS ad oggetti, rilasciato in maniera open source da PostgreSQL Global Development Group.

Postman

Postman è una piattaforma per progettare, creare e testare API.

StarUML

StarUML è uno strumento di ingegneria del software per la modellazione di sistemi utilizzando il linguaggio UML.

Capitolo 2

Descrizione dello stage

In questo capitolo viene descritta la Pianificazione e l'organizzazione dello stage

2.1 Obiettivi

Notazione

Si farà riferimento ai requisiti secondo le seguenti notazioni:

- *O* per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
- *D* per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
- *F* per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.

Le sigle precedentemente indicate saranno seguite da una coppia sequenziale di numeri, identificativo del requisito.

Obiettivi fissati

Si prevede lo svolgimento dei seguenti obiettivi:

- Obbligatori
  - *O-01*: Acquisizione competenze Spring Java, microservizi, REST;
  - *O-02*: Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma;
  - *O-03*: Portare a termine l'implementazione dei microservizi richiesti con una percentuale di superamento pari a 80.
- Desiderabili
  - *D-01*: Portare a termine l'implementazione dei microservizi richiesti con una percentuale di superamento pari a 100.

- Facoltativi
  - *F-01*: Utilizzo della containerizzazione per portare tutti i microservizi su Docker[G].

2.2 Analisi preventiva dei rischi

Al fine di comprendere al meglio le funzionalità da sviluppare è stata effettuata un'analisi dei rischi che si possono riscontrare nell'ambito del progetto.
A ciascun rischio individuato viene assegnato un numero compreso tra 1 e 5 che ne definisce l'entità del rischio come descritto di seguito:

- **1**: il rischio è di bassa entità e non influisce in alcun modo sullo svolgimento del progetto;
- **2**: il rischio è di medio-bassa entità e può influenzare in maniera minima lo svolgimento del progetto;
- **3**: il rischio è di media entità e può influenzare in maniera moderata lo svolgimento del progetto;
- **4**: il rischio è di medio-alta entità e può influenzare in maniera significativa lo svolgimento del progetto;
- **5**: il rischio è di alta entità e può influenzare in maniera elevata lo svolgimento del progetto.

Di seguito è riportata una tabella riassuntiva dei rischi individuati e delle relative azioni di mitigazione:

Rischio	Descrizione	Grado	Contromisura
Lettura XML difficoltosa	Creare un oggetto Java a partire da un file XML può risultare difficoltoso oltre ad avere un gran costo. Tramite l'uso del framework Spring risulta più facile creare un oggetto a partire da un file JSON	1	funzione di conversione dal formato XML al formato JSON
Affidabilità stato sensore	Possibile perdita di dati a causa di malfunzionamenti durante la lettura dei dati del sensore	2	funzione di polling temporizzata
Mancato aggiornamento sensore	Uno o più record del sensore nel file sorgente potrebbero non aggiornarsi per lungo tempo a causa di un guasto	5	Aggiunta campo dati `lastSurvey` di tipo `Timestamp` che si aggiorna ogni qual volta un qualsiasi dato del sensore viene aggiornato

Rischio	Descrizione	Grado	Contromisura
Sensore scarico	La batteria del sensore ha un livello basso o esaurito	4	Aggiunta campo dati `charge` di tipo `String` che segna la parte intera di Volt rimanenti per la batteria del sensore
Sensore guasto o manomesso	Il sensore è guasto o è stato manomesso e non invia più aggiornamenti	4	aggiunto campo dati `isActive` di tipo `boolean` che segnala lo stato di attività del sensore

Tabella 2.1: Rischi individuati

2.3 Pianificazione

Lo stage è stato strutturato in due parti della durata di un mese ciascuna:

- nella prima parte viene effettuata l'analisi del problema, la progettazione e lo studio delle tecnologie;
- nella seconda parte viene realizzata parte della soluzione individuata.

Cronoprogramma settimanale

- **Prima Settimana - Formazione base (40 ore)**
  - Incontro con persone coinvolte nel progetto per discutere i requisiti e le richieste relativamente al sistema da sviluppare;
  - Verifica credenziali e strumenti di lavoro assegnati;
  - Ripasso Java Standard Edition e tool di sviluppo (IDE ecc.);
  - Studio teorico dell'architettura a microservizi: passaggio da monolite ad architetture a microservizi con pro e contro;
  - Ripasso principi della buona programmazione (SOLID, CleanCode);
- **Seconda Settimana - Formazione Microservizi, Spring Core/Spring Boot e ORM (40 ore)**
  - Studio teorico dell'architettura a microservizi: Api Gateway, Service Discovery e Service Registry, Circuit Breaker e Saga Pattern;
  - Studio Spring Core/Spring Boot;
  - Studio ORM[G], in particolare il framework Spring Data JPA.
- **Terza Settimana - Formazione REST, broker e sistemi producer-consumer (40 ore)**
  - Studio servizi REST;

  - Studio dei broker[G] e sistemi producer-consumer.

- **Quarta Settimana - Analisi e prima implementazione modulo di raccolta dati dai sensori (40 ore)**
  - Analisi dell'idea architetturale attuale di SmartParking;
  - Inizio implementazione del modulo di raccolta dati dai sensori.
- **Quinta Settimana - Implementazione modulo di raccolta dati dai sensori (40 ore)**
  - Completamento del modulo di raccolta dati dai sensori.
- **Sesta Settimana - Implementazione servizi di esposizione al Frontend (40 ore)**
  - Inizio implementazione dei servizi per l'esposizione dei dati al frontend.
- **Settima Settimana - Implementazione servizi di esposizione al Frontend (40 ore)**
  - Conclusione implementazioni.
- **Ottava Settimana - Test e conclusione (40 ore)**
  - Test di integrazione finali.

2.4 Organizzazione dello stage

L'azienda permette ai propri dipendenti di lavorare da remoto. Data l'importanza del lavoro in azienda il tutor ha richiesto la presenza in sede per almeno un giorno a settimana. Si è comunque scelto di lavorare in presenza ogni qual volta fosse presente il tutor, con una frequenza minima di due giorni a settimana.

Smart working

L'azienda ha messo a disposizione il server Discord[G] aziendale, il che ha permesso di rimanere sempre aggiornati sul lavoro svolto e di comunicare in maniera efficiente con gli stakeholders.
Sullo stesso server sono presenti dipendenti delle varie sedi Sync Lab in Italia, con i quali era possibile comunicare per qualunque dubbio o problema riscontrato.

Lavoro in sede

Almeno una volta a settimana si sono svolti incontri in sede per aggiornare gli stakeholders sullo stato di avanzamento del progetto e meglio coordinare le attività da svolgere nel periodo prossimo.
In ufficio i colleghi si sono sempre rivelati disponibili a condividere la loro esperienza lavorativa in un ambiente cordiale e amichevole.

Capitolo 3

Analisi dei requisiti

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

3.1 Confronto con gli stakeholders

Per prima cosa, al fine di aver chiari i requisiti, è stata effettuata una discussione con il proponente, ovvero l'azienda Sync Lab.

Chiariti gli obiettivi di massima ho iniziato a definire meglio gli obiettivi e i bisogni che la soluzione avrebbe dovuto coprire.
Ho dunque presentato la mia proposta al tutor aziendale che si è detto soddisfatto e mi ha autorizzato a procedere.

3.2 Entità

Al fine di tener meglio traccia dei dati necessari alla soluzione si è reso necessario documentare le entità di dominio. Il risultato di questa operazione è stato il seguente:

Sensore

Un `Sensore` è una generalizzazione di un qualunque tipo di sensore, definito come segue:

1. **Id**: un numero univoco identificativo del sensore.;
2. **Nome**: nome del sensore;
3. **Batteria**: stringa che indica la quantità di carica residua della batteria del sensore;
4. **Carica Residua**: un numero estratto dalla stringa **Batteria** che serve a tenere traccia dei sensori scarichi;
5. **Tipo**: tipo del sensore;

6. **Attivo**: valore booleano per tenere traccia dello stato di attività (acceso/spento) del sensore;
7. **Ultimo aggiornamento ricevuto**: campo di tipo data-ora che tiene traccia della data e dell'ora dell'ultimo aggiornamento, qualsiasi valore del sensore questo riguardi.

Postazione di parcheggio

Una `Postazione di parcheggio` è una vera e propria tipologia di sensore, definito come segue:

1. **Id**: un numero univoco identificativo della postazione di parcheggio;
2. **Id sensore**: riferimento al sensore generico collegato alla postazione di parcheggio;
3. **Latitudine**: stringa che indica il valore della latitudine della postazione di parcheggio;
4. **Longitudine**: stringa che indica il valore della longitudine della postazione di parcheggio;
5. **Indirizzo**: Indirizzo della postazione di Parcheggio;
6. **Stato**: valore booleano per tenere traccia dello stato (libero/occupato) del sensore;
7. **Ultimo aggiornamento**: campo di tipo data-ora che tiene traccia della data e dell'ora di aggiornamento dello stato del sensore.

Manutentore

Un `Manutentore` è un riferimento a coloro i quali si occupano della manutenzione del sensore, definito come segue:

1. **Id**: un numero univoco identificativo del manutentore;
2. **Id sensore**: riferimento al sensore generico da mantenere;
3. **Nome**: nome del manutentore;
4. **Cognome**: cognome del manutentore;
5. **Compagnia**: compagnia del manutentore;
6. **Telefono**: numero di telefono del manutentore;
7. **Mail**: mail del manutentore;
8. **Tipo**: tipo del sensore;
9. **Sensore spento**: valore booleano per tenere traccia dello stato (acceso/spento) del sensore;
10. **Sensore scarico**: valore booleano per tenere traccia dello stato (carico/scarico) del sensore;
11. **Sensore corrotto**: valore booleano per tenere traccia dello stato (funzionante/-corrotto) del sensore.

Statistiche di parcheggio

Una `Statistica di parcheggio` è una raccolta di informazioni relative a una postazione di parcheggio che permettono successivamente di definire delle statistiche di occupazione del sensore, definita come segue:

1. **Id**: un numero univoco identificativo della statistica di parcheggio;
2. **Id sensore**: riferimento al sensore generico collegato alla postazione di parcheggio;
3. **Data**: campo di tipo data-ora che tiene traccia della data e dell'ora di aggiornamento della statistica di parcheggio;
4. **Stato**: valore booleano per tenere traccia dello stato (libero/occupato) del sensore;

3.3 Definizione casi d'uso

Non appena le entità necessarie sono state definite, si è proceduto alla definizione dei casi d'uso.
Viene riportata di seguito l'ultima versione individuata dei casi d'uso suddivisa per macro categorie definite dalle entità precedentemente descritte.

È importante sottolineare che, poiché si tratta di API REST, l'unico attore individuato è un client generico in grado di effettuare richieste HTTP al servizio.

3.3.1 UC1 Operazioni CRUD sensore

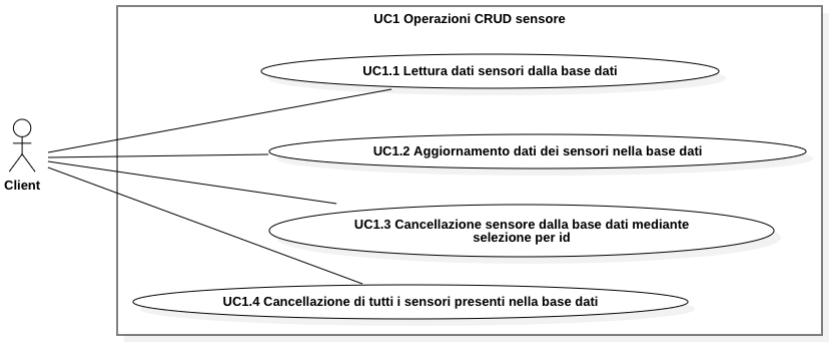


Figura 5: UC1 - Lettura e scrittura dati dal file risorsa

UC1.1: Lettura dati sensore dalla base dati

Attori Principali: Client.

Precondizioni: Esiste almeno un sensore salvato nella base dati.

Postcondizioni: I dati dei sensori sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati dei sensori salvati nella base di dati.

UC1.2: Aggiornamento dati dei sensori nella base dati

Attori Principali: Client.

Precondizioni: I sensori che si vogliono aggiornare sono presenti nella base dati.

Postcondizioni: I dati dei sensori riferiti sono stati aggiornati nella base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare i dati di uno o più sensori.

UC1.3: Cancellazione sensore dalla base dati mediante selezione per id

Attori Principali: Client.

Precondizioni: Un sensore con l'id specificato è presente nella base dati.

Postcondizioni: Il sensore con l'id specificato è stato eliminato dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare un sensore dalla base dati mediante la selezione per mezzo dell'id.

UC1.4: Cancellazione di tutti i sensori presenti nella base di dati

Attori Principali: Client.

Precondizioni: Esiste almeno un sensore nella base dati.

Postcondizioni: Tutti i sensori presenti nella base dati sono stati eliminati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare tutti i sensori presenti nella base dati.

UC1.1 Lettura dati sensori dalla base dati

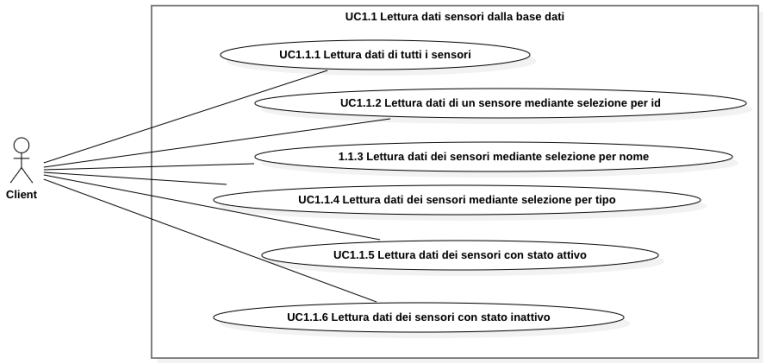


Figura 6: UC1.1 - Lettura dati sensori dalla base dati

UC1.1.1: Lettura dati di tutti i sensori

Attori Principali: Client.

Precondizioni: Esiste almeno un sensore nella base dati.

Postcondizioni: Tutti i dati di tutti i sensori sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori presenti nella base di dati.

UC1.1.2: Lettura dati di un sensore mediante selezione per id

Attori Principali: Client.

Precondizioni: Un sensore con l'id specificato è presente nella base dati.

Postcondizioni: I dati del sensore con l'id specificato sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di un sensore mediante la selezione per mezzo dell'id.

UC1.1.3: Lettura dati dei sensori mediante selezione per nome

Attori Principali: Client.

Precondizioni: Esiste almeno un sensore nella base dati.

Postcondizioni: Tutti i dati dei sensori che hanno il nome specificato sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di uno o più sensori mediante la selezione per mezzo del nome.

UC1.1.4: Lettura dati dei sensori mediante selezione per tipo

Attori Principali: Client.

Precondizioni: Esiste almeno un sensore nella base dati.

Postcondizioni: Tutti i dati dei sensori che hanno il tipo specificato sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati dei sensori che hanno il tipo specificato.

UC1.1.5: Lettura dati dei sensori con stato attivo

Attori Principali: Client.

Precondizioni: Almeno un sensore è presente nella base dati.

Postcondizioni: Tutti i dati dei sensori con stato attivo sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori con stato attivo.

UC1.1.6: Lettura dati dei sensori con stato inattivo

Attori Principali: Client.

Precondizioni: Almeno un sensore è presente nella base dati.

Postcondizioni: Tutti i dati dei sensori con stato inattivo sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori con stato inattivo.

