UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Elaborazione di dati ambientali mediante framework Spring Batch in una applicazione di monitoraggio dei parcheggi

Relatore:
prof. Fantozzi Carlo

Laureando:
Andrei Ovidiu Danciu

Correlatore:
dott. Fabio Pallaro

ANNO ACCADEMICO: 2022/2023
Data di laurea: 14 Novembre 2022

Sommario

La relazione descrive le attività svolte durante il periodo di tirocinio a Sync Lab dal 05/09/2022 al 19/10/2022 per una durata di 225 ore. Il percorso si è articolato in una prima parte dedicata a studio teorico e pratico (architettura a microservizi, moduli del framework Java Spring come Spring Boot, Spring Data e Spring Batch) seguita dall'analisi del progetto Smart city simulator, con progettazione e sviluppo attraverso Spring Batch di un servizio di elaborazione e salvataggio dati in un contesto di smart parking.

Indice

Capitolo 1

Introduzione

Organizzazione del testo

Per quanto riguarda la stesura del seguente testo, sono state applicate le seguenti convenzioni tipografiche:

- gli acronimi e i termini tecnici o ambigui vengono definiti nel glossario, collocato alla fine della relazione;
- i termini in lingua inglese o appartenenti al gergo tecnico sono evidenziati con il carattere *corsivo*;
- i termini che fanno riferimento ad elementi del codice sono evidenziati con il carattere ***grassetto e corsivo***

1.1 L'azienda

Sync Lab [10] nasce nel 2002 a Napoli come *software house*. L'azienda, propone sul mercato interessanti quanto innovativi prodotti *software*, nati nel loro laboratorio di ricerca e sviluppo. A seguito di una maturazione delle competenze tecnologiche, metodologiche ed applicative nel dominio del *software*, Sync Lab è riuscita rapidamente a trasformarsi in *System Integrator*, conquistando significative fette di mercato nei settori mobile, videosorveglianza e sicurezza delle infrastrutture informatiche aziendali. Ad oggi conta oltre 300 dipendenti, suddivisi tra le 6 sedi in Italia: Roma, Napoli, Milano, Verona, Padova e Como.

Figura 1.1: Logo azienda Sync Lab. Fonte: [10]

1.2 Il percorso di tirocinio

Il percorso di tirocinio da me intrapreso si è articolato in due fasi. Il primo periodo era dedicato principalmente allo studio e all'applicazione dei concetti necessari per poter poi lavorare sul progetto che mi era stato proposto. Ho iniziato da un ripasso teorico di

argomenti già trattati in precedenza all'università, come il linguggio *Java*, i protocolli di rete *HTTP*, il lavoro seguendo i principi di Agile e SCRUM, HTML5 e CSS3. In seguito, ho studiato anche concetti nuovi come i servizi REST e i messaggi in formato JSON, le differenze tra *architettura a monolite* e *a microservizi* ed infine le basi di *Spring* e *Spring Core*. Ho avuto modo di affrontare in maniera più pratica invece i moduli *Spring Data* e *Spring Batch*, sviluppando anche degli applicativi in parallelo allo studio delle tecnologie che stanno dietro.

Una volta concluso il percorso di formazione, ho iniziato a lavorare sul progetto di smart parking. Innanzitutto, prima dell'inizio del tirocino, mi era stata proposta la sceltra tra un progetto di sviluppo di un front end con il framework Angular oppure di un back end con il framework *Java Spring*. Ho optato per il secondo siccome avevo conoscenze pregresse di Java e mi interessava di più approfondire il processo di creazione di un back end. In più, mi ero trovato soddisfatto per quanto riguarda il modo di ragionare richiesto per sviluppare progetti, anche complessi, in *Java* durante i corsi universitari. Il lavoro si è articolato in una fase di analisi e progettazione assieme al responsabile del progetto, seguito da implementazione e test più in autonomia, con eventuali rivalutazioni dei requisiti iniziali in base ai problemi riscontrati. Sono risultati particolarmente utili i prototipi sviluppati le prime settimane, in quanto sono stati dei buoni punti di partenza nel momento in cui ho inziato a dedicarmi effetivamente allo sviluppo di codice per il progetto finale del tirocinio.

1.3 Il progetto SmartCitySimulator

1.3.1 Il concetto di Smart City

Il progetto *SmartCitySimulator*[5] di Sync Lab è un simulatore misto di una *smart city*. In generale, per *smart city*, si intende un'area urbana tecnologicamente moderna che fa uso di tecnologie e sensori per raccogliere una serie di dati rilevanti. Questi dati verranno poi analizzati ed elaborati per poter gestire risorse e servizi in maniera più efficiente. In specifico, vengono integrati concetti di Information and Communication Technologies (ICT), assieme a dispositivi fisici collegati all'IoT, per valutare possibili ottimizzazioni della città. Ad esempio, l'applicazione di strumenti come l'*intelligenza artificiale* e l'*analisi computazionale di dati* può essere sfruttata per valutare modi più efficienti di utilizzare e gestire l'infrastruttura fisica della città, portando a costi e consumo minori. Oltre a questo, le tecnologie permettono il collegamento ai dispositivi dei cittadini e dei visitatori, offrendo accesso libero e diretto ai servizi proposti dalla *smart city*. In questo modo, il sistema punta ad un miglioramento della qualità di vita, lavorando su aspetti come la mobilità, la distribuzione di informazioni e conoscenza ed in generale il supporto fornito alle persone bisognose.

Un servizio particolarmente rilevante è lo *smart parking* [7]. Questo sistema combina la tecnologia con l'innovazione umana, nel tentativo di utilizzare il minor numero di risorse possibili (carburante, tempo, spazio) per ottenere un parcheggio di veicoli più veloce, facile e ottimizzato durante il periodo in cui queste vetture restano inutilizzate. Il sistema di smart parking consiste in sensori, raccolta dati in tempo reale e *analytics*, e in sistemi di pagamento automatico che consentono alle persone di trovare parcheggio nel luogo desiderato e di versare la somma dovuta in anticipo. In particolare, per individuare gli spazi disponibili, si possono installare sensori nel terreno, o installare telecamere sui pali della luce o sulle mura degli edifici. Un software di riconoscimento di immagini traduce le immagini riprese dalle telecamere pre-installate in dati relativi all'occupazione del suolo. In questo modo i sensori sono in grado di individuare la presenza di veicoli all'interno di un parcheggio. I dati vengono poi inviati via *wireless* a un *gateway* e rilasciati su una

piattaforma di *smart parking* sul *cloud*. Aggregati con altri dati relativi ai parcheggi, contribuiscono a formare una mappa complessiva dei parcheggi in tempo reale.

1.3.2 Smart city simulator

Il progetto *Smart city simulator*[5] sfrutta, in primo luogo sensori, in grado di misurare la temperatura, l'umidità e le polveri sottili, sia pm2.5 che pm10, per determinare le condizioni dell'ambiente e la qualità dell'aria all'interno del parcheggio. Al momento, solamente i sensori per le polveri sottili sono attivi e riportano i dati all'interno del seguente sito [11]. Per questo motivo, i dati utilizzati come *input* in questa fase di sviluppo, sono misure di temperatura, umidità e polveri sottili effettuate tra le ore 15:00 del giorno 25/03/2022 e le ore 16:00 del giorno 28/03/2022. La frequenza di campionamento corrispondeva ad una misura ogni ora, offrendo 72 diversi orari in cui sono state salvate le misure ambientali.

Produttore	Nova PM	
Modello	SDS011	
Alimentazione	5 Vcc	(min. 4,7 Vcc - max 5,3 Vcc)
Gamma di misura	da 0,0 µg/m3 a 999,9 µg/m3	
Risoluzione	0.3 µg/m3	

Figura 1.2: Esempio specifiche di sensore pm10 e pm2.5. Fonte: [11]

Assieme ai sensori ambientali, verranno utilizzati anche sensori installati sulle piazzole di parcheggio e comunicheranno con un ricevitore per determinare la presenza o meno di auto e inviare tale informazione. Al momento, questi sensori non sono stati installati, ma sono simulati ed i loro dati vengono trasmessi attraverso un *file* in *formato XML*. All'interno del *file* di dati utilizzato durante il tirocinio, vengono riportati 15 sensori di parcheggio con le loro informazioni, tra cui il loro id, le ipotetiche coordinate assieme all'indirizzo, carica e tipo di batteria, se il sensore è attivo o meno e se è stata rilevata la presenza di un'auto o meno.

TECHNICAL INFO	PARAMETERS
DETECTION METHOD	MAGNETIC + SPECIAL IR
DIMENSIONS	Φ185MM / H38MM
COMMUNICATION RANGE	200M
DETECTION SPEED	10 S
ACCURACY	95-98%
PROTECTION GRADE	IP67
OPERATING TEMPERATURE	-40℃ - +85℃
POWER SUPPLY	BUILD IN BATTERY

Figura 1.3: Esempio specifiche di sensore di parcheggio

TECHNICAL INFO	PARAMETERS
WIRELESS RADIO FREQUENCY	433/915/868MHz
COMMUNICATION RANGE	200M
DIMENSIONS	L230 / W100 / H49MM
DEBUG INTERFACE	USB
DATA INTERFACE	RS232/485/ ETHERNET
NETWORK CAPACITY	50
POWER SUPPLY	DC 9V - 36V

Figura 1.4: Esempio specifiche del ricevitore dei sensori di parcheggio

I dati forniti dai sensori verranno elaborati e manipolati da *servizi web* per fornire agli utenti in tempo reale informazioni sullo stato dei sensori. In particolare, ci sarà un back end che, inizialmente, gestirà la lettura e scrittura delle informazioni per salvarle in una base dati e fare in modo che queste vengano costantemente aggiornate. In futuro verrà implementata anche un'elaborazione dei dati per creare uno storico e delle indagini statistiche sull'andamento settimanale. Una volta impostato il sistema per manipolare questi dati, questo verrà collegato ad un front end in grado di comunicare col back end ed accessibile liberamente dagli utenti. Il front end dovrà comunicare le misure rilevate dai sensori attraverso un'interfaccia dinamica, mappa con le posizioni dei sensori ed una tabella riassuntiva con le misure e posizioni di ciascun sensore.

Figura 1.5: Esempio interfaccia front end con mappa. Fonte: [6]

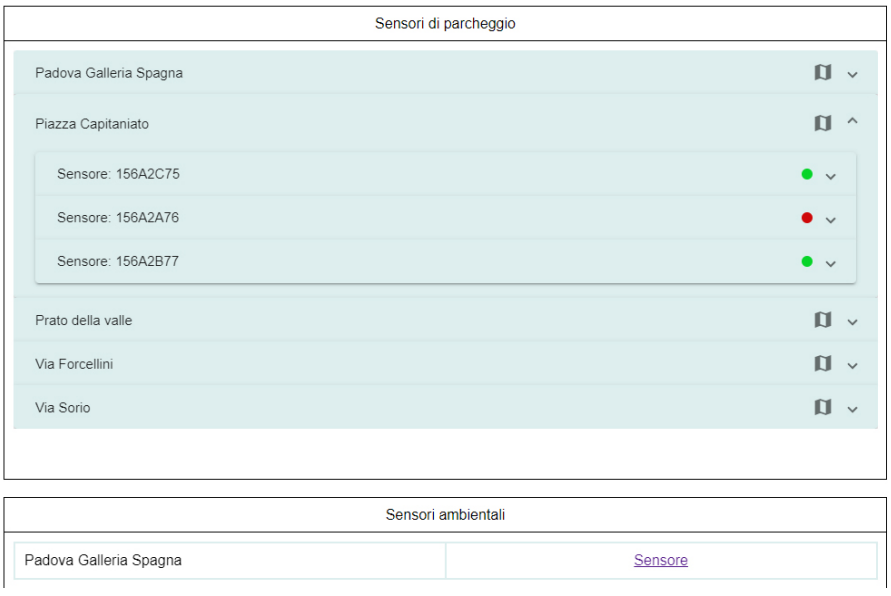

Figura 1.6: Esempio interfaccia front end con tabella sensori. Fonte: [6]

Il front end dovrà essere accessibile al seguente sito [6] e, eventualmente in futuro, attraverso un'applicazione per telefono. Infine, verrà creato un sistema di accesso per i manuntentori dei sensori per poter accedere ad un'interfaccia con ulteriori informazioni sui sensori, come il livello di carica, id e se il sensore è attivo o meno.

Per realizzare tutto ciò, sono stati inizialmente individuati i seguenti requisiti:

- persistenza dei dati dei sensori, delle misure e del parcheggio all'interno di una base dati;
- tecnologia per la lettura periodica dei dati dagli appositi file generati dai sensori
- creazione di API REST per le operazioni CRUD sulla base di dati;
- realizzazione di un front end reattivo;
- creazione di una sezione utente ed una amministratore con privilegi di visibilità opportuni;

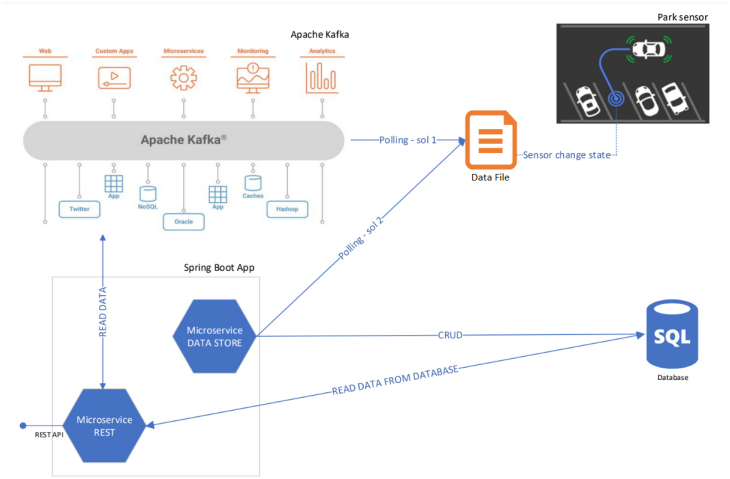

Figura 1.7: Prima proposta di schema architetturale del back end

In relazione alle tecnologie utilizzate, il front end verrà sviluppato con il framework Angular, la base dati sarà di tipo *relazionale SQL* con *PostgreSQL* o *MySQL*, mentre il back end farà uso del framework *Java Spring*, in particolare il modulo *Spring Batch* per la lettura periodica ed elaborazione delle informazioni, *Spring Data Rest* per gestire la persistenza dei dati e gli endpoint REST API e *Spring Boot* per lanciare i vari servizi.

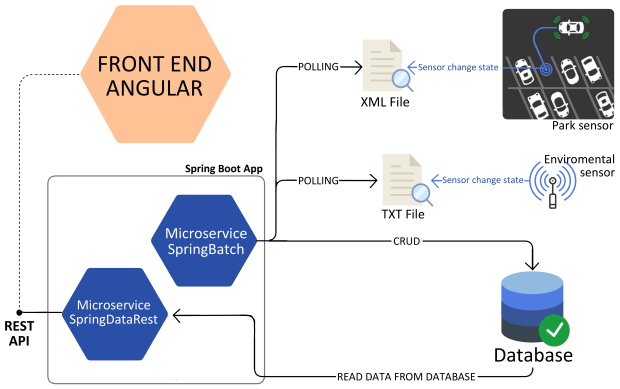

Figura 1.8: Progettazione implementazioni durante la fase di sviluppo

Ulteriori sviluppi includono l'impostazione di un'ulteriore base dati NoSql e la creazione di una versione del back end con il framework Node.js. Questo renderà possibile un'analisi sulla differenza di struttura, di prestazioni e di verbosità del codice rispetto a all'implementazione con *Spring*.

1.4 Strumenti e Tecnologie utilizzati

I primi strumenti hanno contribuito alla comunicazione e pianificazione del percorso di tirocinio

- **Discord**: piattaforma di messaggistica istantanea e distribuzione di materiale digitale, progettata per la comunicazione attraverso chiamate vocali, videochiamate e messaggi testuali in *chat* private o come membri di un *server* comune *Discord*. All'interno del *server Discord* di Sync Lab, vi era la possibilità di accedere a canali testuali con già presenti *link* a *pagine web* utili alla formazione, leggere domande e risposte già poste in passato e presentare dubbi relativi al proprio percorso.

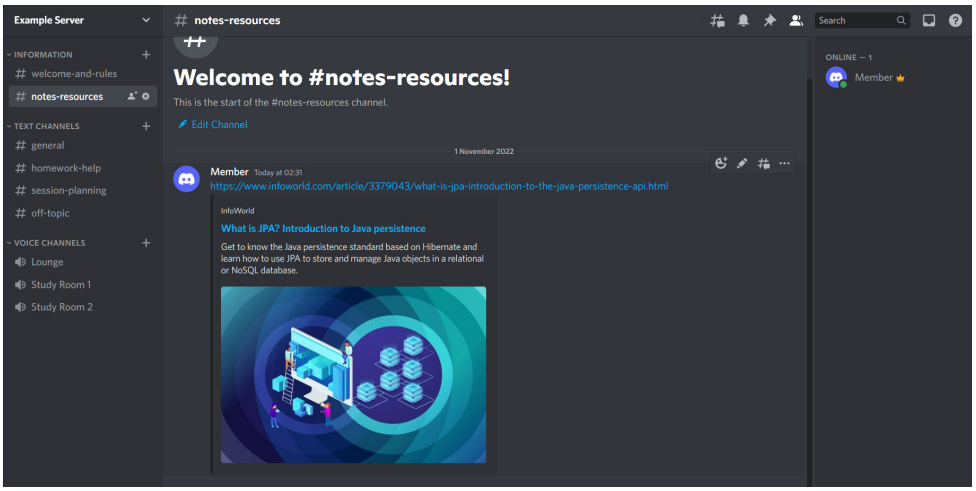

Figura 1.9: Esempio di server Discord

- **Trello**: applicazione web in stile Kanban, con l'obiettivo di suddividere le attività settimanali del percorso di tirocinio e facilitare l'analisi del progresso fatto ogni settimana.

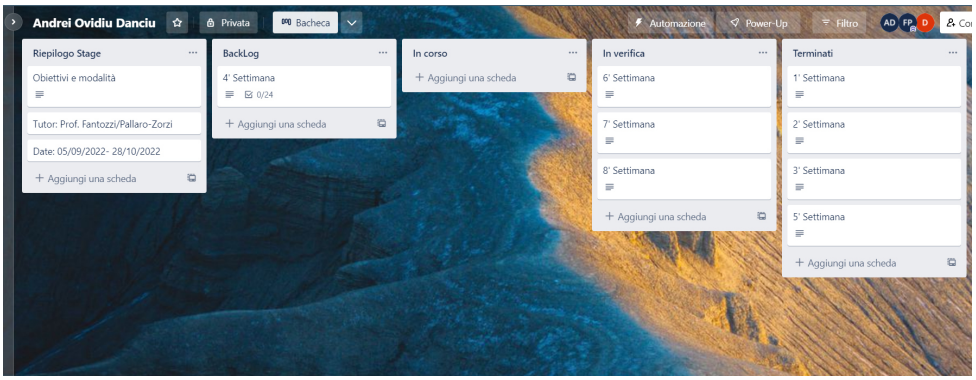

Figura 1.10: Esempio di bacheca Trello

- **Google calendar**: servizio di organizzazione delle attività e programmazione dei giorni in cui sono presenti persone nell'ufficio dell'azienda. Questo è stato particolarmente rilevante nel mio caso, in quanto mi ha permesso di coordinare facilmente le giornate con il tutor aziendale. Tutti i dipendenti della sede hanno accesso al calendario Sync Lab condiviso ed è richiesto a loro di aggiungere la presenza in corrispondenza dei giorni che hanno intenzione di lavorare in ufficio.

Gli strumenti seguenti sono, invece, stati utilizzati per gestire e facilitare la creazione e lo sviluppo dei prototipi e del progetto durante il tirocinio:

- **SpringToolSuite4**: IDE basato su Eclipse, con tutte le impostazioni ed estensioni installate per poter immediatamente sviluppare progetti basati su *Spring*.
- **Postman**[4]: piattaforma API che fornisce ausilio nello sviluppo con strumenti per testare altre API. attraverso il lancio di richieste personalizzate. Ho utilizziato il *client desktop*, assieme alla *piattaforma web* per poter lanciare richieste personalizzate verso gli endpoint REST impostati attraver *Spring REST* e *Spring Data REST* coi prototipi durante la fase di formazione.

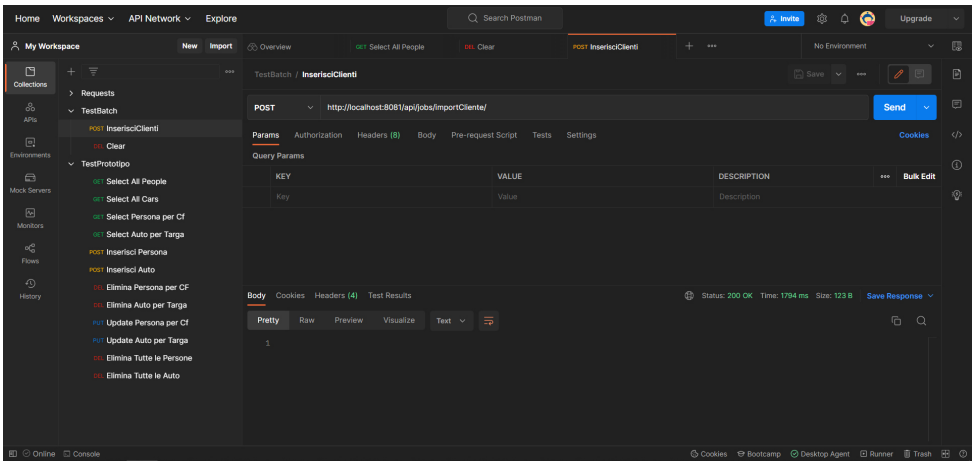

Figura 1.11: Interfaccia di Postman

- **Maven**[3]: strumento per standardizzare la gestione di progetti Java, fornire ulteriori strumenti di analisi ed invitare gli sviluppatori a seguire le *best practices*. *Maven* lavora con un elemento chiamato Project Object Model (POM), ossia la rappresentazione in *formato XML* del progetto. All'interno del file *pom.xml* si trovano tutte le caratteristiche utili del progetto ed, in particolare, la lista di dipendenze da altre librerie, progetti o framework necessari per il progetto. Infine, *Maven* si occupa in automatico di scaricare ed implementare gli elementi inseriti in questa lista.

```
1     <?xml version="1.0" encoding="UTF-8"?>
2 <project ...>
3     ...
4     <name>smartcitysim</name>
5     <description>Smart parking API</description>
6     <properties>
7      <java.version>17</java.version>
8     </properties>
9     <dependencies>
10     <dependency>
11       ...
12     </dependency>
13        ...
14    </dependencies>
15    ...
16 </project>
```

Listing 1.1: Esempio del formato di un file pom.xml

- **PostgreSQL**: DBMS ad oggetti utilizzato assieme ai moduli *Spring*, in particolare *Spring Data JPA*, e strumenti come *Psql* e *PgAdmin4* per accedere ai dati all'interno del database ed analizzare gli effetti del progetto sui dati da registrare o pre-esistenti. *Psql* è un terminale interattivo con comandi disponibili e supporto per interrogazioni in *linguaggio SQL* che permettono di interagire direttamente con le basi dati e gli elementi al loro interno. *PgAmin4* è un *software* di amministrazione di basi dati con strumenti per poter scrivere e lanciare *interrogazioni in SQL*, ma anche un'interfaccia grafica per interagire direttamente, ad esempio, con tabelle ed i dati all'interno di queste.