UC1.1.3 Lettura dati dei sensori mediante selezione per nome

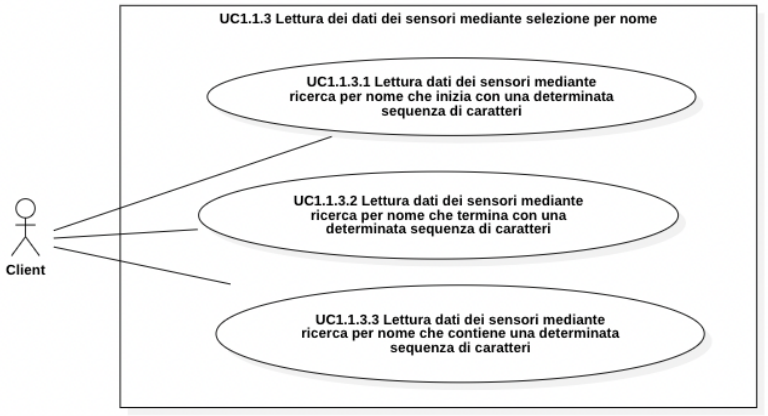


Figura 7: UC1.1.3 - Lettura dati dei sensori mediante selezione per nome

UC1.1.3.1: Lettura dati dei sensori mediante ricerca per nome che inizia con una determinata sequenza di caratteri

Attori Principali: Client.

Precondizioni: Almeno un sensore è presente nella base dati.

Postcondizioni: Tutti i dati dei sensori il cui nome inizia con la sequenza di caratteri specificata sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori il cui nome inizia con la sequenza di caratteri specificata.

UC1.1.3.2: Lettura dati dei sensori mediante ricerca per nome che termina con una determinata sequenza di caratteri

Attori Principali: Client.

Precondizioni: Almeno un sensore è presente nella base dati.

Postcondizioni: Tutti i dati dei sensori il cui nome termina con la sequenza di caratteri specificata sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori il cui nome termina con la sequenza di caratteri specificata.

UC1.1.3.3: Lettura dati dei sensori mediante ricerca per nome che contiene una determinata sequenza di caratteri

Attori Principali: Client.

Precondizioni: Almeno un sensore è presente nella base dati.

Postcondizioni: Tutti i dati dei sensori il cui nome contiene la sequenza di caratteri specificata sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutti i sensori il cui nome contiene la sequenza di caratteri specificata.

UC1.2 Aggiornamento dei dati dei sensori nella base dati

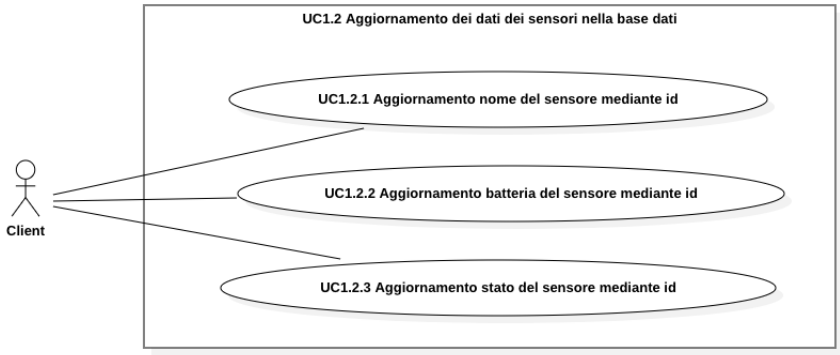


Figura 8: UC1.2 - Aggiornamento dei dati dei sensori nella base dati

UC1.2.1: Aggiornamento nome del sensore mediante id

Attori Principali: Client.

Precondizioni: Un sensore con l'id specificato è presente nella base dati e il client inserisce un nuovo nome come parametro di aggiornamento.

Postcondizioni: Il nome del sensore con l'id specificato è stato aggiornato nella base dati al valore specificato dal client.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare il nome del sensore con l'id specificato.

UC1.2.2: Aggiornamento batteria del sensore mediante id

Attori Principali: Client.

Precondizioni: Un sensore con l'id specificato è presente nella base dati e il client inserisce nel corpo della richiesta il valore della batteria.

Postcondizioni: La batteria del sensore con l'id specificato è stata aggiornata nella base dati al valore specificato dal client.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare la batteria del sensore con l'id specificato.

UC1.2.3: Aggiornamento stato del sensore mediante id

Attori Principali: Client.

Precondizioni: Un sensore con l'id specificato è presente nella base dati e il client inserisce nel corpo della richiesta il valore dello stato.

Postcondizioni: Lo stato del sensore con l'id specificato è stato aggiornato nella base dati al valore specificato dal client.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare lo stato del sensore con l'id specificato.

3.3.2 UC2 Operazioni CRUD postazione di parcheggio

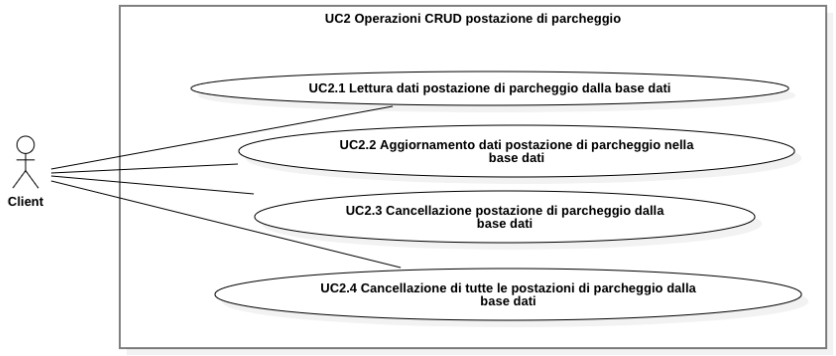


Figura 9: UC2 - Operazioni CRUD postazione di parcheggio

UC2.1: Lettura dati postazione di parcheggio dalla base dati

Attori Principali: Client.

Precondizioni: Esiste almeno una postazione di parcheggio nella base dati.

Postcondizioni: I dati delle postazioni di parcheggio sono stati letti dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati delle postazioni di parcheggio.

UC2.2: Aggiornamento dati postazione di parcheggio nella base dati

Attori Principali: Client.

Precondizioni: Sono presenti le postazioni di parcheggio che si desidera aggiornare all'interno della base dati.

Postcondizioni: I dati delle postazioni di parcheggio nella base dati sono stati

aggiornati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare i dati delle postazioni di parcheggio.

UC2.3: Cancellazione postazione di parcheggio dalla base dati

Attori Principali: Client.

Precondizioni: All'interno della base dati sono presenti le postazioni di parcheggio che si desidera eliminare.

Postcondizioni: I dati delle postazioni di parcheggio riferite sono stati eliminati dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare i dati delle postazioni di parcheggio.

UC2.4: Cancellazione di tutte le postazioni di parcheggio dalla base dati

Attori Principali: Client.

Precondizioni: All'interno della base dati è presente almeno una postazione di parcheggio.

Postcondizioni: I dati di tutte le postazioni di parcheggio sono stati eliminati dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare i dati di tutte le postazioni di parcheggio.

UC2.1 Lettura dati postazione di parcheggio dalla base dati

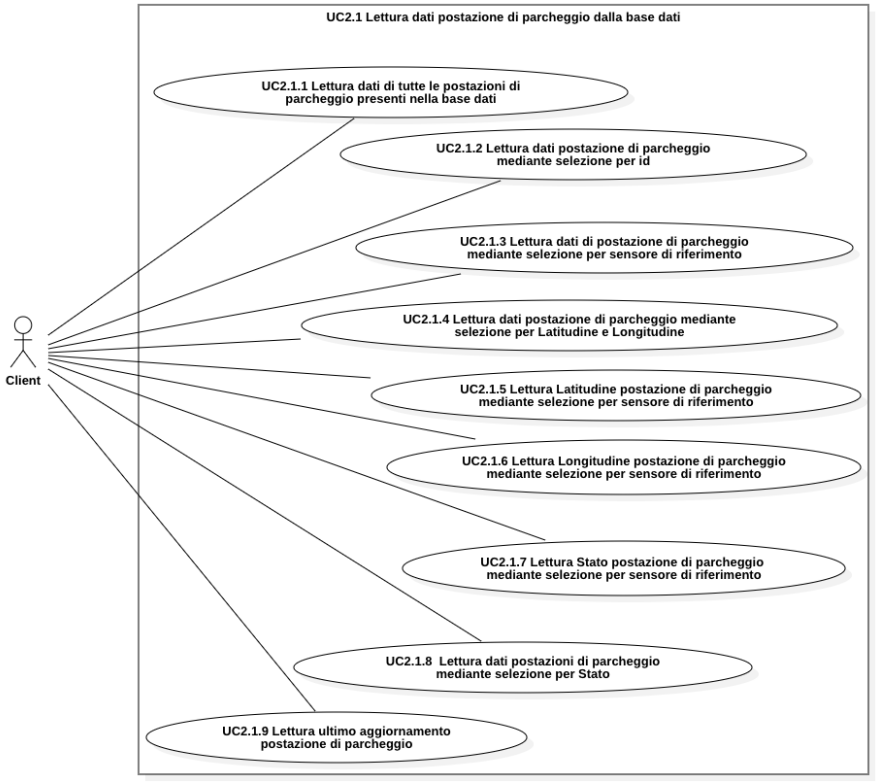


Figura 10: UC2.1 - Lettura dati postazione di parcheggio dalla base dati

UC2.1.1: Lettura dati di tutte le postazioni di parcheggio presenti nella base dati

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una postazione di parcheggio.

Postcondizioni: Tutte le postazioni di parcheggio contenute all'interno della base dati sono state lette.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati di tutte le postazioni di parcheggio.

UC2.1.2: Lettura dati postazioni di parcheggio mediante selezione per id

Attori Principali: Client.

Precondizioni: La base dati contiene un sensore con id uguale a quello specificato.

Postcondizioni: I dati della postazione di parcheggio con l'id specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati della postazione di parcheggio con l'id specificato.

UC2.1.3: Lettura dati postazioni di parcheggio mediante selezione per sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio che ha a sua volta un riferimento al sensore specificato.

Postcondizioni: I dati della postazione di parcheggio con riferimento al sensore specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati della postazione di parcheggio con riferimento al sensore specificato.

UC2.1.4: Lettura dati postazioni di parcheggio mediante selezione per latitudine e longitudine

Attori Principali: Client.

Precondizioni: Ogni occorrenza di postazione di parcheggio nella base dati ha come chiave latitudine e longitudine.

Postcondizioni: I dati della postazione di parcheggio con latitudine e longitudine di riferimento sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati della postazione di parcheggio con latitudine e longitudine di riferimento.

UC2.1.5: Lettura Latitudine postazione di parcheggio mediante selezione per sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio che ha a sua volta un riferimento al sensore specificato.

Postcondizioni: La latitudine della postazione di parcheggio con riferimento al sensore specificato è stata letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere la latitudine della postazione di parcheggio con riferimento al sensore specificato.

UC2.1.6: Lettura Longitudine postazione di parcheggio mediante selezione per sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio che ha a sua volta un riferimento al sensore specificato.

Postcondizioni: La longitudine della postazione di parcheggio con riferimento al

sensore specificato è stata letta.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere la longitudine della postazione di parcheggio con riferimento al sensore specificato.

UC2.1.7: Lettura Stato postazione di parcheggio mediante selezione per sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio che ha a sua volta un riferimento al sensore specificato.

Postcondizioni: Lo stato della postazione di parcheggio con riferimento al sensore specificato è stato letto.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere lo stato della postazione di parcheggio con riferimento al sensore specificato.

UC2.1.8: Lettura dati postazione di parcheggio mediante selezione per Stato

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una postazione di parcheggio.

Postcondizioni: Tutti i dati della postazione di parcheggio con lo stato specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere i dati della postazione di parcheggio con lo stato specificato.

UC2.1.9: Lettura ultimo aggiornamento postazione di parcheggio mediante selezione per id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato. Tale postazione deve essersi aggiornata almeno una volta nel suo ciclo di vita.

Postcondizioni: L'ultimo aggiornamento della postazione di parcheggio con l'id di riferimento è stato letto.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per ottenere l'ultimo aggiornamento della postazione di parcheggio con l'id specificato.

UC2.2 Aggiornamento dati postazione di parcheggio

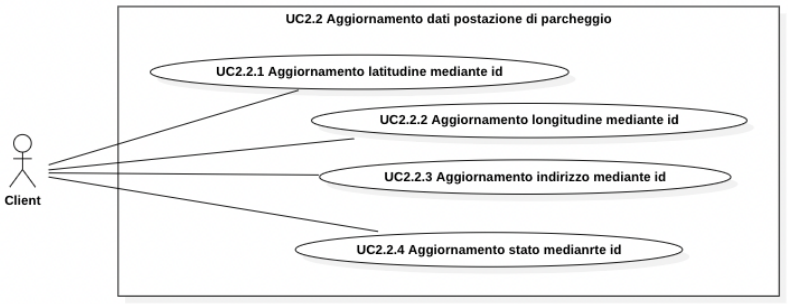


Figura 11: UC2.2 - Aggiornamento dati postazione di parcheggio dalla base dati

UC2.2.1: Aggiornamento latitudine mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato.

Postcondizioni: La latitudine della postazione di parcheggio con l'id specificato è stata aggiornata.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare la latitudine della postazione di parcheggio con l'id specificato.

UC2.2.2: Aggiornamento longitudine mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato.

Postcondizioni: La longitudine della postazione di parcheggio con l'id specificato è stata aggiornata.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare la longitudine della postazione di parcheggio con l'id specificato.

UC2.2.3: Aggiornamento indirizzo mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato.

Postcondizioni: L'indirizzo della postazione di parcheggio con l'id specificato è stato aggiornato.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare l'indirizzo della postazione di parcheggio con l'id specificato.

UC2.2.4: Aggiornamento stato mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato.

Postcondizioni: L'indirizzo della postazione di parcheggio con l'id specificato è stato aggiornato.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare lo stato della postazione di parcheggio con l'id specificato.

UC2.3 Cancellazione dati postazione di parcheggio dalla base dati

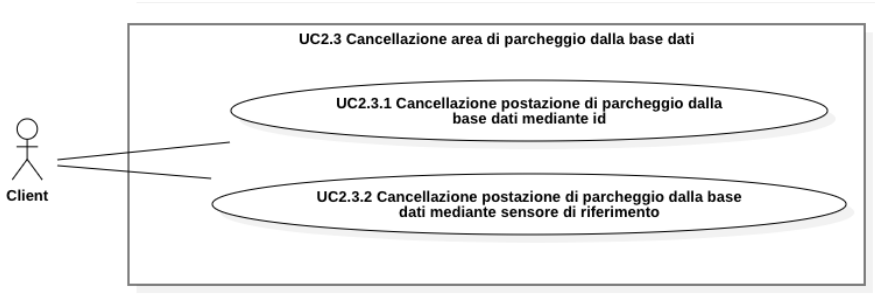


Figura 12: UC2.3 - Cancellazione dati postazione di parcheggio dalla base dati

UC2.3.1: Cancellazione postazione di parcheggio dalla base dati mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio con l'id specificato.

Postcondizioni: La postazione di parcheggio con l'id specificato è stata eliminata dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare la la postazione di parcheggio con l'id specificato.

UC2.3.2: Cancellazione postazione di parcheggio dalla base dati mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una postazione di parcheggio che si riferisce al sensore specificato.

Postcondizioni: La postazione di parcheggio che si riferisce al sensore specificato è stata cancellata dalla base dati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare la postazione di parcheggio che si riferisce al sensore specificato.