Capitolo 2

Framework Spring

2.1 Spring

Spring[8] è un *framework open source* per lo sviluppo di applicazioni su piattaforma *Java*. *Spring* lascia ampia libertà al programmatore, fornendo allo stesso tempo una vasta e ben documentata gamma di soluzioni semplici adatte alle problematiche più comuni. Al suo inteno, si possono identificare altri progetti come *Spring Boot*, *Spring Data* e *Spring Batch*, che aggiungono funzionalità specifiche.

2.1.1 Spring Core

Uno dei principi fondamentali implementati da *Spring*, è l'Inversion of Control (IoC) [1]: un *pattern* dell'ingegneria del software che trasferisce il controllo di oggetti o porzioni di programma ad un *container* o framework. A differenza della programmazione tradizionale, dove il codice del programmatore chiama una libreria, IoC permette ad un framework di prendere il controllo del programma e fare chiamate al codice. Se si volesse implementare un comportamento specifico, risulterebbe necessario estendere le classi proposte dal framework.

Questa tipologia di architettura porta con sè un serie di vantaggi, in particolare:

- permette di disaccoppiare l'esecuzione di una *task* dalla sua implementazione,
- semplifica il passaggio da un'implementazione verso un'altra,
- rende il programma più modulare,
- facilita i *test* isolando i diversi componenti o le dipendenze a loro associate.

Il framework *Spring* implementa la IoC attraverso una delle sue Tecnologie Chiave (in inglese, *Core Tecnologies*): la Dependency Injection (DI). Questa permette di invertire il controllo attraverso delle dipendenze che vengono inserite (in inglese, *injected*) negli oggetti.

```
1  public class Store {
2      private Item item;
3
4      public Store() {
5          item = new ItemImpl1();
6      }
7  }
```

Listing 2.1: Esempio di dipendenza nella programmazione tradizionale

```
public class Store {
    private Item item;

    public Store(Item item) {
        this.item = item;
    }
}
```

Listing 2.2: Esempio di dipendenza con la Dependency Injection

All'interno del framework *Spring*, queste dipendenze vengono gestite dall'interfaccia *pplicationContext* che rappresenta lo *Spring IoC Container*. Questo elemento, attraverso delle annotazioni specificate nel codice, è in grado di instanziare e configurare oggetti chiamati *beans*, oltre a gestire il loro ciclo di vita. Per implementare la Dependency Injection si possono sfruttare le seguenti tecniche:

- **Dependency Injection attraverso il costruttore**: solitamente gestito da una classe di *configurazione*, le dipendenze vengono impostate attraverso chiamate allo *Spring IoC Container* dei costruttori con gli appositi argomenti.

```
@Configuration
public class AppConfig {
    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

Listing 2.3: Esempio di Dependency Injection attraverso costruttore

L'annotazione *@Configuration* indica che la classe è una fonte di definizioni di *bean*. Si può aggiungere l'annotazione *@Bean* su di un metodo per definire un *bean*.

- **Dependency Injection attraverso l'autowiring**: aggiungendo la notazione *@Autowired*, se non vi è presente altro modo per inserire il *bean*, questo verrà inserito, grazie all'*IoC Container di Spring*, nella classe sfruttando il meccanismo delle *reflection*.

```
public class Store {
    @Autowired
    private Item item;
}
```

Listing 2.4: Esempio di Dependency Injection attraverso *autowiring*

2.1.2 Spring Boot

Il modulo *Spring Boot*[8] permette di creare un'applicazione *web* basata su *Spring* in maniera molto semplice, infatti non richiede di un *server web*, come *Tomcat*, e propone diversi modi per aggiungere dipendenze e librerie in caso di necessità.

2.2 Spring Data

Il modulo *Spring Data*[8] ha l'obiettivo di fornire un modello di programmazione basato su *Spring* coerente, per quanto riguarda la gestione di dati, e che tiene conto dei sistemi in cui i dati vengono salvati. *Spring Data* comprende al suo interno diversi sotto-progetti, ciascuno specializzato per le tipologie di tecnologie con cui devono interagire. I moduli più rilevanti per il mio percorso di tirocinio sono stati *Spring Data JPA* e *Spring Data REST*.

2.2.1 JPA

Un Jakarta Persistence API (JPA) [2] è una serie di meccanismi che permettono di definire regole in linguaggio *Java* per associare oggetti ed entità di una base dati. L'interazione tra i due avviene grazie allo strato ORM, solitamente gestito da framework come *Hibernate ORM* o *EclipseLink*. Lo strato ORM è responsabile della conversione di oggetti software affichè questi possano interagire con tabelle e colonne in una base dati relazionale. Di *default* le classi vengono trasformate in entità ed i campi diventano le colonne delle tabelle.

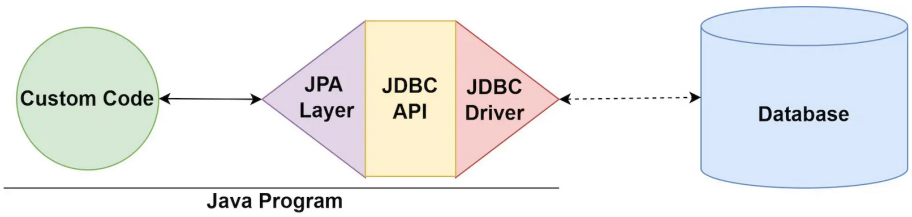

Figura 2.1: JPA e lo strato ORM di *Java*

Nel caso specifico di *Java*, lo strato ORM si comporta da *adapter*, in quanto adatta il linguaggio ad oggetti in linguaggio *SQL* e tabelle relazionali. In questo modo, lo sviluppatore riesce a creare oggetti che persistono oltre la chiusura del programma, senza mai dover uscire dal paradigma di programmazione orientato agli oggetti.

Per quanto riguarda invece l'utilizzo del JPA, non è richiesto specificare come gli oggetti vengano salvati, ma è sufficiente effettuare la mappatura di questi agli elementi della base dati. Questo avviene all'interno delle classi con delle annotazioni specifiche. Queste classi, il cui unico scopo è di immagazzinare dati, vengono definite Data Transfer Object (DTO).

```
1   @Entity
2   @Table(name = "person")
3   @Data
4   public class Person {
5       @Id
6       private String cf;
7       private String name;
8       private String surname;
9
10      @OneToMany(mappedBy = "person")
11      private Set<Car> car;
12  }
```

Listing 2.5: Esempio di un DTO con il lato della relazione *uno-a-molti*

L'annotazione *@Entity* specifica che la classe fa riferimento ad un'entità, *@Table* indica a quale tabella specifica associare la classe, in questo caso *person*, e *@Id* segnala quale

campo verrà trasformato in chiave primaria. Infine, l'annotazione *@OneToMany* indica il tipo di relazione che viene creata tra le entità *person* e *car*. A livello di programmazione ad oggetti, ciò viene implementato riportando i riferimenti agli elementi della tabella *car* in una collezione di oggetti di tipo ***Car*** all'interno della classe ***Person***, siccome tra le due tabelle c'è una relazione di tipo *uno-a-molti* dal punto di vista di *person*.

```
1     @Entity
2     @Table(name = "car")
3     @Data
4     public class Car {
5         @Id
6         private String plate;
7         private String mark;
8         private String model;
9
10        @ManyToOne
11        @JoinColumn(name = "cf", referencedColumnName = "cf")
12        private Person person;
13    }
14
```

Listing 2.6: Esempio di un DTO con il lato della relazione *molti-ad-uno*

@JoinColumn indica che la *chiave esterna* verrà inserita all'interno della tabella *car*, si chiamerà cf e farà riferimento al campo dello stesso nome. Infine, l'annotazione *@Data*, dalla *libreria lombok*, fa in modo che vengano inseriti in automatico all'interno dell'*editor* i metodi *getter*, *setter*, *toString* e *equals* per tutti i campi della classe, senza doverli scrivere esplicitamente.

2.2.2 Spring Data JPA

Nel caso di *Spring Data JPA*, vengono seguiti tutti i meccanismi delle JPA presentati sopra, in aggiunta a strumenti utili per semplificare l'utilizzo e l'interazione con la base dati. In particolare, la fase di installazione viene ridotta ad una semplice aggiunta delle dipendenze nel *file pom.xml*, mentre *Spring Boot* si occuperà di integrare il JPA nell'applicazione in automatico.

Tra gli strumenti forniti da *Spring Data JPA* ci sono anche delle interfacce chiamate *repository* con metodi per le operazioni CRUD sulla base dati e per operazioni di ordinamento. Si possono aggiungere metodi personalizzati estendo queste interfacce.

Figura 2.2: Esempi *repository* offerte da *Spring Data*

Infine, il collegamento effettivo alla base dati viene stabilito impostando il data source e le credenziali di accesso all'interno del file application.properties. In questo modo, è possibile indicare a *Spring Boot* e *Spring Data* a quale base dati collegarsi e che tipo di permessi avrà.

```
## PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/AutoDB
spring.datasource.username=username
spring.datasource.password=password
spring.jpa.show-sql=true

## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen
    ↪ database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
    ↪ PostgreSQLDialect
```

Listing 2.7: Esempio istruzioni per il collegamento ad una base dati di tipo *PostgreSQL*

2.2.3 Spring Data REST

Spring Data REST permette di costruire *servizi web REST* sfruttando le *repository* di *Spring Data*. Questo viene fatto creando i diversi endpoint REST e, attraverso gli altri strumenti forniti dai moduli di *Spring Data*, implementando operazioni CRUD sulla base dati. Durante il periodo di formazione, ho avuto occasione di creare un prototipo che implementasse due *servizi REST*, ciascuno con endpoint dedicati alle operazioni CRUD che agissero sui dati di una semplice base dati con due entità: *person* e *car*.

Per sviluppare questo sistema, vengono implementati i seguenti componenti:

- **Strato di persistenza**: comprende componenti come le DTO impostate in modo da essere utilizzate dalla JPA e le *repository* che estendono le interfacce di *Spring Data*, permettendo così l'interazione con i dati all'interno della base dati. Vengono utilizzati i metodi di repository come ***JpaRepository*** per le operazioni CRUD più comuni, dopodichè, si possono aggiungere metodi per operazioni specifiche.