3.3.3 UC3 - Operazioni CRUD Servizio di manutenzione e controllo sensori

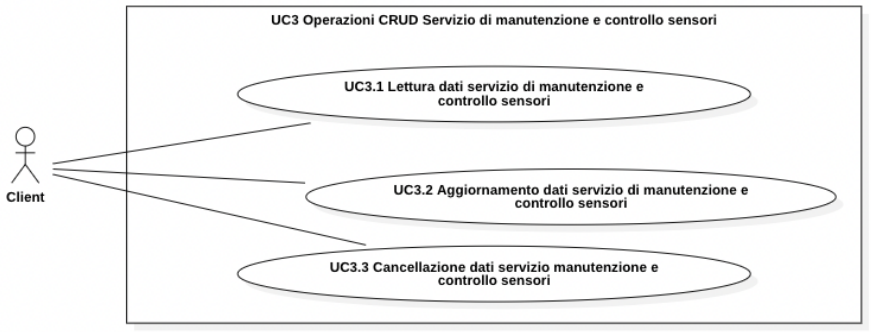


Figura 13: UC3 - Operazioni CRUD Servizio di manutenzione e controllo sensori

UC3.1: Lettura dati servizio di manutenzione e controllo sensori

Attori Principali: Client.

Precondizioni: La base dati contiene almeno un manutentore.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati del servizio di manutenzione e controllo sensori.

UC3.2: Aggiornamento dati servizio di manutenzione e controllo sensori

Attori Principali: Client.

Precondizioni: La base dati contiene almeno un manutentore.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori sono stati aggiornati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per leggere i dati del servizio di manutenzione e controllo sensori.

UC3.3: Cancellazione dati servizio di manutenzione e controllo sensori

Attori Principali: Client.

Precondizioni: La base dati contiene almeno un manutentore.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per eliminare i dati del servizio di manutenzione e controllo sensori.

UC3.1 Lettura dati servizio di manutenzione e controllo sensori

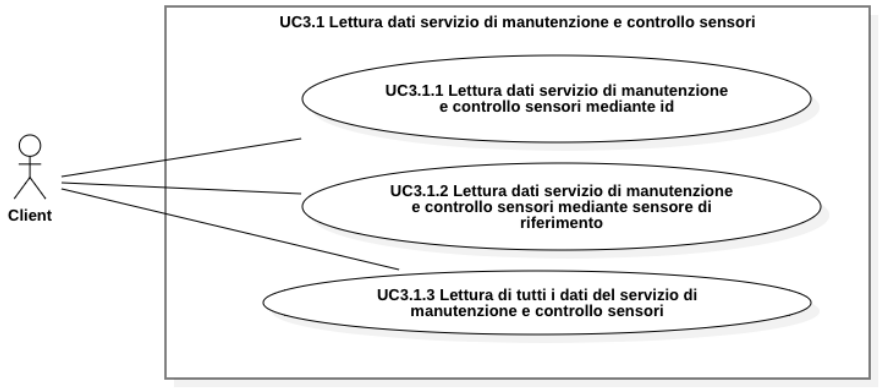


Figura 14: UC3.1 - Lettura dati servizio di manutenzione e controllo sensori

UC3.1.1: Lettura dati servizio di manutenzione e controllo sensori mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore con l'id specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori con l'id specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati del servizio di manutenzione e controllo sensori con l'id specificato.

UC3.1.2: Lettura dati servizio di manutenzione e controllo sensori mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore che si riferisce al sensore specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato.

UC3.1.3: Lettura di tutti i dati del servizio di manutenzione e controllo sensori

Attori Principali: Client.

Precondizioni: La base dati contiene almeno un manutentore.

Postcondizioni: Tutti i dati del servizio di manutenzione e controllo sensori sono

stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere tutti i dati del servizio di manutenzione e controllo sensori presenti all'interno della base dati.

UC3.2 Aggiornamento dati servizio di manutenzione e controllo sensori

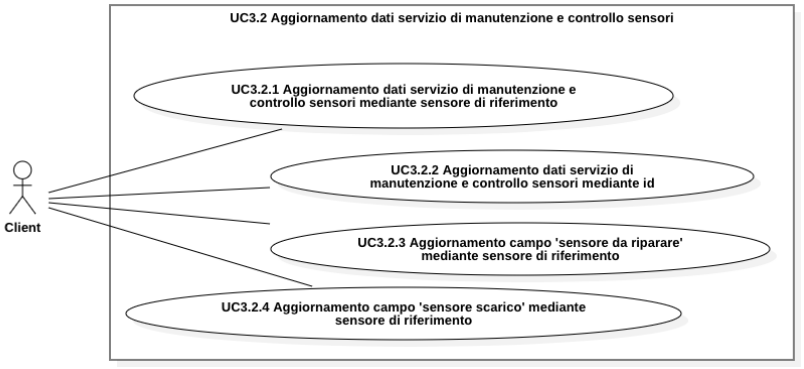


Figura 15: UC3.2 - Aggiornamento dati servizio di manutenzione e controllo sensori

UC3.2.1: Aggiornamento dati servizio di manutenzione e controllo sensori mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore che si riferisce al sensore specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato sono stati aggiornati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare i dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato.

UC3.2.2: Aggiornamento dati servizio di manutenzione e controllo sensori mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore con l'id specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori con l'id specificato sono stati aggiornati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare i dati del servizio di manutenzione e controllo sensori con l'id specificato.

UC3.2.3: Aggiornamento del campo 'sensore da riparare' mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore che si riferisce al sensore specificato.

Postcondizioni: Il campo 'sensore da riparare' del servizio di manutenzione e controllo sensori riguardante il sensore specificato è stato aggiornato.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare il campo 'sensore da riparare' del servizio di manutenzione e controllo sensori riguardante il sensore specificato.

UC3.2.4: Aggiornamento del campo 'sensore scarico' mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore che si riferisce al sensore specificato.

Postcondizioni: Il campo 'sensore scarico' del servizio di manutenzione e controllo sensori riguardante il sensore specificato è stato aggiornato.

Scenario Principale: Il client effettua una richiesta HTTP di tipo PUT al servizio per aggiornare il campo 'sensore da riparare' del servizio di manutenzione e controllo sensori riguardante il sensore specificato.

UC3.3 Cancellazione dati servizio manutenzione e controllo sensori

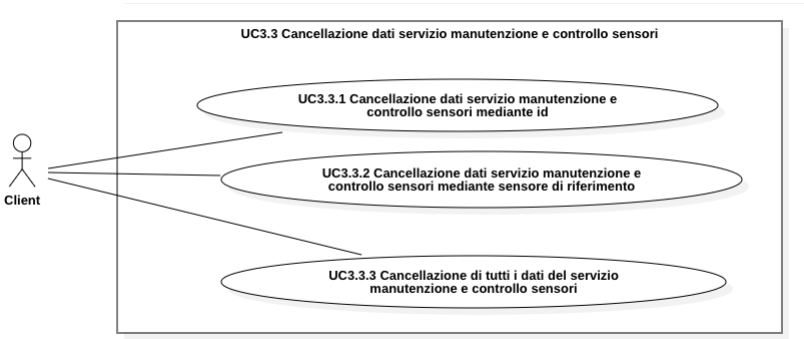


Figura 16: UC3.3 - Cancellazione dati servizio manutenzione e controllo sensori

UC3.3.1: Cancellazione dati servizio manutenzione e controllo sensori mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore con l'id specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori con l'id specificato sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati del servizio di manutenzione e controllo sensori con l'id specificato.

UC3.3.2: Cancellazione dati servizio manutenzione e controllo sensori mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un manutentore che si riferisce al sensore specificato.

Postcondizioni: I dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati del servizio di manutenzione e controllo sensori riguardanti il sensore specificato.

UC3.3.3: Cancellazione di tutti i dati del servizio manutenzione e controllo sensori

Attori Principali: Client.

Precondizioni: La base dati contiene almeno un manutentore.

Postcondizioni: Tutti i dati del servizio di manutenzione e controllo sensori sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per Cancellate tutti i dati del servizio di manutenzione e controllo sensori.

3.3.4 UC4 - Operazioni CRUD Statistiche di parcheggio

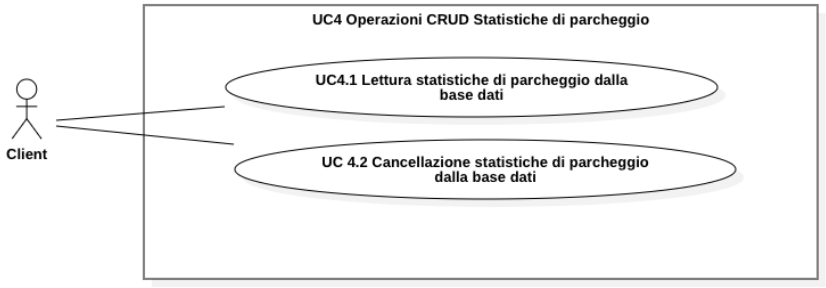


Figura 17: UC4 - Operazioni CRUD Statistiche di parcheggio

UC4.1: Lettura statistiche di parcheggio dalla base di dati

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una statistica di parcheggio.

Postcondizioni: I dati delle statistiche di parcheggio sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati delle statistiche di parcheggio.

UC4.2: Cancellazione statistiche di parcheggio dalla base di dati

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una statistica di parcheggio.

Postcondizioni: I dati delle statistiche di parcheggio sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati delle statistiche di parcheggio.

UC4.1 Lettura statistiche di parcheggio dalla base di dati

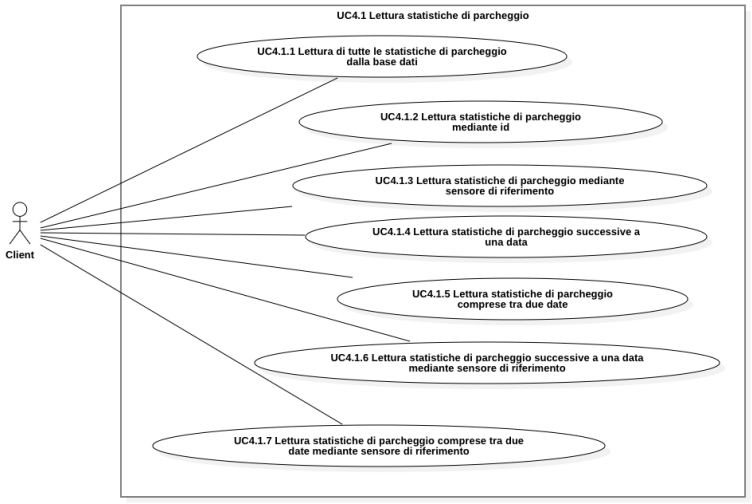


Figura 18: UC4.1 - Lettura statistiche di parcheggio dalla base di dati

UC4.1.1: Lettura di tutte le statistiche di parcheggio dalla base dati

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una statistica di parcheggio.

Postcondizioni: I dati di tutte le statistiche di parcheggio sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati di tutte le statistiche di parcheggio.

UC4.1.2: Lettura statistiche di parcheggio mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una statistica di parcheggio con l'id specificato.

Postcondizioni: I dati della statistica di parcheggio con l'id specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati della statistica di parcheggio con l'id specificato.

UC4.1.3: Lettura statistiche di parcheggio mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene una statistica di parcheggio che si riferisce al sensore specificato.

Postcondizioni: I dati della statistica di parcheggio che si riferisce al sensore specificato sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati della statistica di parcheggio che si riferisce al sensore specificato.

UC4.1.4: Lettura statistiche di parcheggio successive a una data

Attori Principali: Client.

Precondizioni: La base dati contiene una statistica di parcheggio.

Postcondizioni: I dati delle statistiche di parcheggio rilevate in data successiva alla data specificata sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati delle statistiche di parcheggio rilevate in data successiva alla data specificata.

UC4.1.5: Lettura statistiche di parcheggio comprese tra due date

Attori Principali: Client.

Precondizioni: La base dati contiene una statistica di parcheggio.

Postcondizioni: I dati delle statistiche di parcheggio rilevate in data compresa tra le date specificate sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati delle statistiche di parcheggio rilevate in data compresa tra le date specificate.

UC4.1.6: Lettura statistiche di parcheggio successive a una data mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un sensore con l'id del sensore di riferimento.

Postcondizioni: I dati delle statistiche di parcheggio che si riferiscono al sensore

specificato e che sono state rilevate in data successiva alla data specificata sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati delle statistiche di parcheggio che si riferiscono al sensore con l'id specificato e che sono state rilevate in data successiva alla data specificata.

UC4.1.7: Lettura statistiche di parcheggio comprese tra due date mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un sensore con l'id del sensore di riferimento.

Postcondizioni: I dati delle statistiche di parcheggio che si riferiscono al sensore specificato e che sono state rilevate in data compresa tra due le date specificate sono stati letti.

Scenario Principale: Il client effettua una richiesta HTTP di tipo GET al servizio per leggere i dati delle statistiche di parcheggio che si riferiscono al sensore con l'id specificato e che sono state rilevate in data compresa tra le due date specificate.

UC4.2 Cancellazione statistiche di parcheggio dalla base dati

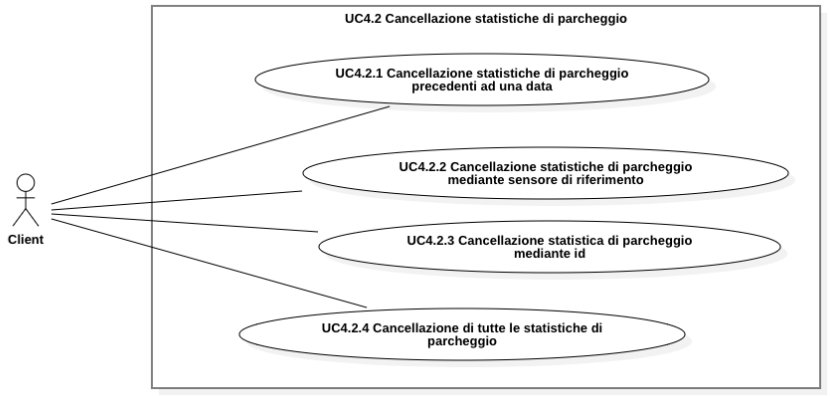


Figura 19: UC4.2 - Cancellazione statistiche di parcheggio dalla base dati

UC4.2.1: Cancellazione statistiche di parcheggio precedenti ad una data

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una statistica di parcheggio.

Postcondizioni: I dati delle statistiche di parcheggio rilevate in data precedente alla data specificata sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati delle statistiche di parcheggio rilevate in data precedente alla data specificata.

UC4.2.2: Cancellazione statistiche di parcheggio mediante sensore di riferimento

Attori Principali: Client.

Precondizioni: La base dati contiene un sensore con l'id del sensore di riferimento.

Postcondizioni: I dati delle statistiche di parcheggio che si riferiscono al sensore di riferimento sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati delle statistiche di parcheggio che si riferiscono al sensore di riferimento.

UC4.2.3: Cancellazione statistiche di parcheggio mediante id

Attori Principali: Client.

Precondizioni: La base dati contiene una statistica di parcheggio con l'id indicato.

Postcondizioni: I dati della statistica di parcheggio con l'id indicato sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare i dati della statistica di parcheggio con l'id indicato.

UC4.2.4: Cancellazione di tutte le statistiche di parcheggio

Attori Principali: Client.

Precondizioni: La base dati contiene almeno una statistica di parcheggio.

Postcondizioni: I dati di tutte le statistiche di parcheggio sono stati cancellati.

Scenario Principale: Il client effettua una richiesta HTTP di tipo DELETE al servizio per cancellare tutti i dati delle statistiche di parcheggio.