```
    @Repository
    public interface PersonRepository
    ↪ extends JpaRepository<Person,String> {
        public Person findByCf(String cf);
    }

```

Listing 2.8: Esempio di classe repository

  Nel caso di metodi personalizzati come nell'esempio sopra, *Spring* capisce in automatico in base al nome che tipo di operazione fare sulla base dati in quanto segue degli standard di nomenclatura per creare le corrispettive *query* in *linguaggio SQL*. In questo caso, infatti, quando un oggetto di tipo ***PersonRepository*** farà una chiamata al metodo ***findByCf(String cf)***, questo farà in modo che venga lanciata sulla base dati la *query* necessaria per selezionare e ritornare la persona con il cf corrispondente. L'annotazione *@Repository* identifica la classe per *Spring* come parte della categoria delle *repository*. Questo può risultare utile nella gestione di eccezioni specifiche allo strato di persistenza.

- **Strato di servizio**: classi di servizio che gestiscono le implementazioni dei metodi necessari per fare operazioni sui dati all'interno della base dati. Sfruttano le *repository* inserite come dipendenze.

```
    @Service
    @Transactional
    public class PersonService {
        @Autowired
        private PersonRepository personRepository;
        public List<Person> SelAll()
        {   return personRepository.findAll();    }

        public Person SelPersonByCf(String cf)
        {   return personRepository.findByCf(cf);  }

        public void InsPerson(Person person)
        {   personRepository.saveAndFlush(person);  }

        public void DelPersonByCf(String cf)
        {   personRepository.deleteById(cf);    }
        ...
    }
```

Listing 2.9: Esempio di classe di servizio

L'annotazione *@Service* identifica per *String* che la classe si trova nella categoria di servizio, dunque che al suo interno è presente della business logic. *@Transactional* indica che il servizio gestisce *transazioni*, ossia operazioni da fare su una base di dati. In questo modo, il comportamento viene lasciato in mano a *Spring Data* per quanto riguardano situazioni in cui sia necessario il *roll-back* della transazioni, quali eccezioni o errori.

- **Strato di controllo**: classi definite *controller* gestiscono le *richieste web* che arrivano ai diversi endpoint specificati. Questi include i paramentri che si devono ricevere, il tipo di operazioni che si devono eseguire, grazie alla dipendenza da una classe di servizio, e la risposta da ritornare al *client*.

```
@RestController
@RequestMapping(value = "api/person")
public class PersonController {
    @Autowired
    private PersonService personService;
    @GetMapping(produces = "application/json")
    public ResponseEntity<List<Person>> selAll() {
        //restituisce tutte le person e una risposta al client
        ...
    }
    @PostMapping(value = "/insert")
    public ResponseEntity<Person> insPerson(@RequestBody Person p){
        //inserisce una persona e ritorna una risposta al client
        ...
    }
    @DeleteMapping(value = "/delete/{cf}")
    public ResponseEntity<?> delPersonByCf(@PathVariable("cf")
   ↪ String cf) {
        //elimina una persona e ritorna una risposta al client
        ...
    }
    ...
}
```

Listing 2.10: Esempio di REST *controller* con una parte degli endpoint

L'annotazione *@RestController*, assieme a *@RequestMapping*, imposta la REST API in modo che il *controller* sia in grado di gestire richieste web in corrispondenza degli endpoint specificati. Le operazioni vengono implementate attraverso metodi a cui vengono associate annotazioni come *@GetMapping* per ricevere e gestire richieste *GET*, *@PostMapping* per *POST*, *@DeleteMapping* per *DELETE* e *@PutMapping* per *PUT*. Per l'esempio di operazione di *POST*, viene richiesto come parametro nel *body* della richiesta un oggetto di tipo ***Person*** attraverso l'annotazione *@RequestBody*. Guardando invece all'esempio di *DELETE*, viene estratta con *@PathVariable* dall'URI la stringa corrispondente al cf della persona da eliminare.

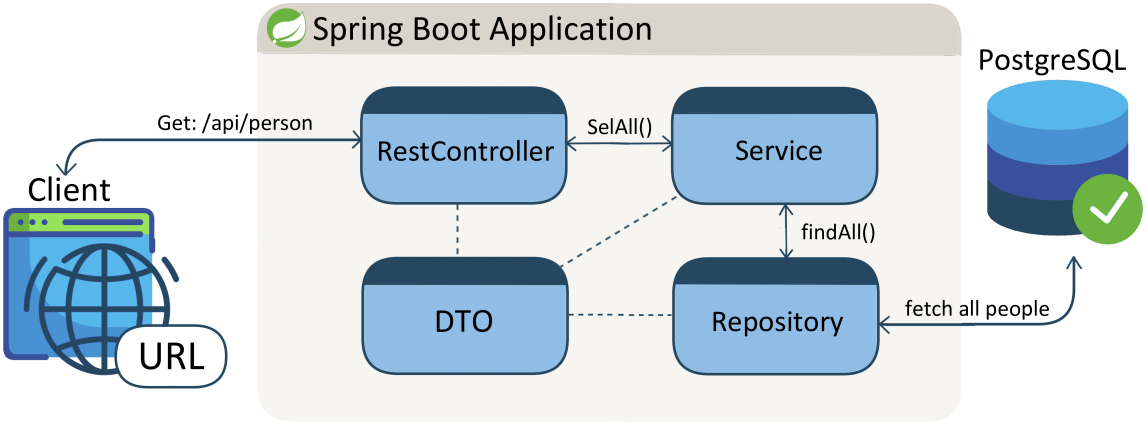

Figura 2.3: Esempio struttura applicazione con *Spring Data REST*

2.3 Spring Batch

2.3.1 Batch Jobs

Un *batch job* è un programma o una serie di programmi eseguiti in *batch mode*: una sequenza di comandi da eseguire dal sistema operativo messi in un *file* (*batch file, file di comando, job script o shell script*) e inviati per essere eseguiti come un'unica entità. L'opposto di un *batch job* è l'*interactive processing*, in cui un utente invia comandi individuali da eseguire immediatamente. In molti casi, i *batch jobs* vengono eseguiti durante la sera o in un periodo di tempo in cui il computer non viene utilizzato. Questo è solitamente il modo migliore per far eseguire un programma che richiede molte risorse dal computer.

Su *cluster* di alte prestazioni, gli utenti generalmente inviano *batch jobs* a gruppi predefiniti di partizioni che vengono gestiti da un'applicazione di gestione delle risorse. Alcuni *cluster* implementano un pianificatore di *job* che invia *batch jobs* sulla base della disponibilità delle risorse, dei requisiti del *job* specificati dall'utente, e delle politiche di utilizzo impostate dagli amministratori del *cluster*.

2.3.2 Spring Batch

Spring Batch[8] [9] fornisce funzioni essenziali nell'elaborazione di grandi volumi di dati, incluse la registrazione/tracciamento, gestione transazioni, *job*, elaborazioni statistiche, *job restart e skip* e gestione risorse. In più, fornisce servizi altamente tecnici e caratteristiche che permettono di eseguire *batch jobs* con ampi volumi di dati, mantenendo prestazioni alte e fornendo tecniche di partizione della memoria. Per quanto riguarda il

tracciamento dei *job*, *Spring Batch* crea all'interno della base dati di destinazione una serie di tabelle in cui registra i dettagli sull'andamento dei *job* che vengono lanciati. La struttura di un servizio *Spring Batch* si articola nei seguenti componenti: un *Job Launcher* in grado di lanciare un *Job*, il quale a sua volta è stato configurato come *bean* in una classe apposita. All'interno di questa classe di *configurazione*, vengono definiti anche i *bean* degli *Step*, che un *Job* deve eseguire in ordine. Infine, in base alla strategia di implementazione, gli *Step* si possono articolare in altri componenti, eventualmente da configurare.

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;
    @Autowired
    private ClientRepository clientRepository;

    @Bean
    public ItemReader<Client> reader()
    {   /*configurazione reader ... */  }

    @Bean
    public ClientProcessor processor()
    {   /*configurazione processor ... */ }

    @Bean
    public ItemWriter<Cliente> writer()
    {   /*configurazione writer ... */  }

    @Bean
    public Step step()
    { /*configurazione step ... */ }

    @Bean
    public Job runJob()
    {   /*configurazione job ... */    }
}
```

Listing 2.11: Esempio di classe di configurazione dei *bean* in un servizio *Spring Batch*

Il *Job launcher* può essere chiamato da un REST *controller* oppure, più comunemente da uno *scheduler*, ossia un componente che fa eseguire il *Job* al *launcher* ad intervalli regolari di tempo o in corrispondenza di orari specifici.

```
@Component
@EnableScheduling
public class JobScheduler {
    @Autowired
    private JobLauncher jobLauncher;
    @Autowired
    private Job job;

    @Scheduled(cron="* */2 * * * *") //Ogni due minuti
    public void importCsvToDBJob()
    {   //lancio job ...    }
}
```

Listing 2.12: Esempio di *job scheduler*

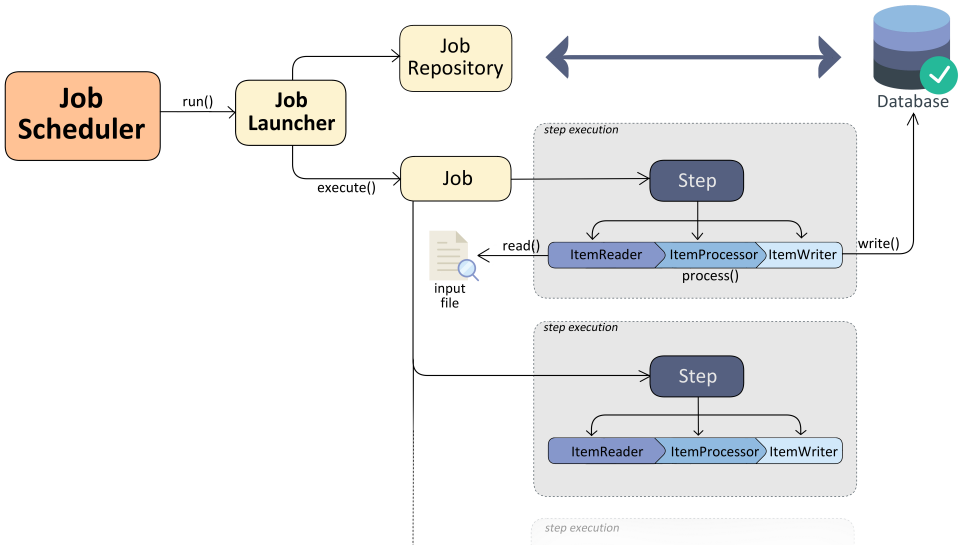

Figura 2.4: Esempio architettura di un servizio *Spring Batch* con *chunk processing*