3.4 Definizione e tracciamento dei requisiti

Completata la redazione dei casi d'uso si è proceduto alla fase di definizione e tracciamento dei requisiti. A ogni requisito individuato è stato assegnato un codice univoco del tipo:

RQX.Y

in cui:

- la prima lettera è **R** che sta per 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;
  - **S** per i requisiti di sicurezza.

- X è il numero corrispondente alla cifra che appare nel macro UC;
- Y è un numero crescente che parte da 1 e viene incrementato di 1 per ogni requisito dello stesso tipo e con lo stesso macro UC.

3.4.1 Requisiti funzionali

Codice	Descrizione	Rilevanza	Fonti
RF1.1	Deve essere possibile leggere i dati del sensore dal relativo file in formato XML e scriverli nel DataBase	Obbligatorio	Azienda
RF1.2	Le risposte prodotte dal servizio devono essere in formato JSON	Obbligatorio	Azienda
RF1.3	Se durante la lettura sono presenti più sensori di quelli registrati nel DB, questi devono essere registrati nel DB come nuovi sensori	Obbligatorio	Scelta interna
RF1.4	Deve essere possibile leggere i dati di tutti i sensori dal DB	Obbligatorio	UC1.1.1
RF1.5	Deve esserre possibile leggere i dati di un sensore selezionato mediante id/nome/tipo	Obbligatorio	UC1.1.2, UC1.1.3, UC1.1.4
RF1.6	Deve essere possibile leggere i dati dei sensori attivi/inattivi	Obbligatorio	UC1.1.5, UC1.1.6
RF1.7	Deve essere possibile aggiornare nome/batteria/stato dei sensori nel DB	Obbligatorio	UC1.2.1, UC1.2.2, UC1.2.3
RF1.8	Deve essere possibile cancellare un sensore selezionato mediante id dal DB	Obbligatorio	UC1.3
RF1.9	Deve essere possibile cancellare tutti i sensori dal DB	Obbligatorio	UC1.4
RF1.10	In caso di malfunzionamento di un sensore deve essere generata una mail automatica che verrà inviata al manutentore del sensore	Obbligatorio	Aziedna
RF1.11	In caso di sensore con batteria scarica deve essere generata una mail automatica che verrà inviata al manutentore del sensore	Obbligatorio	Azienda
RF1.12	In caso di spegnimento di un sensore deve essere generata una mail automatica che verrà inviata al manutentore del sensore	Obbligatorio	Azienda

Codice	Descrizione	Rilevanza	Fonti
RF2.1	Deve essere possibile leggere i dati di tutte le postazioni di parcheggio presenti nel DB	Obbligatorio	UC2.1.1
RF2.2	Deve essere possibile leggere i dati delle postazioni di parcheggio presenti nel DB mediante selezione per id/sensore di riferimento/latitudine e langitudine	Obbligatorio	UC2.1.2, UC2.1.3, UC2.1.4
RF2.3	Deve essere possibile leggere la latitudine/longitudine/stato(libero/occupato) delle postazioni di parcheggio presenti nel DB mediante selezione per sensore di riferimento	Obbligatorio	UC2.1.5, UC2.1.6, UC2.1.7
RF2.4	Deve essere possibile leggere i dati delle postazioni di parcheggio con stato libero/occupato	Obbligatorio	UC2.1.8
RF2.5	Deve essere possibile leggere la data dell'ultimo aggiornamento di una postazione di parcheggio	Obbligatorio	UC2.1.9
RF2.6	Deve essere possibile aggiornare latitudine/-longitudine/indirizzo/stato della postazione di parcheggio mediante selezione per id	Obbligatorio	UC2.2.1, UC2.2.2, UC2.2.3, UC2.2.4
RF2.7	Deve essere possibile cancellare una postazione di parcheggio dal DB mediante selezione per id/sensore di riferimento	Obbligatorio	UC2.3.1, UC2.3.2
RF2.8	Deve essere possibile cancellare tutte le postazioni di parcheggio dal DB	Obbligatorio	UC2.4
RF3.1	Deve essere possibile leggere i dati del servizio di manutenzione e controllo sensori mediante id/sensore di riferimento	Obbligatorio	UC3.1.1, UC3.1.2
RF3.2	Deve essere possibile aggiornare i dati del servizio di manutenzione e controllo sensori mediante id/sensore di riferimento	Obbligatorio	UC3.2.1, UC3.2.2, UC3.2.3, UC3.2.4
RF3.3	Deve essere possibile cancellare i dati del servizio di manutenzione e controllo sensori mediante id/sensore di riferimento	Obbligatorio	UC3.3.1, UC3.3.2
RF3.4	Deve essere possibile cancellare tutti i dati del servizio di manutenzione e controllo sensori	Obbligatorio	UC3.3.3
RF4.1	Deve essere possibile ottenere le statistiche di ciascun sensore in modo da poter verificare il tasso di occupazione di ciascun parcheggio in un periodo di tempo stabilito	Obbligatorio	Azienda

Codice	Descrizione	Rilevanza	Fonti
RF4.2	Deve essere possibile leggere tutte le statistiche di parcheggio dal DB	Obbligatorio	UC4.4.1
RF4.3	Deve essere possibile leggere le statistiche di parcheggio dal DB mediante selezione per id/-sensore di riferimento	Obbligatorio	UC4.1.2, UC4.1.3
RF4.4	Deve essere possibile leggere dalla base di dati le statistiche di parcheggio rilevate in data successive a una data indicata	Obbligatorio	UC4.1.4
RF4.5	Deve essere possibile leggere dalla base di dati le statistiche di parcheggio rilevate in data compresa tra due date indicate	Obbligatorio	UC4.1.5
RF4.6	Deve essere possibile leggere dalla base di dati le statistiche di parcheggio rilevate in data successiva a una data indicata, mediante selezione per sensore di riferimento	Obbligatorio	UC4.1.6
RF4.7	Deve essere possibile leggere dalla base di dati le statistiche di parcheggio rilevate in data compresa tra due date indicate, mediante selezione per sensore di riferimento	Obbligatorio	UC4.1.7
RF4.8	Deve essere possibile cancellare dalla base di dati le statistiche di parcheggio rilevate in data precedente a una data indicata	Obbligatorio	UC4.2.1
RF4.9	Deve essere possibile cancellare dalla base di dati le statistiche di parcheggio mediante selezione per id/sensore di riferimento	Obbligatorio	UC4.2.2, UC4.2.3
RF4.10	Deve essere possibile Cancellare tutte le statistiche di parcheggio dal DB	Obbligatorio	UC4.2.4

Tabella 3.1: Requisiti funzionali

3.4.2 Requisiti qualitativi

Codice	Descrizione	Rilevanza	Fonti
RQ1	I test di unità del service devono avere una copertura >= 80%	Obbligatorio	Azienda

Codice	Descrizione	Rilevanza	Fonti
RQ2	I test di unità del service devono avere una copertura = 100%	Desiderabile	Azienda
RQ3	Deve essere prodotto un file di integrazione OpenAPI	Obbligatorio	Azienda
RQ4	La gestione delle mail deve essere gestita da un microservizio secondario che legge le informazioni delle mail da inviare mediante l'uso di code JMS	Desiderabile	Azienda
RQ5	Il testo delle mail deve essere scritto su un file di properties esterno	Desiderabile	Azienda

Tabella 3.2: Requisiti qualitativi

3.4.3 Requisiti di vincolo

Codice	Descrizione	Rilevanza	Fonti
RV1	L'applicativo deve essere realizzato mediante l'utilizzo del framework Spring per il linguaggio Java	Obbligatorio	Azienda
RV2	L'applicativo deve essere containerizzato mediante l'uso di Docker	Opzionale	Azienda

Tabella 3.3: Requisiti di vincolo

3.4.4 Requisiti di sicurezza

Codice	Descrizione	Rilevanza	Fonti
RS1	Le API REST prodotte devono essere protette mediante API_KEY[G]	Obbligatorio	Azienda

Tabella 3.4: Requisiti di sicurezza

Capitolo 4

Progettazione e codifica

In questo capitolo viene trattata la progettazione e la successiva realizzazione del prodotto software

4.1 Formazione sulle tecnologie

Come primo passo per la realizzazione del prodotto software è stata effettuata una formazione sulle tecnologie che sono state utilizzate per la fase di progettazione e codifica.

Java

È un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, che si appoggia sull'omonima piattaforma software di esecuzione, specificamente progettato per essere il più possibile indipendente dalla piattaforma hardware di esecuzione.

Spring Boot

Spring Boot è un framework per il linguaggio Java che permette di sviluppare applicazioni stand-alone. In particolare Spring Boot definisce una configurazione di base che include le linee guida per l'uso del framework e tutte le librerie di terze parti rilevanti, rendendo quindi l'avvio di nuovi progetti il più semplice possibile. In questo modo, la creazione di applicazioni indipendenti e pronte per la produzione basate su Spring può essere notevolmente semplificata.
Lo studio di questo framework ha impiegato la maggior parte del tempo di formazione durante lo stage.

Spring Data JPA

Spring Data JPA è un framework per il linguaggio Java che si occupa della gestione della persistenza dei dati di un S relazionale.

ActiveMq

ActiveMQ è un broker di messaggi rilasciato in maniera open source da Apache.

4.2 Valutazione del framework

Il framework scelto per lo sviluppo dell'applicazione è Spring basato sul linguaggio Java.

La scelta di tale framework ricade sul fatto che questo risulta essere consolidato e ben supportato, oltre ad essere molto utilizzato nel mondo del lavoro.
Altro importante fattore di scelta ricade sul fatto che in azienda molti dipendenti lo utilizzano quotidianamente, tanto da avere un topic dedicato a Spring nel server Discord aziendale che raccoglie tutto il materiale ritenuto utile all'apprendimento e all'utilizzo dello stesso, potendo così fornirmi il giusto supporto in caso di dubbi.

Spring fornisce un approccio modulare semplificato per la creazione di app con Java. La famiglia di progetti Spring è iniziata nel 2003 come risposta alle complessità dello sviluppo Java iniziale e fornisce supporto per lo sviluppo di app Java. Il nome stesso, Spring, fa in genere riferimento al framework dell'applicazione o all'intero gruppo di progetti o moduli.
Spring Boot e Spring Data JPA sono dei moduli specifici, creati come estensioni del framework Spring. Tali moduli sono stati quelli maggiormente approfonditi per lo sviluppo dell'applicativo e nel seguito verranno elencati i principali vantaggi:

- **Spring Boot**: Tale modulo ha semplificato di molto lo sviluppo dell'applicativo massimizzando la produttività grazie a caratteristiche quali:
  - Facilità nel generare un progetto Spring Boot di partenza, con dipendenze e configurazioni di base per mezzo del tool Spring Initializr[G];
  - Presenza di server HTTP integrati come Tomcat[G], Jetty[G] ecc. per sviluppare e testare le applicazioni Web molto facilmente;
  - Fornisce molti plugin[G] per sviluppare e testare applicazioni Spring Boot molto facilmente utilizzando Build Tools come Maven e Gradle;
  - È molto facile integrare l'applicazione Spring Boot con il suo ecosistema Spring come Spring JDBC, Spring ORM, Spring Data, Spring Security ecc;
  - Evita di scrivere molto codice "boilerplate[G]", annotazioni e configurazioni XML.
- **Spring Data JPA**: permette di semplificare lo stato di persistenza rimuovendo completamente l'implementazione dei DAO dalla nostra applicazione e creando automaticamente un'implementazione dotata dei metodi CRUD più rilevanti per l'accesso ai dati.

Grazie ai vantaggi sopra elencati, Spring ha permesso di ridurre di molto il tempo di sviluppo e aumentare la produttività.

4.3 Valutazione della base di dati

Grazie all'utilizzo del sopracitato Spring è risultato facile configurare due diversi tipi di database, entrambi relazionali.

Il database su cui è stata sviluppata e testata l'applicazione è un database di tipo PostgreSQL.
I test di maggiore importanza effettuati su tale database sono stati ripetuti su un database di tipo MySQL.

La scelta di utilizzare PostgreSQL come database principale è stata presa per questioni di familiarità con lo stesso dovuta a esperienze di utilizzo precedenti.
Tuttavia, seppur dopo averne parlato con gli stakeholders è stata ritenuta sufficiente un implementazione di tipo PostgreSQL, l'azienda aveva inizialmente richiesto un database di tipo MySQL e, vista la facilità di creazione del database dovuta a Spring Data JPA e alla facilità di cambiare tipo di base di dati semplicemente scegliendo la configurazione che si preferisce grazie a Spring Boot, si è scelto di fornire un'implementazione di entrambi i tipi.

Sono stati scelti database relazionali in quanto ci consentono di mappare nel modo migliore lo scenario che si presta alle funzionalità del software che viene brevemente descritto nel seguito:
l'applicativo presenta una tabella sensori (con la relativa anagrafica) in relazione con altre tabelle specifiche per tipo di sensore che contengono la posizione del sensore ed il valore o stato.
Un database di tipo SQL ci consente di fare query più dettagliate, quali ad esempio query per ottenere statistiche sullo stato di attività del sensore nel tempo, che si adattano meglio alle esigenze richieste per lo sviluppo.

L'immagine seguente mostra il diagramma ER[G] a partire dal quale è stato sviluppato il DB:

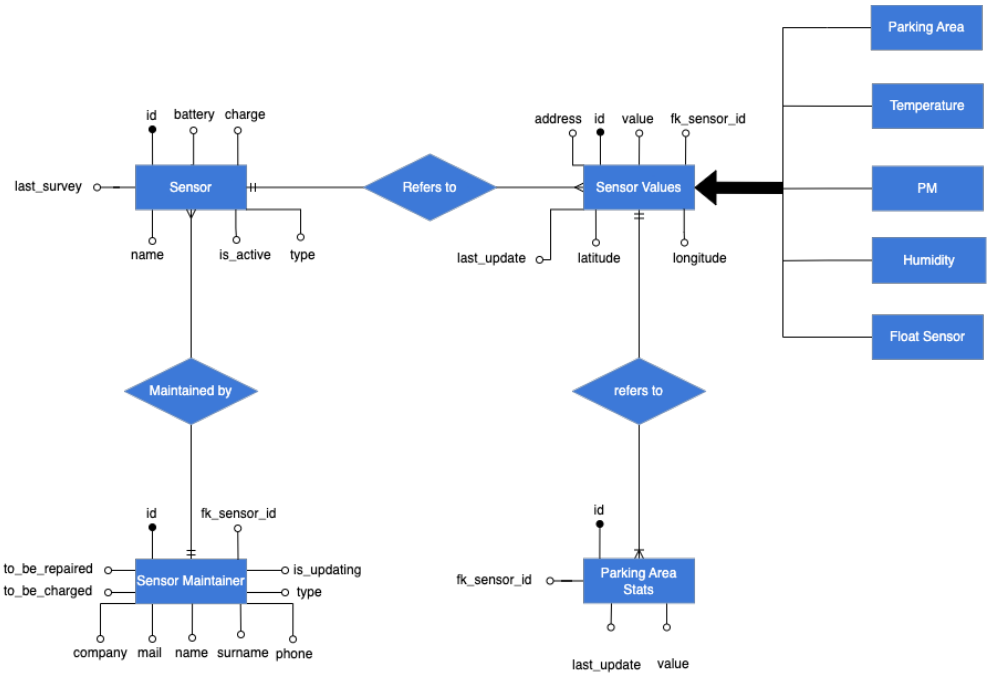


Figura 20: Diagramma ER della base di dati

4.4 Progettazione Architetturale

Lo schema architetturale utilizzato è quello mostrato e spiegato al paragrafo 1.3.

Inizialmente si è effettuato uno studio sulla differenza tra l'utilizzo di un'architettura monolitica e di una a microservizi dal quale è emerso quanto segue:
Un'**architettura monolitica** è un modello tradizionale di un programma software, compilata come un'unità unificata autonoma e indipendente dalle altre applicazioni.

Vantaggi del monolite

- **Semplicità**: È molto semplice da sviluppare e testare fin tanto che risulta essere un software di piccole dimensioni;
- **Debug agevole**: dal momento che il codice si trova in un'unica posizione, è più semplice l'esecuzione del debug.

Svantaggi del monolite

- **Velocità di sviluppo ridotta**: Nel caso in cui il software dovesse raggiungere grandi dimensioni lo sviluppo potrebbe diventare complicato;
- **Affidabilità**: Gli eventuali errori in un modulo potrebbero influire sulla disponibilità di tutta l'applicazione;
- **Mancanza di flessibilità**: Un monolite è vincolato dalle tecnologie già in uso nel monolite stesso. Le modifiche del framework o del linguaggio influiscono sull'intera applicazione, diventando spesso dispendiose in termini di costi e tempo.
- **Scalabilità**: La scalabilità è un problema per un monolite, in quanto non è possibile scalare solo alcune parti dell'applicazione, ma è necessario scalare l'intero monolite.
- **Distribuzione**: una minima modifica a un'applicazione monolitica richiede una nuova distribuzione dell'intero monolite.

Un'**architettura a microservizi** si basa su una serie di servizi distribuibili in modo indipendente. Questi servizi hanno una propria logica con un obiettivo specifico. Le attività di aggiornamento, test, distribuzione e ridimensionamento avvengono all'interno di ciascun servizio. I microservizi sono indipendenti e separati e, anche se non riducono la complessità, la rendono visibile e più gestibile separando i task in processi più piccoli in grado di funzionare in modo indipendente gli uni dagli altri e contribuire all'insieme generale.

Vantaggi dell'architettura a microservizi

- **Agilità di sviluppo**: Sviluppatori diversi possono lavorare in contemporanea su servizi diversi appartenenti alla medesima applicazione in modo indipendente;
- **Scalabilità flessibile**: Se un microservizio raggiunge la capacità di carico, è possibile distribuire rapidamente nuove istanze di tale servizio nel cluster che lo accompagna per ridurre la pressione;

- **Continuous deployment**: Cicli di rilascio più veloci e di conseguenza possibile farli con una maggiore frequenza;
- **Distribuzione indipendente**: Dal momento che i microservizi sono unità individuali, rendono possibile la distribuzione rapida, semplice e indipendente delle singole funzioni.
- **Flessibilità della tecnologia**: Nel caso in cui si voglia cambiare la tecnologia di un microservizio, non è necessario cambiare l'intero sistema;
- **Affidabilità elevata**: Possibilità di distribuire le modifiche di un servizio specifico senza la necessità di arrestare l'intera applicazione.

Svantaggi dell'architettura a microservizi

- **Debug**: Ogni microservizio ha il proprio set di log, il che complica il debug. In più, un singolo processo può essere eseguito su più macchine, complicando ulteriormente l'attività di debug.
- **Mancanza di standardizzazione**: senza una piattaforma comune, può esserci una proliferazione di linguaggi, standard di registrazione e monitoraggio.

L'architettura scelta per lo sviluppo del software è ricaduta sui microservizi in quanto ciascun tipo di sensore (parcheggio, ambientale, ecc.) potrebbe essere gestito dal relativo microservizio garantendo così una maggiore affidabilità poiché il codice di ciascun sensore è isolato nel relativo microservizio.

4.5 Codifica

4.5.1 Model

La prima fase di codifica mi ha visto impegnato nella creazione di un package[G] di classi per la rappresentazione dei dati.
Tale package prende il nome di **Model** e contiene le seguenti classi:

- **Marker**: identifica un sensore, necessaria per la traduzione dei dati rilevati dal sensore in oggetti Java;
- **MarkerList**: lista di marker;
- **SearchDataFilter**: classe per la gestione dei filtri di ricerca;
- **Sensor**: classe che identifica un sensore, necessaria ad effettuare operazioni CRUD nel database;
- **SensorMaintainer**: classe che identifica il manutentore di un sensore, necessaria ad effettuare operazioni CRUD nel database;
- **ParkingArea**: classe che identifica un'area di parcheggio, necessaria ad effettuare operazioni CRUD nel database;
- **ParkingAreaStats**: classe che identifica le statistiche di un'area di parcheggio, necessaria ad effettuare operazioni CRUD nel database;

4.5.2 Repository

La seconda fase di sviluppo è stata dedicata al package **Repository** che contiene le classi che permettono di gestire le operazioni CRUD sul database a partire dalle classi precedentemente citate nel **Model**.
Le classi contenute in tale package sono le seguenti:

- **SensorRepository**: classe che permette di effettuare operazioni CRUD sul database per la classe Sensor;
- **SensorMaintainerRepository**: classe che permette di effettuare operazioni CRUD sul database per la classe SensorMaintainer;
- **ParkingAreaRepository**: classe che permette di effettuare operazioni CRUD sul database per la classe ParkingArea;
- **ParkingAreaStatsRepository**: classe che permette di effettuare operazioni CRUD sul database per la classe ParkingAreaStats;

4.5.3 Service

Una volta completato il Repository, si è proceduto allo sviluppo in parallelo degli altri due package che compongono il primo microservizio.
Il primo, **Service**, che verrà descritto nel seguito di questo paragrafo, ed il secondo **Resources** che contiene l'esposizione degli endpoint delle API REST e per la descrizione del quale si rimanda al paragrafo successivo.

Il package **Service** contiene le classi che compongono la logica del microservizio.

Tutte le operazioni sugli oggetti avvengono a questo livello.

In particolare è stata creata una classe `StartUpServices` cuore dell'applicazione, più una classe di servizi per ciascun modello presente all'interno del database in maniera tale da gestire ogni tipo di oggetto in maniera indipendente dagli altri.

Il metodo sicuramente più importante della classe `StartUpServices` è il seguente:

```
90      // Do this every 2 minutes (Polling Function)
91      @Scheduled(cron = "${polling.timer}")
92      public void updateSensorsData() {
93          logger.debug(message: "StartUpServices START updateSensorsData");
94          try {
95              MarkerList sensors = readDataFromSources();
96              writeSensorsIfAdded(sensors);
97              updateDBData(sensors);
98          } catch (Exception e) {
99              logger.error(message: "StartUpServices ERROR updateSensorsData", e);
100         }
101         logger.debug(message: "StartUpServices END updateSensorsData");
102     }
```

Figura 21: Funzione di polling

in cui la variabile `polling.timer` è definita nel file `application.properties` come segue:

```
polling.timer = 0 */2 * * * *
```

Figura 22: Definizione della variabile `polling.timer`

Il metodo sopra riportato è quello che, dopo aver letto i dati trasmessi dai sensori, aggiorna il database mantenendolo costantemente aggiornato.
L'annotazione `@Scheduled` del framework Spring alla riga 91 insieme alla cron expression passata come parametro permette di richiamare tale funzione ogni due minuti.
All'interno di tale metodo vengono richiamati altri metodi che permettono di aggiungere nuovi eventuali sensori rilevati e di aggiornare i dati di quelli già esistenti qualora fossero stati rilevati cambiamenti dall'ultima rilevazione.
Avendo accesso solo al file dati del sensore, il polling è una soluzione per avere un dato quasi sempre aggiornato sullo stato del sensore, per simulare uno stato quasi in real time con intervallo di due minuti.

Altri metodi di StartUpService che meritano di essere citati:

- `convertXMLToJson(String xml)`: presa in input una stringa XML, ne effettua il parsing e restituisce una stringa in formato JSON. Questo metodo semplifica la creazione di oggetti Java grazie alla facilità di integrazione del formato JSON con il framework Spring;
- `readDataFromSources()`: metodo che legge i dati dai sensori e li restituisce sotto forma di oggetto di tipo markerList;

Le altre classi di `Service` contengono metodi per la creazione e la gestione degli oggetti per ciascun tipo di dato e si occupano di inviare i dati necessari alla composizione delle mail a una coda implementata utilizzando ActiveMQ.

La parte finale dello stage ha visto protagonista l'implementazione di un nuovo microserizio in grado di gestire l'invio delle email necessarie leggendo i dati dalla precedentemente citata coda JMS
La struttura del microservizio è rimasta simile a quella appena descritta. L'unico metodo degno di nota all'interno del services si occupa di impacchettare i dati letti dalla coda componendo una mail che verrà successivamente inviata.

4.6 Progettazione API REST

Durante lo sviluppo dell'applicativo sono state progettate e sviluppate 59 API REST. Nel seguente paragrafo verrà descritta la progettazione delle API REST di maggior rilievo che permettono di interagire con l'applicazione.

In particolare verranno descritte per ciascuna API:

- La definizione dell'interfaccia di input;
- La definizione del formato del tipo di ritorno;
- La gestione degli errori e le relative risposte.

Ciascuna API viene identificata in maniera univoca per mezzo di un path descritto come parametro delle annotazioni di mapping di Spring.

Le API sono state definite in classi diverse, una per ogni entità tra quelle descritte al paragrafo 3.2.
Ciascuna richiesta inviata da una di queste classi viene mappata a `/scs` che dunque comporrà la prima parte del path per accedere ciascuna API.

Viene riportata nel seguito la progettazione delle API di maggior rilievo.

4.6.1 Ottenere i dati di un sensore a partire dall'id

L'API per ottenere i dati di un sensore è stata definita nella classe `SensorResources` e mappata a `/scs/sensor/data/{sensorId}` con una richiesta di tipo GET.

Interfaccia di input

L'interfaccia di input è costituita da un parametro di tipo `Long` che rappresenta l'id del sensore di cui si vogliono ottenere i dati.
Dalla figura riportata di seguito è possibile notare che lo stesso parametro appare sia nella firma dell'API che nel path tra le parentesi {}.
Il valore di tale parametro verrà infatti prelevato dal path ed elaborato dal metodo in questione.

```java
@GetMapping("/sensor/data/{sensorId}")
@ResponseBody
public ResponseEntity<Object> getSensorData(@PathVariable Long sensorId) {
```

Figura 23: Mapping e firma dell'API per ottenere i dati di un sensore a partire dall'id.

Tipo di ritorno

Il tipo di ritorno è una ResponseEntity con codice di stato 200(OK) e response body contenente un oggetto di tipo sensore in formato JSON:

```java
return ResponseEntity.status(HttpStatus.OK).body(s);
```

Figura 24: Formato di ritorno dell'API per ottenere i dati di un sensore a partire dall'id.

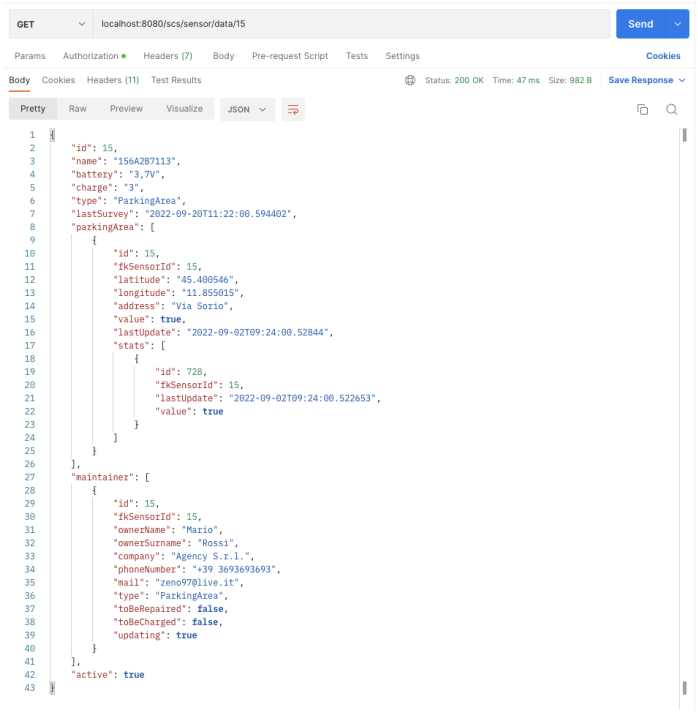


Figura 25: Response body dell'API per ottenere i dati di un sensore a partire dall'id.

Gestione degli errori

Nel caso venga riscontrato un errore viene sollevata un'eccezione che viene gestita in maniera diversa a seconda del tipo di errore catturato.

- Nel caso in cui l'errore catturato dall'eccezione sia di tipo `NullPointerException` viene restituito un oggetto di tipo ResponseEntity con codice di stato 404(NOT FOUND) e response body contenente un messaggio di errore;

- Nel caso in cui l'errore sollevato sia un eccezione di qualunque altro tipo viene restituito un oggetto di tipo ResponseEntity con codice di stato 500(INTERNAL SERVER ERROR).

```
try {
    s = sensorService.getSensorById(sensorId);
} catch (NullPointerException e) {
    logger.error(message: "SensorResources -  error", e);
    return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
}
catch (Exception e) {
    logger.error(message: "SensorResources -  error", e);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

Figura 26: Gestione degli errori dell'API per ottenere i dati di un sensore a partire dall'id.

4.6.2 Aggiornamento del nome di un sensore a partire dall'id

L'API per aggiornare il nome di un sensore a partire dall'id è stata definita nella classe `SensorResources` e mappata a `/scs/sensor/update/name/{sensorId}` con una richiesta di tipo PUT.

Interfaccia di input

L'interfaccia di input è costituita da un parametro di tipo `Long` che rappresenta l'id del sensore di cui si vogliono ottenere i dati.

```
@PutMapping("/sensor/update/name/{sensorId}")
public ResponseEntity<Object> updateSensorNameById(@RequestBody String name, @PathVariable Long sensorId) {
```

Figura 27: Mapping e firma dell'API per aggiornare il nome di un sensore a partire dall'id.

Tipo di ritorno

Il tipo di ritorno è una ResponseEntity con codice di stato 200(OK) e response body vuoto.

```
return ResponseEntity.status(HttpStatus.OK).build();
```

Figura 28: Formato di ritorno dell'API per aggiornare il nome di un sensore a partire dall'id.

Gestione degli errori

Nel caso venga riscontrato un errore viene sollevata un'eccezione che viene gestita in maniera diversa a seconda del tipo di errore catturato.

- Nel caso in cui l'errore catturato dall'eccezione sia di tipo `NullPointerException` viene restituito un oggetto di tipo ResponseEntity con codice di stato 404(NOT FOUND) e response body contenente un messaggio di errore;

- Nel caso in cui l'errore sollevato sia un eccezione di qualunque altro tipo viene restituito un oggetto di tipo ResponseEntity con codice di stato 500(INTERNAL SERVER ERROR).

```
} catch (NullPointerException e) {
    logger.error(message: "SensorResources -  error", e);
    return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
}
catch (Exception e) {
    logger.error(message: "SensorResources -  error", e);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

Figura 29: Gestione degli errori dell'API per aggiornare il nome di un sensore a partire dall'id.

4.6.3 Ottenere i dati di tutte le postazioni di parcheggio libere

L'API per ottenere i dati di tutte le postazioni di parcheggio libere è stata definita nella classe `ParkingAreaResources` e mappata a `/scs/parking-area/free` con una richiesta di tipo GET.

Interfaccia di input

L'interfaccia di input non presenta alcun parametro in ingresso.

```
@GetMapping("/parking-area/free")
@ResponseBody
public ResponseEntity<Object> getFreeParkingArea() {
```

Figura 30: Mapping e firma dell'API per ottenere i dati di tutte le postazioni di parcheggio libere.