2.3.3 Chunk processing

Spring Batch fornisce due strategie per implementare *Job*: attraverso *tasklet* oppure attraverso *chunks*. Nel primo caso, ogni *Step* deve eseguire una sola *task*, dunque ci sarà uno *Step* per la lettura, uno eventualmente per l'elaborazione ed infine uno per la scrittura di dati. D'altro canto, l'implementazione attraverso *chunks* implica che verranno svolte delle operazioni su blocchi di dati alla volta. A differenza dell'implementazione con i *tasklet*, il flusso degli *Step* segue la strategia rappresentata nel seguente *pseudocodice*:

```
1     List items = new Arraylist();
2     for(int i = 0; i < commitInterval; i++){
3         Object item = itemReader.read();
4         if (item != null) {
5             items.add(item);
6         }
7     }
8
9     List processedItems = new Arraylist();
10    for(Object item: items){
11        Object processedItem = itemProcessor.process(item);
12        if (processedItem != null) {
13            processedItems.add(processedItem);
14        }
15    }
16
17    itemWriter.write(processedItems);
```

Listing 2.13: Pseudocodice con le operazioni eseguite da uno *Step* nel *chunk processing*

In primo luogo, viene chiamato l'oggetto ***itemReader*** che, attraverso il metodo ***read()***, aggiunge tanti elementi ad una lista di oggetti quanto è ampia la dimensione dei *chunk*. In seguito alla lettura, gli oggetti della lista vengono elaborati dall'oggetto ***itemProcessor***, che restituisce una nuova lista di elementi. Infine, l'oggetto ***itemWriter*** riceve la lista definitiva di elementi e fa un'operazione di scrittura con il metodo ***write(List list)***. La fase di elaborazione dati ed il *processor* sono entrambi facoltativi.

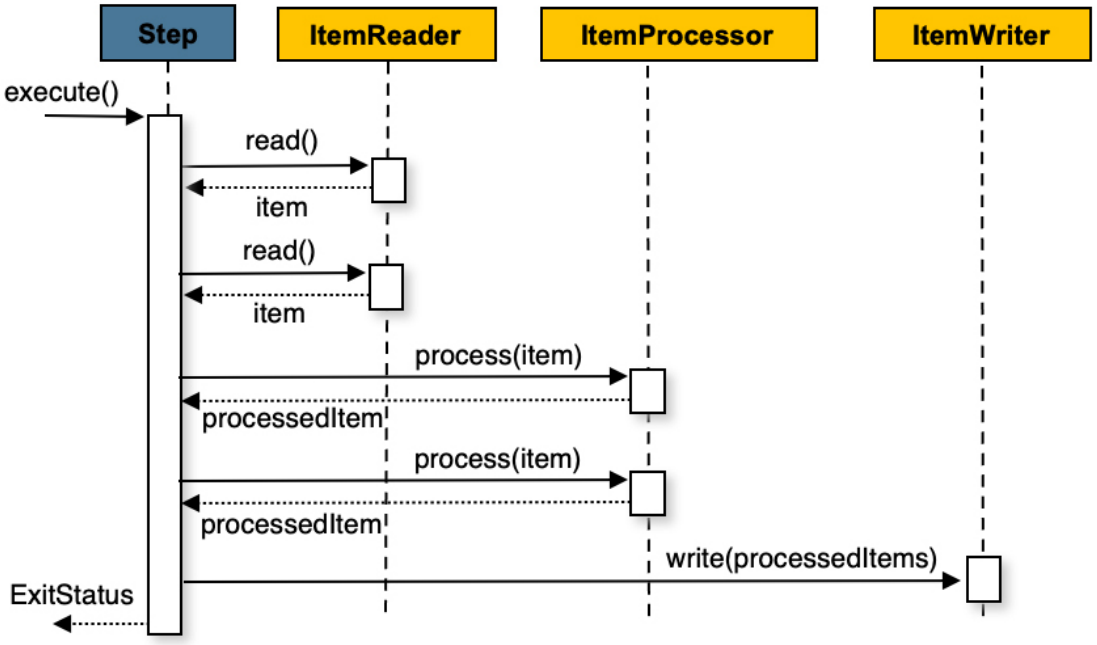

Figura 2.5: Flusso di uno *Step* con l'implementazione a *chunks*. Fonte: [9]

Capitolo 3

Il progetto

3.1 Requisiti

In primo luogo, ho considerato i requisiti specifici di ciò che dovevo sviluppare:

1. Prima di iniziare, dovevo risultare preparato su tutti i concetti dei moduli *Spring Data* e *Spring Batch* che avrei utilizzato nel progetto, in particolare le *repository* di *Spring Data*, il funzionamento del *chunk processing*, assieme alle classi *reader* e *writer* specifiche di *Spring Batch*;

2. Analisi dello stato del progetto per determinare ciò che era stato sviluppato ed eventuali parti mancanti;

3. Progettazione di un sistema con *Spring Batch* per leggere e scrivere i dati dei sensori di parcheggio ed uno per i dati dei sensori ambientali verso una base dati;

4. Implementazione, partendo anche dai prototipi sviluppati in fase di formazione, del servizio *Spring Batch* che gestisca la configurazione ed il lancio di *Job* che leggono dal *file di input* generato dai sensori di parcheggio ed inseriscono i dati all'interno delle tabelle nella base dati;

5. Implementazione servizio *Spring Batch* con obiettivo analogo, ma per i sensori ambientali;

6. Creazione *Job scheduler* per impostare il lancio periodico dei *Job*;

7. Eventuale fase di collaudo con *test di unità*, automatici o manuali in base al tempo rimanente;

3.2 Analisi iniziale

3.2.1 Analisi sullo stato del progetto

Ho iniziato l'analisi dall'osservazione dei componenti che erano già stati sviluppati durante percorsi di tirocinio precedenti al mio. In particolare, era già state creato in parte un back end con i componenti caratteristici di un servizio basato su Spring Data REST:

- strato di persistenza con le entità collegate secondo i meccanismi della JPA e le repository relative alle entità. Quest'ultime, nonostante estendessero quelle proposte da *Spring Data*, avevano i metodi impostati attraverso il *linguaggio SQL*, invece di lasciare la conversione al JPA. Questa non risulta una scelta ottimale in quanto, in

base al sistema con cui vengono salvati i dati, ci possono essere differenze nel lessico e quindi riscontrare possibili problemi nel passaggio da un sistema ad un altro.

```
@Transactional
@Repository
public interface SensorsRepository
    ↪ extends CrudRepository<Sensors,Long>  {
     @Query("select s from Sensors s where s.id = ?1 order by id")
     public Sensors getSensorById(Long sensorId);

     @Query("select s from Sensors s order by id")
     public List<Sensors> getAllSensorsFromDB();
     ...
}
```

- strato di servizio che implementava le operazioni CRUD sulla base dati con l'ausilio delle *repository*
- strato di controllo con i REST *controller* con gli endpoint impostati per ricevere e gestire le richieste che arriveranno dal front end

Ogni entità implementata presentava una DTO associata, una *repository*, un servizio ed un *REST controller*, tuttavia non tutte le entità erano presenti. Dalla struttura delle classi nel back end era chiaro che anche la base dati era stata già progettata in parte, tuttavia mancavano le tabelle per i dati ambientali.

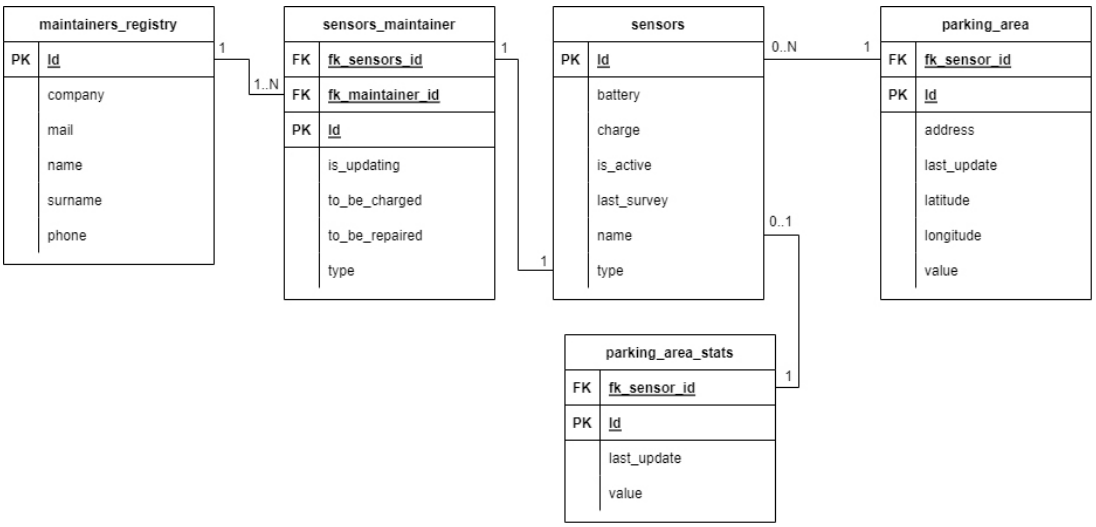

Figura 3.1: Struttura del database dalle classi del back end

Infine, era stato implementato un servizio di campionamento che, attraverso le repository di Spring Data e funzioni di lettura *file*, fosse in grado di aggiornare ogni 2 minuti i dati dei sensori all'interno della tabella sensors nella base dati. Questo servizio non utilizzava gli strumenti forniti da *Spring Batch*, dunque risultava più complesso sia concettualmente, sia a livello di codice. Questo era vero in particolare per la lettura dai *file* e la conversione di dati in oggetti che risulta più semplice sfruttando i *reader* di *Spring Batch*.

Per quanto riguarda la parte di front end, questa era ancora in fase di sviluppo, mentre la base dati non aveva un'installazione ufficiale. Per questo motivo, durante lo sviluppo ho creato un'istanza locale della base dati, seguendo lo schema relazionale, sulla quale ho eseguito le operazioni ed i test.

3.2.2 Analisi formato dei file di input

Per procedere con la fase di progettazione, ho analizzato in particolare anche il formato in cui i dati venivano registrati nei *file di input*. Nel caso del *file XML* con i dati dei sensori di parcheggio, era presente un *tag* principale XML con nome *markers*, al cui interno c'erano 15 instanze di *tag* con nome *marker*. All'interno di ogni marker, vi erano attributi che corrispondevano alle informazioni relative ai sensori, in particolare id, nome, indirizzo, latitudine, longitudine, stato della piazzola, livello di carica della batteria e se il sensore era attivo o meno. Confrontando questi dati con la tabella dei sensori di parcheggio, i dati rilevanti risultavano l'id del sensore, la latitudine, la longitudine e lo stato, da aggiungere ai campi dell'id della misura (autogenerato ed incrementale) e orario dell'ultimo aggiornamento.

```
<markers>
    <marker id="1" name="156A2C71" address="Padova Galleria Spagna" lat=
    ↪ "45.389040" lng="11.928577" state="0" battery="3,7V" active="1"/>
    ...
    ...
    <marker id="15" name="156A2B7113" address="Via Sorio" lat="45.400546
    ↪ " lng="11.855015" state="1" battery="3,7V" active="1"/>
</markers>
```

Listing 3.1: Formato *file input* XML

Per quanto riguarda il *file* in formato *txt* con i dati dei sensori ambientali, per ogni riga vi era una serie di campi suddivisi da un *token* seguendo lo schema seguente:

```
    date;hour;pm2.5;pm10;temperature;humidity
```

Listing 3.2: Formato *file input txt*

Le altre caratteristiche dei sensori ambientali non sono state ancora registrate in un *file* facilmente accessibile come *input*. Per questo motivo, i campi restanti sono stati valorizzati direttamente all'interno del codice, oppure attraverso riferimento il file application.properties prima di salvare i dati sulla base dati.