Tipo di ritorno

Il tipo di ritorno è una ResponseEntity con codice di stato 200(OK) e response body contenente un oggetto di tipo `List<PostazioneDiParcheggio>` in formato JSON:

```
return ResponseEntity.status(HttpStatus.OK).body(freeAreas);
```

Figura 31: Formato di ritorno dell'API per ottenere i dati di tutte le postazioni di parcheggio libere.

GET localhost:8080/scs/parking-area/free Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Body 200 OK 148 ms 2.21 KB Save Response

Pretty Raw Preview Visualize JSON


```
[
    {
        "id": 3,
        "fkSensorId": 3,
        "latitude": "45.389028",
        "longitude": "11.928631",
        "address": "Padova Galleria Spagna",
        "value": false,
        "lastUpdate": "2022-09-01T09:42:00.307712",
        "stats": [
            {
                "id": 636,
                "fkSensorId": 3,
                "lastUpdate": "2022-09-01T09:42:00.294966",
                "value": false
            }
        ]
    },
    {
        "id": 4,
        "fkSensorId": 4,
        "latitude": "45.392648",
        "longitude": "11.904846",
        "address": "Via Forcellini",
        "value": false,
        "lastUpdate": "2022-08-26T17:10:00.487454",
        "stats": []
    },
    {
        "id": 7,
        "fkSensorId": 7,
        "latitude": "45.407958",
        "longitude": "11.872594",
        "address": "Piazza Capitaniato",
        "value": false,
        "lastUpdate": "2022-08-26T17:10:00.494007",
        "stats": []
    },
    {
        "id": 9,
        "fkSensorId": 9,
        "latitude": "45.407913",
        "longitude": "11.87258",
        "address": "Piazza Capitaniato",
        "value": false,
        "lastUpdate": "2022-08-26T17:10:00.497644",
        "stats": []
    },
```


Figura 32: Response body dell'API per ottenere i dati di tutte le postazioni di parcheggio libere.

Gestione degli errori

Nel caso venga riscontrato un errore viene sollevata un'eccezione e viene restituito un oggetto di tipo ResponseEntity con codice di stato 500(INTERNAL SERVER ERROR).

```
} catch (Exception e) {
    logger.error(message: "ParkiAreaResource -  error", e);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

Figura 33: Gestione degli errori dell'API per ottenere i dati di tutte le postazioni di parcheggio libere.

4.6.4 Eliminare i dati di un manutentore a partire dall'id

L'API per eliminare i dati di un manutentore a partire dall'id è stata definita nella classe `SensorsMaintainerResources` e mappata a `/scs/maintainers/delete/{id}` con una richiesta di tipo DELETE.

Interfaccia di input

L'interfaccia di input è costituita da un parametro di tipo Long che rappresenta l'id del manutentore di cui si vogliono eliminare i dati.

```
@DeleteMapping("/maintainers/delete/{id}")
public ResponseEntity<Object> deleteSensorMaintainersById(@PathVariable Long id) {
```

Figura 34: Mapping e firma dell'API per eliminare i dati di un manutentore a partire dall'id.

Tipo di ritorno

Il tipo di ritorno è una ResponseEntity con codice di stato 200(OK) e response body vuoto.

```
return ResponseEntity.status(HttpStatus.OK).build();
```

Figura 35: Formato di ritorno dell'API per eliminare i dati di un manutentore a partire dall'id.

Gestione degli errori

Nel caso venga riscontrato un errore viene sollevata un'eccezione che viene gestita in maniera diversa a seconda del tipo di errore catturato.

- Nel caso in cui l'errore catturato dall'eccezione sia di tipo `NullPointerException` viene restituito un oggetto di tipo ResponseEntity con codice di stato 404(NOT FOUND) e response body contenente un messaggio di errore;

- Nel caso in cui l'errore sollevato sia un eccezione di qualunque altro tipo viene restituito un oggetto di tipo ResponseEntity con codice di stato 500(INTERNAL SERVER ERROR).

```
} catch (NullPointerException e) {
    logger.error(message: "SensorMaintainerResources -  error - getSensorsMaintainersBySensorId", e);
    return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
} catch (Exception e) {
    logger.error(message: "SensorMaintainerResources -  error - getSensorsMaintainersBySensorId", e);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

Figura 36: Gestione degli errori dell'API per ottenere i dati di tutte le postazioni di parcheggio libere.

4.6.5 Ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo

L'API per ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo è stata definita nella classe `ParkingStatsResources` e mappata a `/scs/parking-stats/from-date-to-date` con una richiesta di tipo GET.

Interfaccia di input

L'interfaccia di input è costituita da un parametro di tipo `SearchDataFilter` che sarà inserito nel request body tramite il frontend.
Il tipo `searchDataFilter` è stato definito internamente ed è composto da due campi di tipo `LocalDateTime` e relativi getter e setter che rappresentano rispettivamente la data di inizio e la data di fine dell'intervallo di tempo in cui si vogliono ottenere le statistiche.

```
@GetMapping("/parking-stats/from-date-to-date")
@ResponseBody
public ResponseEntity<Object> getParkingAreaStatsFromDataToData(@RequestBody SearchDateFilter date) {
```

Figura 37: Mapping e firma dell'API per ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo.

Tipo di ritorno

Il tipo di ritorno è una ResponseEntity con codice di stato 200(OK) e response body contenente un oggetto di tipo `List<Statistiche>` in formato JSON:

```
return ResponseEntity.status(HttpStatus.OK).body(stats);
```

Figura 38: Formato di ritorno dell'API per ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo.

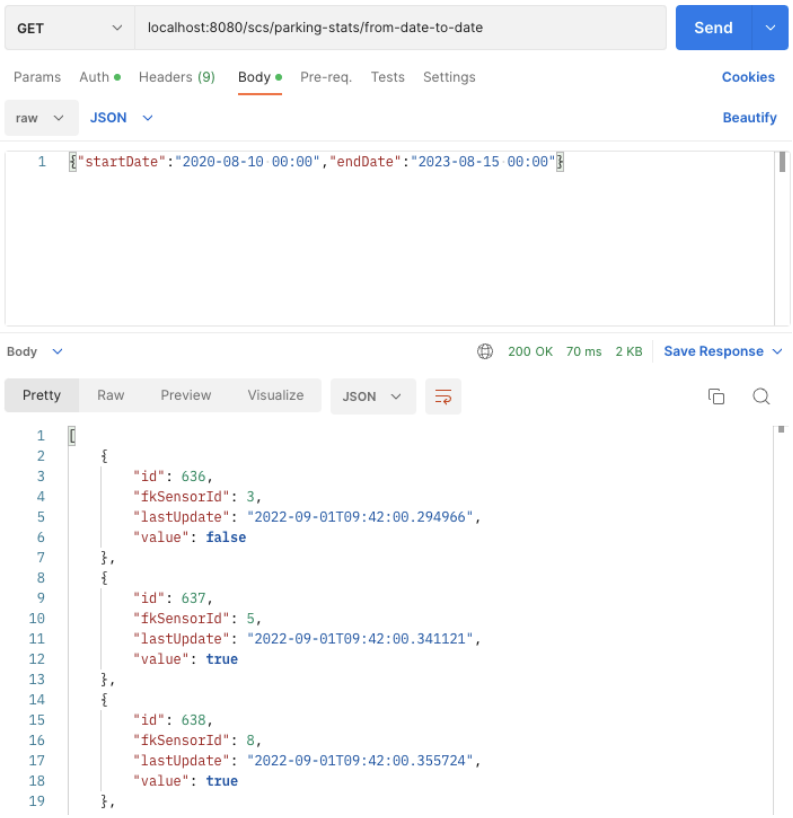


Figura 39: Response body dell'API per ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo.

Gestione degli errori

Nel caso venga riscontrato un errore viene sollevata un'eccezione che viene gestita in maniera diversa a seconda del tipo di errore catturato.

- Nel caso in cui l'errore catturato dall'eccezione sia di tipo `NullPointerException` viene restituito un oggetto di tipo ResponseEntity con codice di stato 404(NOT FOUND);
- Nel caso in cui l'errore sollevato sia un eccezione di qualunque altro tipo viene restituito un oggetto di tipo ResponseEntity con codice di stato 500(IINTERNAL SERVER ERROR).

```
} catch (NullPointerException e) {
    logger.error(message: "ParkingStatsResources -  error - getParkingAreaStatsById", e);
    return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
} catch (Exception e) {
    logger.error(message: "ParkingStatsResources -  error - getParkingAreaStatsById", e);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
}
```

Figura 40: Gestione degli errori dell'API per ottenere le statistiche delle postazioni di parcheggio in un determinato intervallo di tempo

4.6.6 Protezione delle API con Spring Security

Considerando il contesto di utilizzo delle API, che saranno principalmente consegnate agli sviluppatori che si occuperanno del frontend, è stato pensato di inserire un'API_KEY come parametro di input nell'header del request body di ogni richiesta.

Grazie alle librerie del framework Spring, e in particolare al modulo Spring Security, è stato semplice integrare un sistema di API_KEY così da rendere possibile l'accesso alle API solo a coloro i quali risultano autorizzati a farlo.
Mediante l'uso di Postman è poi stato possibile effettuare dei test sul funzionamento di tale meccanismo.
Le immagini che seguono mostrano come con la giusta chiave, il codice di stato restituito della chiamata è `200(OK)`, mentre con la chiave non autorizzata il codice di stato che si ottiene dalla chiamata è il `403(FORBIDDEN)`:

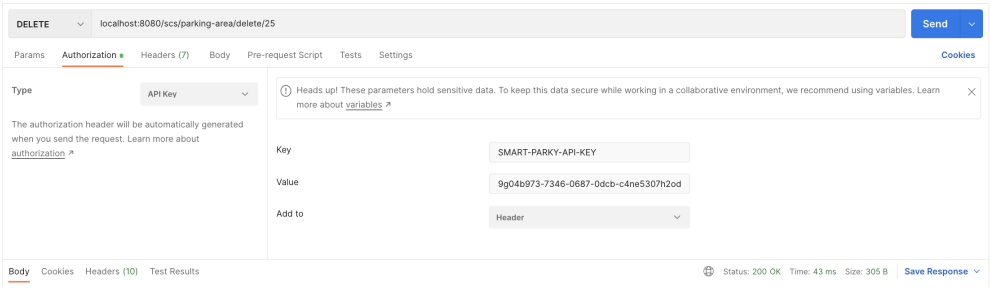


Figura 41: Risultato di una chiamata ad un'API mediante chiave autorizzata

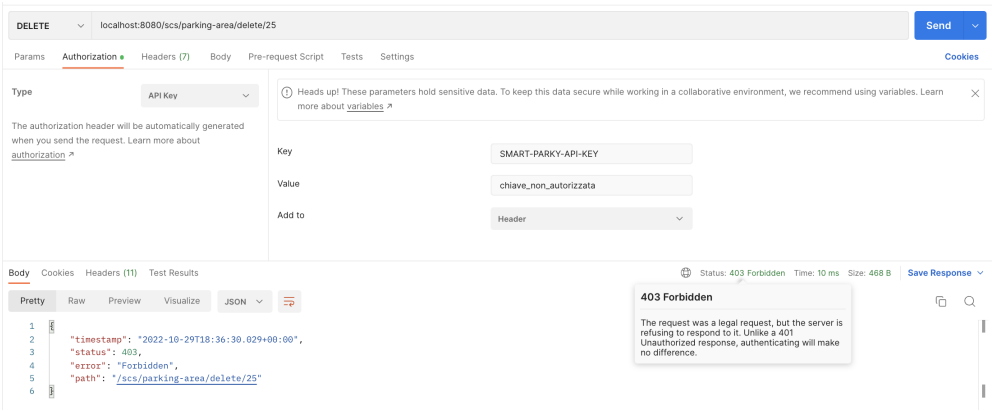


Figura 42: Risultato di una chiamata ad un'API mediante chiave non autorizzata

4.7 Logging di sistema

Durante la fase di codifica è stato implementato un sistema di logging che permette di tenere traccia di tutte le operazioni effettuate sul sistema. In particolare, tale sistema di logging è stato implementato utilizzando il framework Log4j.

La seguente tabella definisce i livelli dei log e i messaggi in Log4j in ordine decrescente di severità. La colonna di sinistra indica il livello di log designato, mentre quella di destra riporta una breve descrizione.

Livello	Descrizione
OFF	Il livello più alto possibile, viene usato per disattivare i log
FATAL	Errore importante che causa un prematuro termine dell'esecuzione
ERROR	Un errore di esecuzione o una condizione imprevista
WARN	Usato per ogni condizione inaspettata o anomalia di esecuzione, che però non necessariamente ha comportato un errore
INFO	Usato per segnalare eventi di esecuzione (esempio: startup/shutdown)
DEBUG	Usato nella fase di debug del programma
TRACE	Usato per tenere traccia di alcune informazioni dettagliate

Tabella 4.1: Livelli di logging

I livelli di logging maggiormente utilizzati all'interno del progetto sono stati i seguenti:

- **INFO**: utilizzato per segnalare l'entrata e l'uscita da un metodo, in modo da tenere traccia del funzionamento e verificarne la correttezza in fase di esecuzione;
- **ERROR**: utilizzato all'interno delle condizioni catch delle API REST che restituiscono una risposta di errore al client (400 - BAD REQUEST, 404 - NOT FOUND);
- **DEBUG**: utilizzato per tenere traccia dei valori delle variabili all'interno di un metodo o per segnalare informazioni riguardo al database.

È stato possibile definire delle proprietà all'interno di un file denominato `Log4j2.xml` che permettono di scrivere i log riguardanti l'esecuzione del software in uno o più file esterni a seconda dei livelli di log che si vogliono utilizzare. Una volta scelto il livello di log di partenza, saranno scritti all'interno del file tutti i log del livello definito e dei livelli con severità maggiore.