Ciascuna delle tabelle dei dati ambientali presentava colonne molto simili: id della misura (autogenerato ed incrementale), id del sensore, indirizzo, latitudine, longitudine, orario della misura ed infine valore misurato. Ciò significa che per ogni tabella di misure ambientali, tutti i campi dei diversi *record* riportati erano simili, ad eccezione del valore misurato che le contraddistingueva.

```
    sensor.environmental.address = Corso Spagna 30
    sensor.environmental.latitude = 45.388638
    sensor.environmental.longitude = 11.928341
```

Listing 3.3: Inserimento dei campi all'interno del *file* application.properties

```
    @Value("${sensor.environmental.address}")
    private String address;
    @Value("${sensor.environmental.latitude}")
    private String latitude;
    @Value("${sensor.environmental.longitude}")
    private String longitude;
```

Listing 3.4: Lettura dei campi dal *file* application.properties

3.3 Progettazione

Durante la fase di progettazione, è stato deciso come organizzare la lettura e l'elaborazione dei campi per poterli suddividere ed associarli agli attributi delle entità nel database già progettato, aggiungendo eventulmente le tabelle mancanti. Con l'aiuto dei tutor aziendali, siamo arrivati ad una struttura con due servizi separati implementati con *Spring Batch*.

3.3.1 Progettazione ParkingSensorsBatch

Il primo servizio, chiamato ***ParkingSensorsBatch***, doveva lanciare dei *Job* che, ad intervalli di tempo regolari, leggessero i dati dal *file XML* e li inserissero nella tabella dei sensori di parcheggio della base dati, attraverso le *repository* costruite sulle interfacce fornite da *Spring Data JPA*. Infine, siccome all'interno del *file* era riportato l'id del sensore e la DTO ***ParkingSensors*** richiede un riferimento ad un oggetto ***Sensor***, ho creato una classe ***ParkingSensorsInputData*** che ricevesse in *input* i dati necessari, e che potesse essere facilmente convertita in un oggetto di tipo ***ParkingSensors*** partendo dai suoi campi.

```
1     public ParkingSensors(ParkingSensorsInputData p)  {
2         this.address = p.getAddress();
3         this.latitude = p.getLatitude();
4         this.longitude = p.getLongitude();
5         this.value = p.isValue();
6         this.timestamp = LocalDateTime.now();
7     }
8
```

Listing 3.5: Uno dei costruttori della classe DTO ***ParkingSensors***

3.3.2 Progettazione EnvironmentalSensorsBatch

Il secondo servizio, chiamato ***EnvironmentalSensorsBatch***, doveva a sua volta lanciare dei *Job* ad intervalli di tempo regolari, ma con più di uno *step* in questo caso. Siccome tutti i dati ambientali erano riportati nello stesso *file*, abbiamo trovato opportuno creare una classe ***SensorsData*** il cui unico scopo era ricevere i dati in *input*.

```
1     public class SensorsData {
2         private String date;
3         private String time;
4         private String pm25Values;
5         private String pm10Values;
6         private String tempValues;
7         private String humidValues;
8     }
9
```

Listing 3.6: Struttura della classe ***SensorsData***

Il primo *step* risultava dunque la lettura dei dati dal *file txt*, per creare in seguito una lista di oggetti di tipo *SensorsData* all'interno della quale salvare i dati. Una volta valorizzati i campi, si potevano estrarre con degli *step* successivi i dati desiderati dalla lista, in base al tipo di misura ambientale, per registrarli nelle apposite tabelle.

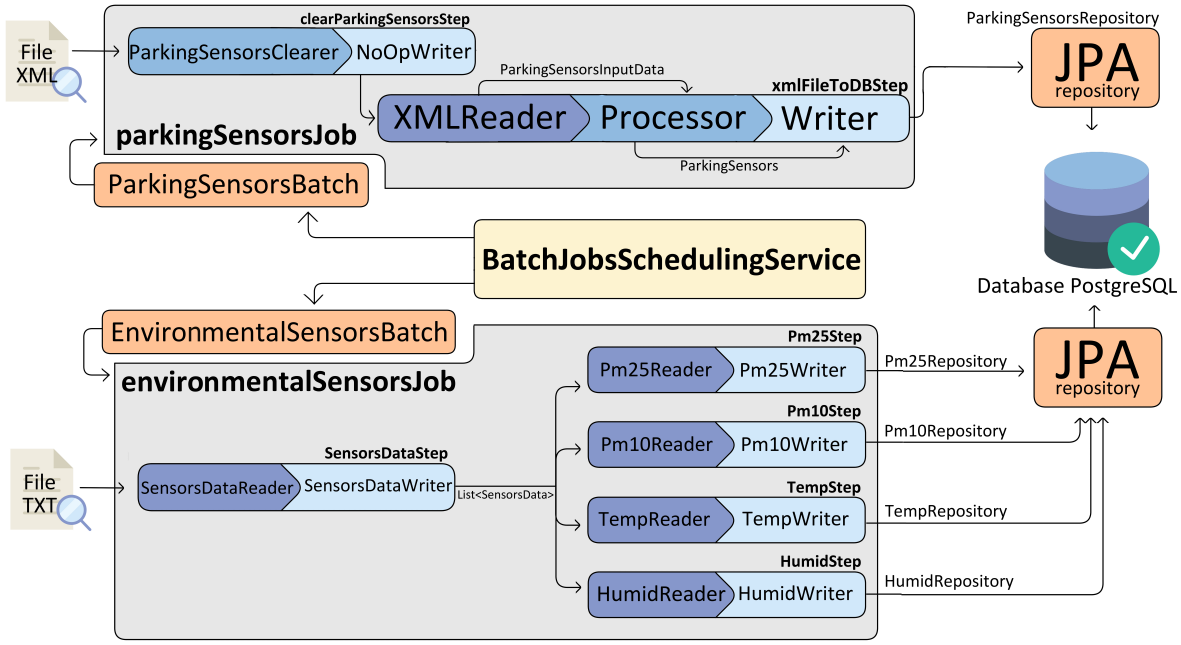

Figura 3.2: Struttura progettata

3.3.3 Problema dei dati duplicati

Un altro aspetto da valutare è stato il comportamento del servizio nel caso di dati duplicati. Considerando che si trattasse di una lettura periodica dello stesso *file*, risultava molto facile che due record letti dal servizio in due momenti diversi risultassero essenzialmente uguali, se non per l'id generato automaticamente. Per questo motivo, inizialmente vi era una duplicazione continua dei dati ad ogni lancio dei *Job*. Per risolvere questo problema, abbiamo optato per due approci diversi nei due sistemi. Nel caso dei sensori di parcheggio, dato che l'obiettivo al momento era solamente un continuo aggiornamento dei dati per avere informazioni in tempo reale, abbiamo considerato sufficiente rimuovere all'inizio di ogni lancio dei *Job* i dati riportati all'interno della tabella *parking_sensors* per poi inserire quelli aggiornati. Per i sensori ambientali, considerando che l'intenzione fosse di creare uno storico dei dati, non risultava accettabile l'eliminazione e il reinserimento. In questo caso, al lancio del programma sarebbero stati innanzitutto eliminati i record vecchi, ancora presenti all'interno delle tabelle della base dati. In secondo luogo, verrebbero lanciati i *Job* periodici di lettura e scrittura dei dati. All'interno di ogni esecuzione dei *Job*, ci sarebbe stato un controllo dei dati da importare rispetto a quelli già importati dai *Job* passati, tutti ancora presenti nella lista ***SensorsDatas***. In questo modo, i dati sarebbero stati inseriti nelle tabelle della base dati solamente se fossero risultati nuovi.

3.3.4 Analisi e modifica della base dati preesistente

Esaminando infine la base dati che era stata progettata, era chiaro che sarebbe stato necessario introdurre le tabelle utili per i dati ambientali. In tale occasione, abbiamo anche rivalutato la struttura delle altre entità e le loro relazioni, in quanto in alcuni casi presentavano delle imprecisioni. In particolare, alcuni nomi di tabelle erano poco esplicativi, ad esempio non era chiaro se *parking_area* si riferisse ad un'unica piazzola, ad un intero parcheggio oppure ad allo stato registrato dal sensore di parcheggio. Inoltre, la relazione tra le tabelle *sensors* e *parking_area* erano di tipo *uno-a-molti*, dunque ad ogni sensore venivano associati gli stati di molteplici piazzole di parcheggio, il che non

corrispondeva al modello concettuale. Infatti, l'intenzione era di fare in modo che ad un sensore di parcheggio venisse associato lo stato di una sola piazzola di parcheggio e viceversa. A questo punto, abbiamo deciso di alterare lo schema della base dati nel seguente modo:

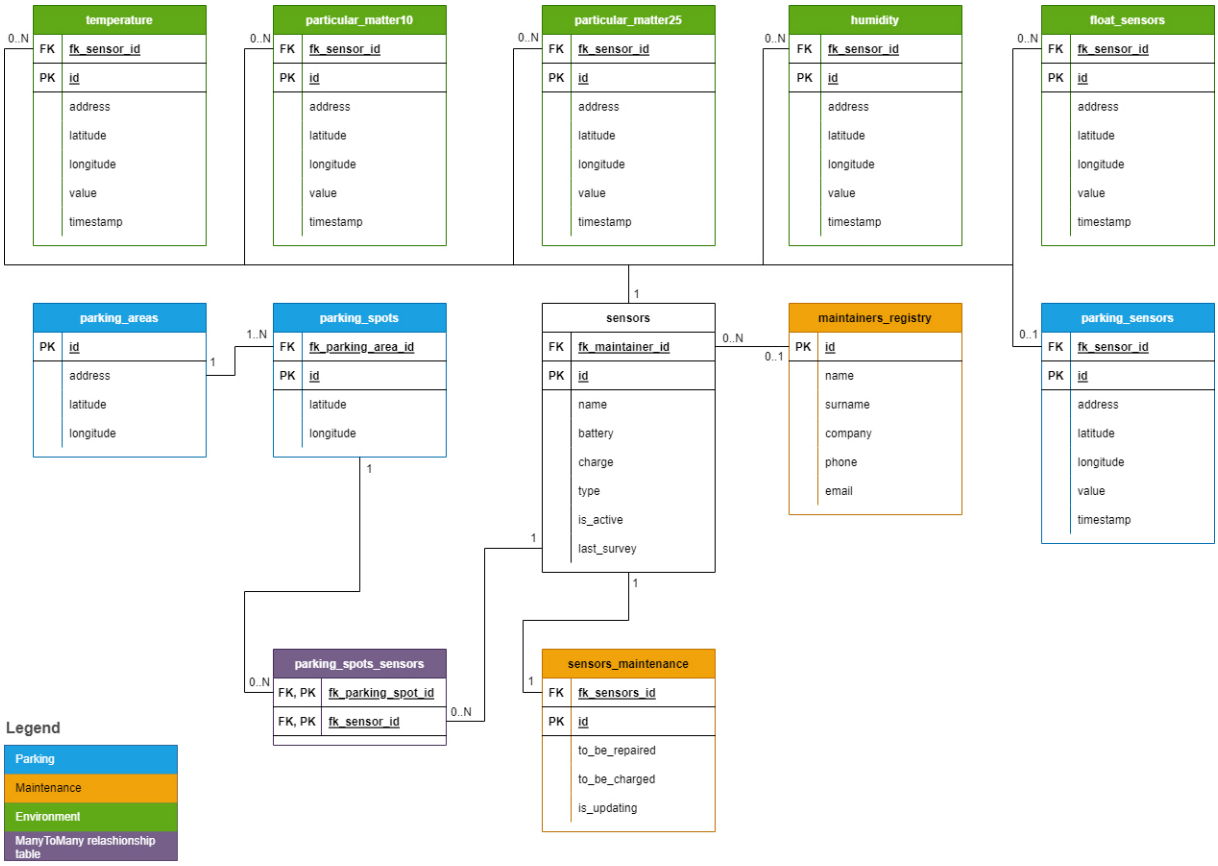

Figura 3.3: Nuova struttura base dati

3.4 Implementazione

3.4.1 JPARepositories

In primo luogo, ho scelto di completare l'implementazione delle entità mancanti, tra cui le tabelle *particular_matter25*, *particular_matter10*, *temperature* e *humidity*, corrispondenti rispettivamente alle classi ***Pm25***, ***Pm10***, ***Temperature*** e ***Humidity***. Ho aggiunto anche una versione semplice delle loro repository, dunqe ***Pm25Repository***, ***Pm10Repository***, ***TempRepository*** e ***HumidRepostory***, che ereditano i metodi dall'interfaccia ***CrudRepository*** fornita da *Spring Data*, senza aggiungerne di nuovi. In questo modo avevo accesso ai metodi base per poter fare le operazioni CRUD sulla base dati. Una particolarità di questi quattro DTO è che, ispirandomi all'implementazione delle altre DTO che erano già state fatte, il campo ***fkSensorId*** di tipo *Long* risulta *chiave esterna* in quanto fa riferimento all'id del sensore associato alla misura. Questa risulta un'implementazione più semplice di quella consigliata all'interno della documentazione di *Spring Data*, in quanto manca il riferimento all'intero oggetto di tipo ***Sensor***, ma viene associato solo il suo campo id dal lato delle tabelle di misura. In ogni caso, se risultasse necessario avere un riferimento al sensore effettivo in futuro, è possibile alterare abbastanza facilmente sia le classi delle entità, che i servizi sviluppati in seguito per importare i dati.

```
    @Entity
    @Table(name = "particular_matter25")
    @Data
    public class Pm25 {
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private Long id;
      @Column(name = "fk_sensor_id")
      private Long fkSensorId;
      private String address;
      private String latitude;
      private String longitude;
      @Basic(optional = false)
      private String value;
      private String timestamp;
            ...
    }
```

Listing 3.7: DTO implementata per le misure di pm2.5

Le annotazioni *@Column* e *@Basic* sono modi per associare esplicitamente i campi della classe a colonne di una tabella e permettono di aggiungere caratteristiche come il nome della colonna, oppure se il campo è obbligatorio. Se tali annotazioni non vengono indicate, JPA trasforma automaticamente i campi compatibili in colonne. *@GeneratedValue* invece indica che quel campo deve essere auto-generato, mentre con *GenerationType.AUTO* che la strategia di generazione dell'id viene lasciata in mano al gestore della persistenza, in questo caso *Hibernate ORM*. Questa impostazione risulta necessaria in quanto alcuni metodi di generazione non sono supportati durante l'esecuzione di istruzioni in modalità *batch*.

```
    @Entity
    @Table(name = "sensors")
    @Data
    public class Sensors {
        @Id
        private Long id;
        private String name;
        private String battery;
        private String charge;
        private String type;
        private boolean isActive;
        private LocalDateTime lastSurvey;

        @OneToOne(mappedBy = "sensors")
        private ParkingSensors parkingSensors;

        @OneToMany(cascade = CascadeType.ALL, mappedBy = "fkSensorId",
    ↪ orphanRemoval = true)
        private List<Pm10> pm10;

        @OneToMany(cascade = CascadeType.ALL, mappedBy = "fkSensorId",
    ↪ orphanRemoval = true)
        private List<Pm25> pm25;

        @OneToMany(cascade = CascadeType.ALL, mappedBy = "fkSensorId",
    ↪ orphanRemoval = true)
        private List<Temperature> temperature;
```

```
25
26        @OneToMany(cascade = CascadeType.ALL, mappedBy = "fkSensorId",
    ↪ orphanRemoval = true)
27        private List<Humidity> humidity;
28
29        @OneToMany(cascade = CascadeType.ALL)
30        @JoinColumn(name = "fk_sensor_id", referencedColumnName = "id")
31        private List<SensorsMaintainer> maintainers;
32
33        @ManyToMany(mappedBy = "sensors")
34        private List<ParkingSpots> parkingSpots;
35        ...
36     }
37
```

Listing 3.8: Struttura della DTO ***Sensors*** con le diverse relazioni

Ho approfittato anche per aggiungere le DTO che risultavano necessarie secondo la nuova struttura dati, come ***ParkingAreas*** per la tabella *parking_areas* e ***ParkingSpots*** per *parking_spots*. In più, ho rinominato la vecchia classe *ParkingArea* in ***ParkingSensors***, dato che la tabella *parking_area* è stata cambiata in *parking_sensors*. Ho effettuato anche una correzione delle relazioni tra le entità: ora tra le tabelle *sensors* e *parking_sensors* vi è una relazione di tipo *uno-ad-uno*. In questo modo, ad ogni sensore viene associato un dato sullo stato della piazzola, mentre tra *sensors* e *parking_spots* è stata impostata una relazione di tipo *molti-a-molti* per considerare la presenza di sensori di diverso tipo associati ad una singola piazzola. Allo stesso tempo, alcuni sensori come quelli ambientali possono effettuare misure su molteplici piazzole. Di conseguenza, ho inserito le nuove associazioni all'interno delle diverse DTO.

Infine, secondo quanto detto nella fase di progettazione in riferimento ai *file* di *input*, ho creato una classe ***SensorsData*** per l'ingresso dei dati ambientali, ed una classe ***ParkingSensorsInputData*** per la lettura dei sensori di parcheggio.

```
1     @Data
2     @XmlRootElement(name="marker")
3     public class ParkingSensorsInputData
4     {
5         @XmlAttribute(name = "id", required = true)
6         private Long fkSensorId;
7         private String address;
8         private String latitude;
9         private String longitude;
10        private boolean value;
11        private LocalDateTime timestamp;
12
13        @XmlAttribute(name = "state", required = true)
14        public boolean isValue()
15        { return value;    }
16             ...
17
18        public ParkingSensorsInputData() {
19          this.timestamp = LocalDateTime.now();
20        }
21    }
22
```

Listing 3.9: Classe ***ParkingSensorsInputData*** e le sue annotazioni

L'annotazione *@XmlRootElement* associa la classe all'*elemento XML*, mentre *@XmlAttribute* collega i campi della classe agli *attributi XML* dell'elemento specificato sopra. In alcuni casi è risultato necessario associare *@XmlAttribute* ai metodi *get* dei campi, tuttavia l'effetto rimane lo stesso. Queste annotazioni sono necessarie per permettere ad un unmarshaller di associare i valori degli attributi presenti all'interno del *file XML* ai campi della classe ***ParkingSensorsInputData***.

3.4.2 ParkingSensorsBatch

Per implementare ***ParkingSensorsBatch***, ho seguito l'approcio di un'architettura a *chunks*, dunque ho configurato un *Job* chiamato ***smartParkingJob***, che esegue in ordine i due *Step*: ***clearParkingSensorsStep*** e ***xmlFileToDbStep***. Il primo dei due svuota la tabella *parking_sensors* attraverso una classe chiamata ***ParkingSensorsClearer*** che implementa ***ItemReader<ParkingSensors>***. In tal modo, un oggetto di questo tipo può sostituire il *reader* nello *step* e, allo stesso tempo, eseguire le istruzioni desiderate. Infatti, al suo interno vi è un riferimento a ***parkingSensorsRepository*** e il metodo ***read()*** non fa altro che chiamare la funzione ***deleteAll()*** di ***parkingSensorsRepository*** e ritornare il valore *null*, indicando dunque che la "lettura" è conclusa.

```
1  public class ParkingSensorsClearer implements ItemReader<ParkingSensors>
2  {
3      private ParkingSensorsRepository parkingSensorsRepository;
4
5      @Override
6      public ParkingSensors read() {
7          parkingSensorsRepository.deleteAll();
8          return null;
9      }
10 }
11
```

Listing 3.10: Classe ParkingSensorsClearer

Per quanto riguarda il *writer* di questo *step* invece, ne è stato fatto sempre uno personalizzato chiamato ***NoOpWriter***, che implementa ***ItemWriter<ParkingSensors>***. Al suo interno, il metodo ***write()*** risulta vuoto. Questo tipo di *writer* era necessario in quanto è obbligatorio assegnare un *writer* allo *step* durante la sua configurazione.

```
1  public class NoOpWriter implements ItemWriter<ParkingSensors>
2  {
3      @Override
4      public void write(List<? extends ParkingSensors> items)
5      {
6      }
7  }
8
```

Listing 3.11: Classe NoOpWriter

In riferimento allo *Step step* ***xmlFileToDbStep***, ho configurato un *reader* di tipo *StaxEventItemReader* che, assieme all'*unmarshaller* di tipo *Jaxb2Marshaller* (dalla libreria *jaxb*), sono in grado di leggere il *file* in *formato XML* ed estrarre i dati per associarli automaticamente ai campi secondo i le annotazioni impostate sulla classe ***ParkingSensorsInputData***.

```
@Bean
public ItemReader<ParkingSensorsInputData> reader() {
    Jaxb2Marshaller markerMashaller = new Jaxb2Marshaller();
    markerMashaller.setClassesToBeBound(ParkingSensorsInputData.class);

    try {
        return new StaxEventItemReaderBuilder<ParkingSensorsInputData>()
                .name("smartCitySimReader")
                .resource(new UrlResource(sensorsUrl))
                .addFragmentRootElements("marker")
                .unmarshaller(markerMashaller)
                .build();
    } catch (MalformedURLException e) {
        e.printStackTrace();
        return null;
    }
}

```

Listing 3.12: Configurazione *reader* per il *file XML*

In seguito alla lettura, un *processor* trasforma gli oggetti letti da ***ParkingSensorsInputData*** a ***ParkingSensors*** collegando ad essi il sensore in base all'id letto grazie alla *repository* ***SensorsRepository***.

```
public class ParkingSensorsProcessor implements
    ↪ ItemProcessor<ParkingSensorsInputData,ParkingSensors>
{
    private SensorsRepository sensorsRepository;

    @Override
    public ParkingSensors process(ParkingSensorsInputData p) {
        ParkingSensors parkingSensors = new ParkingSensors(p);

        parkingSensors.setSensors(
 sensorsRepository.getSensorById(p.getfkSensorId()));

        return parkingSensors;
    }
}

```

Listing 3.13: Classe ***ParkingSensorsProcessor***

Infine, un *writer* di tipo ***RepositoryItemWriter*** sfrutta la *repository* ***ParkingSensorsRepository***, che chiamerà il metodo ***save(ParkingSensor p)*** per inserire i dati nella tabella *parking_sensors* .