In particolare sono stati definiti due file di log. Il primo per il livello INFO, in modo da tenere sotto controllo il flusso di chiamate delle API REST e la corretta esecuzione delle stesse.
Il secondo file di log è stato definito per il livello DEBUG, in modo da tenere traccia delle operazioni di maggior rilievo effettuate dal programma.
Nel seguito viene riportato un estratto dei file di log generati durante l'esecuzione del programma:

```
2022-09-21 11:14:06 http-nio-8080-exec-9 [INFO] ParkingAreaResources:94 - ParkingAreaResource - START getParkingAreaLatitudeById
2022-09-21 11:14:06 http-nio-8080-exec-9 [INFO] ParkingAreaResources:106 - ParkingAreaResource - END getParkingAreaLatitudeById
2022-09-21 11:14:12 http-nio-8080-exec-10 [INFO] ParkingAreaResources:75 - ParkingAreaResource - START getParkingAreaDataBySensorId
2022-09-21 11:14:12 http-nio-8080-exec-10 [INFO] ParkingAreaResources:87 - ParkingAreaResource - END getParkingAreaDataBySensorId
2022-09-21 11:14:16 http-nio-8080-exec-1 [INFO] ParkingAreaResources:94 - ParkingAreaResource - START getParkingAreaLatitudeById
2022-09-21 11:14:16 http-nio-8080-exec-1 [INFO] ParkingAreaResources:106 - ParkingAreaResource - END getParkingAreaLatitudeById
2022-09-21 11:14:21 http-nio-8080-exec-2 [INFO] ParkingAreaResources:113 - ParkingAreaResource - START getParkingAreaLongitudeById
2022-09-21 11:14:21 http-nio-8080-exec-2 [INFO] ParkingAreaResources:125 - ParkingAreaResource - END getParkingAreaLongitudeById
2022-09-21 11:14:23 http-nio-8080-exec-3 [INFO] ParkingAreaResources:113 - ParkingAreaResource - START getParkingAreaLongitudeById
2022-09-21 11:14:23 http-nio-8080-exec-3 [INFO] ParkingAreaResources:125 - ParkingAreaResource - END getParkingAreaLongitudeById
2022-09-21 11:14:27 http-nio-8080-exec-5 [INFO] ParkingAreaResources:203 - ParkiAreangResource - START getParkingAreaByLatitudeAndLongitude
2022-09-21 11:14:27 http-nio-8080-exec-5 [INFO] ParkingAreaResources:215 - ParkiAreangResource - END getParkingAreaByLatitudeAndLongitude
2022-09-21 11:14:34 http-nio-8080-exec-4 [INFO] ParkingAreaResources:203 - ParkiAreangResource - START getParkingAreaByLatitudeAndLongitude
2022-09-21 11:14:34 http-nio-8080-exec-4 [ERROR] ParkingAreaResources:209 - ParkingAreaResources -  error
java.lang.NullPointerException: Cannot invoke "it.synclab.smartparking.repository.model.ParkingArea.getId()" because "p" is null
```

Figura 43: Estratto del file di log per il livello INFO

```
24705 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] SensorServices:126 - ParkingService END getSensorByNameContaining - Sequence:C - sensorsListSize:5
24706 2022-09-05 15:56:51 http-nio-8080-exec-10 [INFO] SensorResources:192 - SensorResources - END getSensorByNameContaining
24707 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] HttpEntityMethodProcessor:268 - Using 'application/json', given [*/*] and supported [application/json, applic
24708 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] AbstractLoadPlanBasedCollectionInitializer:73 - Loading collection: [it.synclab.smartparking.repository.model
24709 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] SQL:144 -
24710     select
24711         parkingare0_.fk_sensor_id as fk_senso3_0_0_,
24712         parkingare0_.id as id1_0_0_,
24713         parkingare0_.id as id1_0_1_,
24714         parkingare0_.address as address2_0_1_,
24715         parkingare0_.fk_sensor_id as fk_senso3_0_1_,
24716         parkingare0_.last_update as last_upd4_0_1_,
24717         parkingare0_.latitude as latitude5_0_1_,
24718         parkingare0_.longitude as longitud6_0_1_,
24719         parkingare0_.value as value7_0_1_
24720     from
24721         parking_area parkingare0_
24722     where
24723         parkingare0_.fk_sensor_id=?
24724 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] ResultSetProcessorImpl:193 - Preparing collection initializer : [it.synclab.smartparking.repository.model.Sen
24725 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] CollectionReferenceInitializerImpl:59 - Found row of collection: [it.synclab.smartparking.repository.model.Se
24726 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] TwoPhaseLoad:171 - Resolving attributes for [it.synclab.smartparking.repository.model.ParkingArea#13]
24727 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] TwoPhaseLoad:184 - Processing attribute `address` : value = Via Sorio
24728 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] TwoPhaseLoad:215 - Attribute (`address`)  - enhanced for lazy-loading? - false
24729 2022-09-05 15:56:51 http-nio-8080-exec-10 [DEBUG] TwoPhaseLoad:184 - Processing attribute `fkSensorId` : value = 13
```

Figura 44: Estratto del file di log per il livello DEBUG

Capitolo 5

Verifica e validazione

In questo capitolo vengono discusse le fasi di verififa e validazione del prodotto realizzato durante lo stage.

L'attività di verifica all'interno del progetto è stata costante grazie a continue revisioni di avanzamento con gli stakeholders.

5.1 Analisi statica

L'analisi statica è stata effettuata con l'ausilio di SonarLint, un plugin per l'IDE Eclipse che permette di analizzare il codice sorgente e di segnalare eventuali errori o bug[G]. Tale plugin permette di scovare errori di sintassi, bug e vulnerabilità[G] già in fase di scrittura del codice:

Figura 45: Segnalazioni SonarLint in fase di codifica

Oltre a tali segnalazioni SonarLint è anche in grado di analizzare un file, un package o l'intero progetto per mezzo di un'analisi che produrrà un risultato simile al seguente:

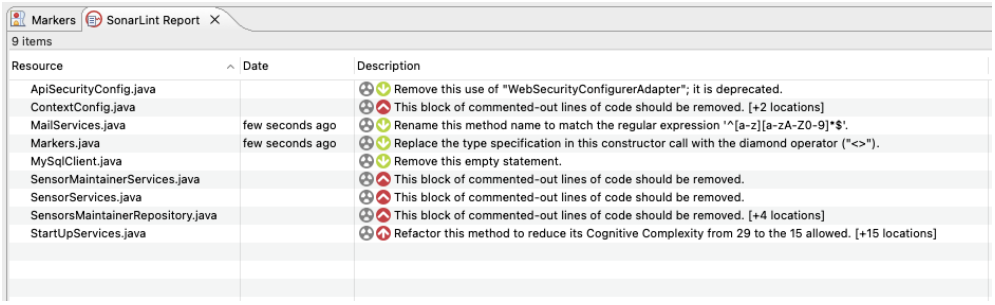


Figura 46: Analisi SonarLint

Dalla figura è possibile vedere alcuni tipi di segnalazioni effettuate a seguito di un'analisi di SonarLint, quali ad esempio import di librerie deprecate, blocchi di codice commentati, metodi che non rispettano le classiche convenzioni nominali di base, segnalazioni di possibile refactoring del codice, ecc.

5.2 Test di unità

Per quanto riguarda i test di unità, essi sono stati sviluppati mediante l'uso delle librerie del framework JUnit.
Anche per queste librerie, come per altre già citate in precedenza, l'integrazione è stata facilitata per mezzo del framework Spring.

L'azienda ha richiesto come requisito obbligatorio una copertura minima dell'80% del package di service per i test di unità.
Le API sono invece state testate manualmente tramite l'utilizzo di Postman e verranno discusse nel paragrafo successsivo.

Per i metodi testati sono stati scritti più test in modo da verificare tutte le possibilità di esecuzione del metodo stesso.

Tutti i metodi di test sono stati annotati con l'etichetta **@Test**, messa a disposizione da JUnit, e scritti secondo il pattern **Arrange Act Assert**.
Tale pattern spiegato in breve si divide nei tre semplici passi:

- **Arrange**: in questo passo si prepara l'ambiente di test, si inizializzano le variabili e si preparano i dati di input;
- **Act**: in questo passo si esegue il metodo da testare;
- **Assert**: in questo passo si verifica che il metodo abbia prodotto il risultato atteso.

Grazie all'etichetta `@Before` fornita da Junit è stato possibile semplificare la fase di Arrange per alcuni metodi di test.
Tale etichetta permette di eseguire un metodo prima dell'esecuzione di ogni metodo di test.
È stato dunque creato un metodo **init()** annotato con `@Before` che inizializza le variabili comuni necessarie per l'esecuzione di più test, così da snellire il codice e rendere

ciascun metodo di test meno verboso e quindi di più facile comprensione.

Per controllare la copertura dei test è stato utilizzato il plugin JaCoCo, che è stato integrato con Maven.
Nel seguito verranno mostrati degli estratti del documento in formato html prodotto da JaCoCo, che mostra la copertura dei test per il package `it.synclab.smartparking.service`:

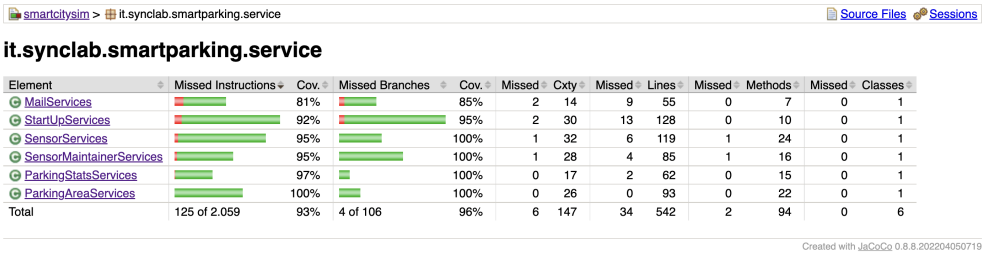


Figura 47: Copertura dei test per il package `it.synclab.smartparking.service`

Come è possibile notare dalla figura sopra riportata Jacoco riporta le percentuali delle istruzioni, dei branch e della copertura totale dei test effettuati per ciascuna classe all'interno del package.
La copertura totale del package di Service si attesta al 93%. Cliccando poi su una delle classi è possibile ottenere le stesse informazioni appena descritte per ogni singolo metodo della classe:

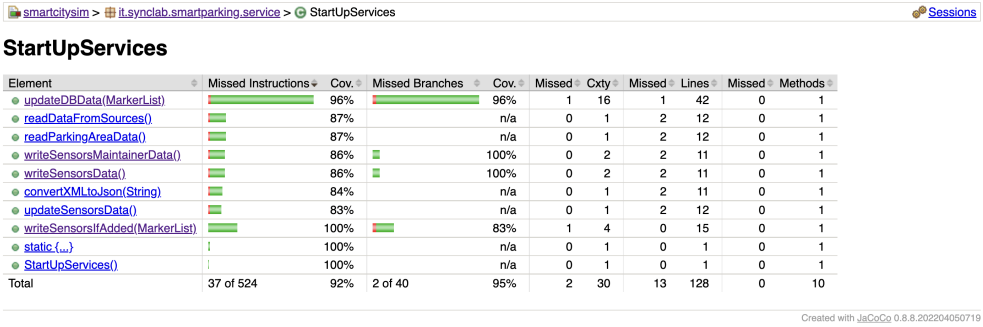


Figura 48: Copertura dei test per la classe `it.synclab.smartparking.service.StartUpService`

Cliccando infine su uno dei metodi è possibile ottenere le informazioni riguardanti la copertura dei singoli branch e istruzioni del metodo stesso:

```
200.    public void writeSensorsData() {
201.        logger.debug("SensorServices START writeSensorData");
202.        try {
203.            MarkerList sensors = readDataFromSources();
204.            for (Marker m : sensors.getMarkers().getMarkers()) {
205.                Sensor s = sensorServices.buildSensorFromMarker(m);
206.                sensorServices.saveSensorData(s);
207.            }
208.        } catch (Exception e) {
209.            logger.error("SensorServices - Error", e);
210.        }
211.        logger.debug("SensorServices END writeSensorData");
212.    }
213.
214.    public void writeSensorsIfAdded(MarkerList sensors) {
215.        int markerListSize = (sensors.getMarkers().markers.size());
216.        int sensorListFromDB = (sensorServices.getAllSensorsFromDB().size());
217.        logger.debug("Number of sensors from file:{}", markerListSize);
218.        logger.debug("Number of sensors from Data Base:{}", sensorListFromDB);
219.        if (sensorListFromDB == 0) {
220.            logger.debug("DB is empty.");
221.            writeSensorsData();
222.            writeSensorsMaintainerData();
223.        }
224.        else if (sensorListFromDB != markerListSize ) {
225.            logger.debug("there is a new sensor.");
226.            for (int i = sensorListFromDB; i < markerListSize; i++) {
227.                Marker marker = sensors.getMarkers().getMarkers().get(i);
228.                sensorServices.saveSensorData(sensorServices.buildSensorFromMarker(marker));
229.                sensorMaintainerServices.saveSensorsMaintainerData(sensorMaintainerServices.buildSensorsMaintainerFromMarker(marker)
230.            }
231.        }
232.    }
```

Figura 49: Esempio di copertura dei test per riga di codice e branch

Dalla figura è possibile notare che le linee di codice interne ai metodi vengono mostrate evidenziate in tre colorazioni diverse:

- **Verde**: indica che la linea di codice è stata eseguita in fase di test almeno una volta;
- **Giallo**: indica che la linea di codice è stata eseguita in fase di test almeno una volta, ma non sono stati coperti tutti i casi.
- **Rosso**: indica che la linea di codice non è stata mai eseguita in fase di test.

Grazie a JaCoCo è stato facile verificare la copertura dei test ed è quindi stato possibile verificare che il requisito richiesto dall'azienda fosse stato soddisfatto.

5.3 Test delle interfacce

Come precedentemente accennato, le interfacce sono state testate manualmente tramite l'utilizzo di Postman.

Postman è un software che permette di testare le API di un'applicazione web con funzionalità quali:

- Effettuare richieste HTTP e visualizzare il risultato ottenuto;
- Salvare le richieste effettuate singolarmente o in collezioni in modo da poterle ripetere in seguito;
- Eseguire il debug delle richieste effettuate.

In particolare è stata creata una collezione `SmartParking SET` contenente tutte le richieste effettuate per testare le api:

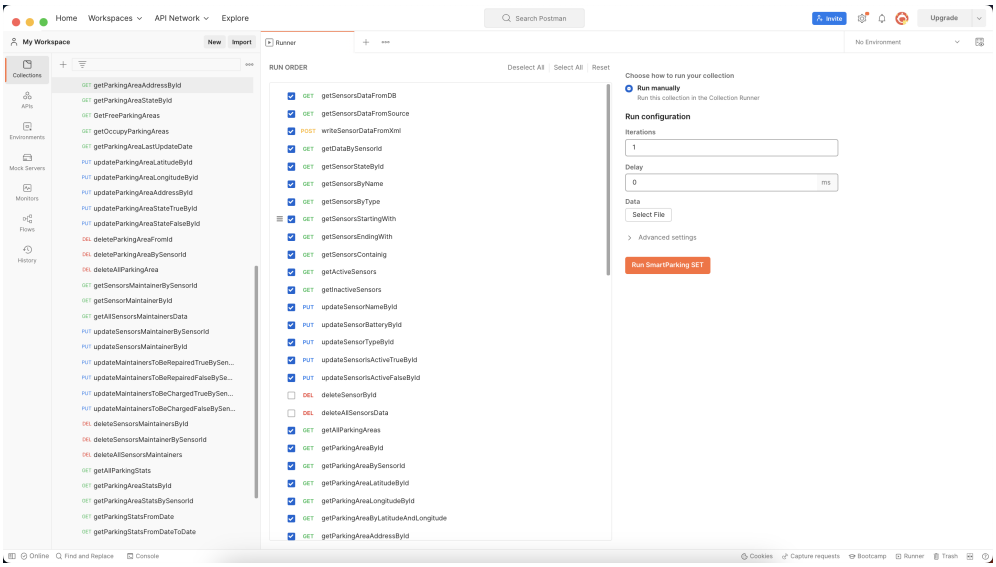


Figura 50: Estratto della collezione `SmartParking SET`

Tale collezione può essere eseguita in maniera automatica ed è possibile per ciascuna richiesta visualizzare informazioni quali il codice di stato della richiesta, il tempo di risposta, l'URL , il response body, ecc.

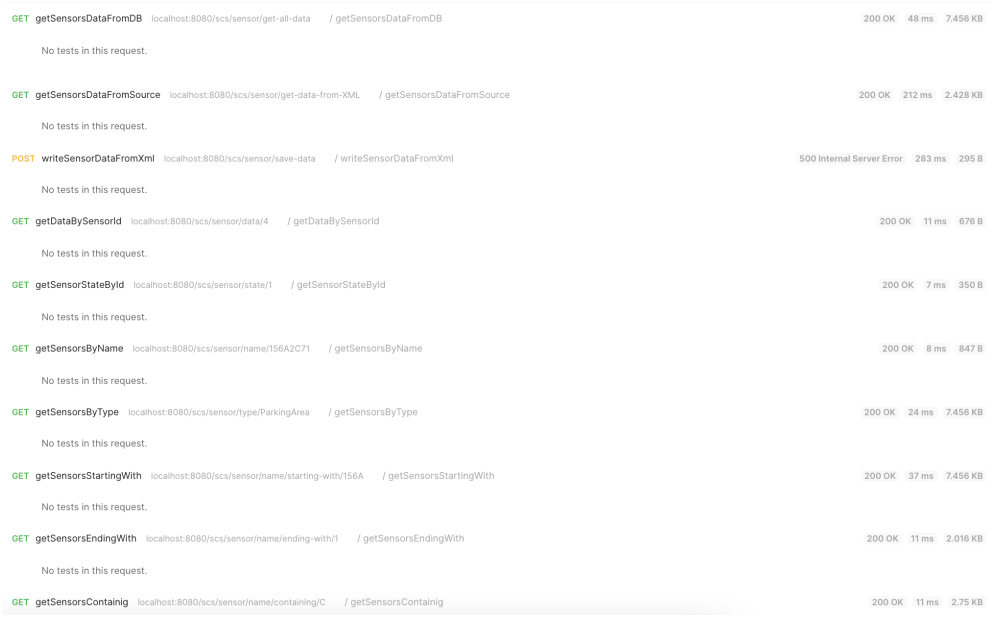


Figura 51: Esempio di esecuzione della collezione `SmartParking SET`

Figura 52: Dettaglio esecuzione collection

Per mezzo di tale collection è stato possibile testare che tutte le API funzionassero correttamente a seguito di ogni modifica effettuata al codice e, nei casi in cui questo non fosse vero, di individuare il problema e correggerlo.

5.4 Validazione requisiti

Al termine dello stage la copertura dei requisiti, considerando solo la parte backend della soluzione software è quasi totale.
Per visualizzare l'elenco completo dei requisiti si rimanda al Paragrafo 3.4.

Tipologia	Coperti	Totale	Percentuale
Funzionali	34	34	100%
Qualitativi	4	5	80%
Di vincolo	1	2	50%
Di sicurezza	1	1	100%
Totale	40	42	95%

Tabella 5.1: Copertura dei requisiti

Come è possibile notare dalla tabella, due requisiti non sono stati coperti.
In particolare si parla dei requisiti individuati dai codici **RQ2** e **RV2**.

RQ2

Tale requisito è stato catalogato come **desiderabile** e richiede una copertura dei test di service pari al 100%.
Il requisito non è stato soddisfatto per mancanza di tempo.

RV2

Tale requisito è stato catalogato come **opzionale** e richiede di containerizzare l'applicativo mediante l'uso di Docker.
Il requisito non è stato soddisfatto per mancanza di tempo.
Già dall'inizio dello stage era stato messo in chiaro dagli stakeholders che questo requisito è stato inserito per ovviare al caso in cui il tempo a disposizione fosse stato più di quello richiesto per lo sviluppo dell'applicativo.

Capitolo 6

Conclusioni

Il prodotto ottenuto alla fine dello stage è composto da un microservizio che espone API REST utilizzabili da client esterni autorizzati e da un microservizio in grado di gestire l'invio delle email mediante la lettura dei dati da una coda JMS.

6.1 Raggiungimento degli obiettivi

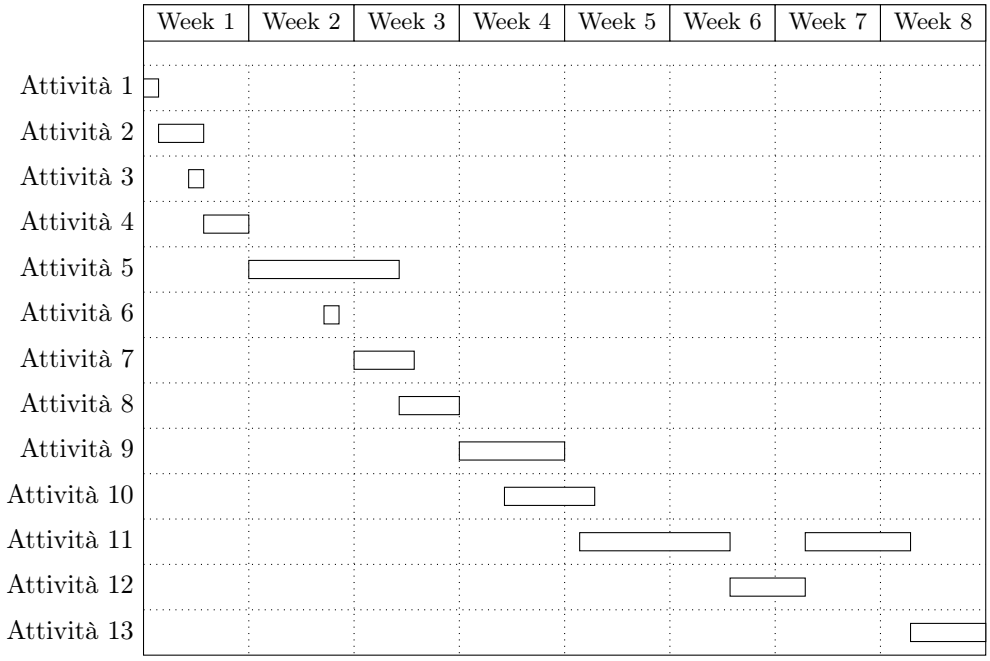
Il raggiungimento degli obiettivi, indicati alla sezione 2, è da ritenere soddisfacente. Lo sviluppo di microservizi mediante il framework Spring mi ha permesso di ottenere competenze su questi ultimi. L'analisi funzionale e l'analisi dei requisiti realizzate sono state ritenute più che soddisfacenti dall'azienda. La fase di progettazione delle API mi ha permesso di prendere familiarità con il protocollo REST e con Postman, strumento molto utilizzato nel mondo del lavoro. La fase di sviluppo del primo microservizio è stata realizzata in autonomia, senza riscontrare grosse criticità. Questo mi ha permesso di realizzare un secondo microservizio e di fare comunicare i due mediante l'uso di code JMS implementate in ActiveMQ.

Codice	Descrizione	Stato
O-01	Acquisizione competenze Spring Java, microservizi e REST	Soddisfatto
O-02	Capacità di raggiungere gli obbiettivi richiesti in autonomia seguendo il cronoprogramma	Soddisfatto
O-03	Portare a termine l'implementazione dei microservizi richiesti con una percentuale di superamento pari a 80	Soddisfatto
D-01	Portare a termine l'implementazione dei microservizi richiesti con una percentuale di superamento pari a 100	Soddisfatto
F-01	Utilizzo della containerizzazione per portare tutti i microservizi su Docker.	Non soddisfatto

Tabella 6.1: Copertura dei requisiti

6.2 Consuntivo finale

Lo stage ha avuto una durata di 320 ore suddivise nell'arco temporale di 8 settimane. Il consuntivo finale è riportato nel diagramma di Gantt seguente:



Dove le attività sono le seguenti:

Attività 1: Formazione metodo aziendale;

Attività 2: Formazione Java;

Attività 3: Ripasso principi di buona programmazione;

Attività 4: Formazione microservizi;

Attività 5: Formazione Spring;

Attività 6: Formazione ORM;

Attività 7: Formazione servizi REST;

Attività 8: Formazione broker e sistemi producer-consumer;

Attività 9: Analisi implementazione attuale e cambio;

Attività 10: Stesura documentazione relativa ad analisi e progettazione;

Attività 11: Implementazione modulo di raccolta dati dai sensori e dei servizi per l'esposizione dei dati al frontend;

Attività 12: Implementazione test di integrazione;

Attività 13: Stesura documentazione finale.

6.3 Conoscenze acquisite

Durante l'esperienza di stage ho appreso diverse tecnologie e strumenti che ritengo mi risulteranno utili anche in futuro.

La progettazione di API REST mi ha permesso di approfondire il funzionamento del protocollo HTTP.

Grazie alla gestione delle mail tramite code JMS ho potuto apprendere il funzionamento dei broker di messaggi potendo sperimentare l'utilizzo di ActiveMQ.

Ho poi maturato una discreta esperienza con il framework Spring che risulta ad oggi molto popolare in ambito lavorativo.
Grazie allo studio e all'utilizzo di tale framework ho inoltre consolidato le mie conoscenze sul linguaggio Java, sulla programmazione a oggetti e sui microservizi.

6.4 Valutazione personale

L'esperienza di stage è stata particolarmente istruttiva e mi ha permesso di avere una prima visione del mondo del lavoro.

Il progetto è stato portato a termine con successo. Il tutor e l'azienda si sono detti soddisfatti. Il carico di lavoro assegnatomi è stato perfettamente pesato per le 320 ore a disposizione.

Il tutor Daniele Zorzi si è sempre dimostrato molto disponibile per ogni tipo di esigenza o chiarimento.

Nel complesso penso che le numerose librerie e gli strumenti utilizzati abbiano molto arricchito il mio bagaglio culturale.
Personalmente mi ritengo molto soddisfatto del lavoro svolto e ho molto apprezzato la possibilità che mi è stata fornita dal corso di Laurea di relazionarmi con il mondo del lavoro.

Acronimi

API Application Programming Interface. 4, 15, 39, 46, 48, 50, 51, 53, 54, 56–58, 60, 64, 67, 69

DB Base di dati. 4, 7, 36–38, 43

HTTP Hyper Text Transfer Protocol. 7, 15–35, 42, 62, 69

IT Information Technology. 3

JMS Java Message Service. 7, 39, 48, 67, 69

JSON JavaScript Object Notation. 7, 10, 36, 47, 49, 51, 54

ORM Object-Relational Mapping. 11, 42, 68

REST REpresentational State Transfer. 4, 5, 9, 11, 15, 39, 46, 48, 57, 58, 67–69

XML EXtensible Markup Language. 4, 10, 36, 42, 47

Glossario

API_KEY È una chiave di sicurezza che viene fornita da un servizio per permettere l'accesso alle sue API. 39, 56

Application Programming Interface (It. Interfaccia di programmazione di applicazione) è una serie di protocolli che consentono di comunicare tra due applicazioni. 71

Backend in informatica col termine backend ci si riferisce alla parte non visibile dell'applicazione che gestisce la logica di dominio. 3–5, 7, 64

Base di dati (En. Database) In informatica si indica un insieme di dati strutturati, ovvero omogeneo per contenuti e formato, memorizzati in un computer, rappresentando di fatto la versione digitale di un archivio dati. 4, 7, 16, 17, 38, 43, 71

Boilerplate È un termine inglese che indica un frammento di codice che viene ripetuto in molte parti di un'applicazione con piccole modifiche o addirittura assenti. 42

Broker È un componente software che si occupa di gestire la comunicazione tra i client e i server di messaggistica. 12, 42, 69

Bug È un errore di programmazione che causa un comportamento anomalo di un programma. 59

Client Indica genericamente un qualunque componente software, presente tipicamente su una macchina host, che accede ai servizi o alle risorse di un'altra componente detta server, attraverso l'uso di determinati protocolli di comunicazione. 7, 8, 15–35, 57, 67

CRUD In informatica, Create, Read, Update, e Delete (in it. creazione, lettura, aggiornamento e rimozione) sono le quattro operazioni basilari della gestione persistente dei dati. 4, 7, 42, 46

Diagramma Entità-Relazione È un modello teorico per la rappresentazione concettuale e grafica dei dati a un alto livello di astrazione. 43

Discord è un'applicazione di di VoIP, messaggistica istantanea e distribuzione digitale progettata inizialmente per la comunicazione tra comunità di videogiocatori. Gli utenti comunicano con chiamate vocali, videochiamate, messaggi di testo, media e file in chat private o come membri di un server Discord. 12, 42

Docker È un software open source progettato per eseguire processi informatici in ambienti isolabili, minimali e facilmente distribuibili chiamati container Linux, con l'obiettivo di semplificare i processi di deployment di applicazioni software. 10, 39, 65, 67

Endpoint è un tipo di nodo per la comunicazione in rete. 4, 7, 46

EXtensible Markup Language È un metalinguaggio per la definizione di linguaggi di markup, ovvero un linguaggio basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo. 71

Framework è un'architettura logica di supporto sulla quale un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore. 7, 8, 10, 11, 39, 41, 42, 44, 47, 56, 57, 60, 67, 69

in informatica col termine frontend ci si riferisce alla parte visibile dell'applicazione che viene esposta all'utente per gestire l'interazione con esso. 4, 5, 12, 54, 56, 68, 74

HTTP È un protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server. Le specifiche del protocollo sono gestite dal W3C. 71

Information Technology termine inglese per informatica. 71

Java è un linguaggio di programmazione a oggetti. 7, 8, 10, 11, 39, 41, 42, 46, 47, 67–69

Java Message Service È l'insieme di API, appartenente a Java EE, che consente ad applicazioni Java presenti in una rete di scambiarsi messaggi tra loro. 48, 71

JavaScript Object Notation Nell'ambito della programmazione web è un formato adatto all'interscambio di dati fra applicazioni client/server facilmente leggibile sia da umani che da macchine.. 71

Jetty È un server web open source scritto in Java. È stato originariamente sviluppato per supportare il framework web Java Servlet, ma supporta anche altri framework web, come Apache Struts, Apache Tapestry, Apache Wicket, JSF, Spring, ecc. 42

Microservizi Un'architettura di microservizi organizza un'applicazione come una raccolta di servizi ad accoppiamento libero. In un'architettura di microservizi, i servizi sono a grana fine e i protocolli sono leggeri. 4, 5, 7, 9–11, 44, 45, 67–69

Object-Relational Mapping È un'architettura software che consente di mappare oggetti di un'applicazione ad oggetti di un database relazionale. 71

Package È un insieme di file che contiene codice, dati e altre risorse necessarie per la creazione di un'applicazione o di un componente software. 46, 59–61

Plugin È un componente software che estende le funzionalità di un'applicazione. 42, 59, 61

Polling È un metodo di comunicazione tra due o più dispositivi che si basa sullo scambio di messaggi. Il polling è un metodo di comunicazione sincrona, ovvero il dispositivo che invia il messaggio attende la risposta del dispositivo destinatario prima di inviare un nuovo messaggio. 7, 10, 47

REpresentational State Transfer è uno stile architetturale per sistemi distribuiti basato sul protocollo HTTP. 71

Spring Initializr È un servizio web che consente di generare un progetto Spring Boot in modo rapido e semplice. 42

Stakeholder È un termine inglese che indica un soggetto che ha interesse in un progetto, un prodotto o un servizio. In un progetto di sviluppo software, gli stakeholders sono le persone che hanno interesse nel prodotto finale. 4, 12, 43, 59, 65

Tomcat È un server web open source, scritto in Java, che implementa il protocollo HTTP e il protocollo Servlet, ed è utilizzato per eseguire applicazioni web scritte in Java. È uno dei più popolari server web open source al mondo. 42

Vulnerabilità È una debolezza di un sistema informatico che può essere sfruttata per violare la sicurezza del sistema stesso. 59

Bibliografia

Siti web consultati

Documentazione ActiveMQ. URL: https://activemq.apache.org/components/classic/documentation.

Documentazione MySQL. URL: https://dev.mysql.com/doc.

Documentazione PostgreSQL. URL: https://www.postgresql.org/docs/current/index.html.

Documentazione Spring Boot. URL: https://docs.spring.io/spring-boot/docs/2.7.2/reference/htmlsingle.

Documentazione Spring Data JPA. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html.

Guida RESTful web services. URL: https://www.html.it/guide/restful-web-services-la-guida.

Maven Repository. URL: https://mvnrepository.com.

Sito Baeldung. URL: https://www.baeldung.com.

Sito Spring. URL: http://spring.io.

Sito Stack Overflow. URL: http://stackoverflow.com.

Sito Sync Lab. URL: https://www.synclab.it (cit. a p. 3).

Spring Initializr. URL: https://start.spring.io.