3.4.3 EnvironmentalSensorsBatch

In modo analogo al ***ParkingSensorsBatch***, anche per ***EnvironmentalSensorsBatch*** ho seguito lo schema del *chunk processing*. Ho configurato un *Job* chiamato ***environmentalSensorsJob***, articolato in cinque *Step*. Il primo si occupa della lettura dei dati dal *file di input* e li salva in una lista di oggetti di tipo **SensorsData** chiamata **SensorsDatas**. La lettura dei dati avviene attraverso un *writer* di tipo ***FlatFileItemReader***, con funzioni per leggere *file* in cui ci sono dei dati separati da un *token*. Questo è possibile indicando a che simbolo corrisponde il *token* e l'ordine in cui i campi della classe

di destinazione si presentano all'interno del *file* di *input*. La scrittura avviene attraverso un *writer* personalizzato, che aggiunge gli elementi appena letti alla lista ***SensorsDatas*** solamente se questi non sono ancora presenti al suo interno, in modo da evitare duplicati. Il controllo avviene attraverso il metodo ***contains(Object o)*** di ***List***<>.

Una volta eseguito il primo *Step*, vengono chiamati i seguenti quattro: ***pm25Step***, ***pm10Step***, ***tempStep*** e ***humidStep***. Questi *Step* hanno comportamenti molto simili, ma ciascuno viene applicato ad un tipo di dato ambientale diverso. In particolare, ciascuno di essi ha un *reader* personalizzato per leggere dalla lista ***SensorsDatas*** il valore della misura corrispondente, assieme alla data e l'orario in cui è avvenuta. Una volta letti i dati, vengono composti gli oggetti, sempre in base al tipo di misura, e aggiunti ad una lista (una per ogni tipo di misura) per tenere traccia di quali sono stati aggiunti al lancio dei *Job* precedenti. Se questi oggetti non sono già presenti nella lista, vengono ritornati per essere inviati al *writer*. Il controllo effettuato è simile a quello degli oggetti SensorsDatas: viene sfruttato il metodo ***contains(Object o)*** di ***List***<>, il quale a sua volta chiama il metodo ***equals(Object o)*** delle classi ***Pm25***, ***Pm10***, ***Temperature*** e ***Humidity***. Quest'ultimo metodo è stato modificato per ignorare il campo id, affinchè due DTO ambientali vengano considerati *uguali* se i campi al loro interno sono uguali, indipendentemente dall'id che viene generato automaticamente alla loro creazione.

```
@Data
public class Pm25Reader implements ItemReader<Pm25>
{
    private int nextDataIndex;
    private List<SensorsData> sensorsDatas;
    private List<Pm25> pm25Data;

    //Set initial data, that does not change, manually
    @Value("${sensor.environmental.address}")
    private String address;

    @Value("${sensor.environmental.latitude}")
    private String latitude;

    @Value("${sensor.environmental.longitude}")
    private String longitude;

    //pm2.5 sensor is number 16
    private Long fkSensorId = 16L;

    @Override
    public Pm25 read()
    {
        Pm25 nextPm25Data = null;
        //Loops until it finds a NEW record or until there are no more
    ↪ records
        while (nextDataIndex<sensorsDatas.size() && nextPm25Data==null)
        {
            SensorsData nextSensorsData=sensorsDatas.get(nextDataIndex);

            String timestamp = nextSensorsData.getDate() + "-" +
    ↪ nextSensorsData.getTime();

            String value = nextSensorsData.getPm25Values();

            nextPm25Data = new Pm25(timestamp,address,latitude,longitude
    ↪ ,value,fkSensorId);

```

```
36              if(!pm25Data.contains(nextPm25Data))
37              {
38                  pm25Data.add(nextPm25Data);
39                  nextDataIndex++;
40                  return nextPm25Data;
41              }
42              else  {
43                  nextDataIndex++;
44                  nextPm25Data = null;
45              }
46          }
47          nextDataIndex = 0;
48          return nextPm25Data;
49      }
50  }
51
```

Listing 3.14: *Reader* implementato per la DTO delle misure di pm2.5

A questo punto, il *writer* di tipo ***RepositoryItemWriter<>***, viene configurato con la *repository* corrispondente al tipo di dato e, chiamando il metodo ***save()***, registra i nuovi dati nella base dati.

```
1       @Bean
2       public RepositoryItemWriter<Pm25> pm25Writer()  {
3           pm25Repository.deleteAll();
4
5           RepositoryItemWriter<Pm25> writer =new RepositoryItemWriter<>();
6           writer.setRepository(pm25Repository);
7           writer.setMethodName("save");
8           return writer;
9       }
10
```

Listing 3.15: Configurazione *writer* per la DTO delle misure di pm2.5

La configurazione del *writer* avviene all'esecuzione del programma, così come tutti gli altri *bean*. In questa occasione, per ogni *writer* delle misure ambientali configurato vengono eliminati i dati preesistenti all'interno base dati chiamando il metodo ***deleteAll()*** delle *repository*. In questo modo, si evitano i duplicati che ci possono essere tra i le misure registrate in passato all'interno base dati ed i dati da leggere dal *file di input*. Una soluzione alternativa potrebbe essere sviluppare un ulteriore controllo sui dati già presenti nella base dati, il che implicherebbe sviluppare metodi appositi all'interno delle *repository* per determinare la presenza o meno di una certa *tupla*.

3.5 Test

Non avendo tempo per sviluppare test di unità automatici con strumenti come JUnit4, mi sono ristretto a test di unità manuali. Ho utilizzato strumenti come il *Logger* del *logging framework Log4J2* all'interno del codice ed un misto tra il *terminale Psql* e il *client di PgAdmin4* di *PostgreSQL* per osservare gli effetti dei *Job* sulla base dati. In particolare, ho sfruttato il *Logger* per inserire messaggi di *log* in corrispondenza di istruzioni importanti, per assicurarmi che queste venissero eseguite e che seguissero l'ordine atteso. Per quanto riguarda i controll sulla base dati, sia *Psql* che *PgAdmin4* mi hanno permesso di fare

interrogazioni con il *linguaggio SQL* sulle tabelle per confermare che i dati venissero inseriti correttamente al loro interno. I test eseguiti erano i seguenti:

1. Test a tabella vuota con un solo lancio del *ob*
    - Stato iniziale: tabella da esaminare vuota.
    - Durante: eseguire il programma ed aspettare affinchè lanci il *Job* una volta.
    - Risultato atteso: tutti i dati inseriti correttamente nella tabella dal *file*.
2. Test a tabella vuota con molteplici lanci del *Job*
    - Stato iniziale: tabella da esaminare vuota.
    - Durante: eseguire il programma ed aspettare affinchè lanci il *Job* almeno due volte.
    - Risultato atteso: tutti i dati inseriti correttamente nella tabella dal *file*, senza duplicati.
3. Test a tabella non vuota con molteplici lanci del *Job*
    - Stato iniziale: tabella da esaminare con all'interno dati già registrati.
    - Durante: eseguire il programma ed aspettare affinchè lanci il *Job* almeno due volte.
    - Risultato atteso: tutti i dati sovrascritti correttamente nella tabella dal *file*, senza lasciare duplicati.

Questi *test* venivano ripetuti per ogni tabella interessata per controllare che il comportamento atteso fosse costantemente quello atteso.

Capitolo 4

Conclusione

4.1 Raggiungimento obiettivi

Il tirocinio è stato svolto nei tempi prefissati e gli obiettivi sono stati raggiunti, sia per quanto riguarda la formazione che il progetto da sviluppare.

4.2 Competenze acquisite

L'esperienza di tirocinio mi ha permesso di interagire per la prima volta con il framework Spring e quindi con l'esperienza di sviluppo di un back end. Ho trovato molto utili le conoscenze pregresse per argomenti come basi di dati, programmazione ad oggetti, linguaggio Java ed i protocolli web come punti di partenza per comprendere meglio i meccanismi che si trovano dietro ai servizi REST ed ai JPA. Tra gli strumenti utilizzati, ho apprezzato in particolare il sostegno offerto da Maven, SpringToolsSuite4 e il modulo Spring Boot per quanto riguarda l'impostazione di dipendenze e moduli per semplificare il lavoro dello sviluppatore.

4.3 Valutazione personale

Il percorso di tirocinio è risultato molto stimolante e motivante. Ho scelto di intraprendere questa esperienza prima della laurea, in quanto la considero molto importante per la crescita personale e per poter mettere in pratica concetti e basi appresi durante i corsi universitari. Mi ha permesso, infatti, di valutare le conoscenze acquisite durante l'università, ed evidenziarne le mancanze. Nonostante io avessi lavorato per la maggior parte in autonomia, sono riuscito comunque ad affrontare ed apprendere gli argomenti del programma proposto, senza particolare bisogno di sostegno. Ho avuto l'opportunità anche di mettermi in gioco e sfidare i miei limiti per quanto riguarda il lavoro in gruppo: inizialmente ho trovato difficile inserirmi in un nuovo contesto, tuttavia col tempo sono riuscito a confrontarmi sempre di più coi percorsi ed i progetti degli altri studenti tirocinanti e addirittura collaborare per certi aspetti come la ristrutturazione del modello della base dati. Personalmente, apprezzo particolarmente il concetto di lavoro di gruppo, anche durante progetti personali e universitari, dunque sono contento delle riflessioni e considerazione che mi ha portato a fare l'esperienza di tirocnio.

Elenco delle figure

Elenco dei frammenti di codice

Glossario

Agile Nell'ingegneria del software, si indica un insieme di metodi di sviluppo del software, direttamente o indirettamente derivati dai principi del "Manifesto per lo sviluppo agile del software", proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, software funzionante e di qualità. 2

Angular Un framework open-source basato su TypeScript per lo sviluppo di applicazioni web. Principalmente utilizzato per il lato front end. 2, 5

API In un programma informatico, con application programming interface si intende l'insieme di procedure,in genere raggruppate per strumenti specifici, atte a risolvere uno specifico problema di comunicazione tra diversi computer o tra diversi software o tra diversi componenti di software. 37

application.properties File di configurazione del progetto Spring che permette di definire specifiche proprietà. 12, 21, 34

back end In informatica, denota la parte che permette il funzionamento delle interazioni accessibili dal front end. Nello sviluppo dei siti web, corrisponde alla parte di amministrazione del sito. 2, 4, 5, 20

business logic In informatica, si riferisce a tutta quella logica di elaborazione (sotto forma di codice sorgente) che rende operativa un'applicazione. Nel caso di applicazioni web, si intende anche la gestione dello scambio di informazioni tra l'interfaccia utente, con eventuali elaborazioni intermedie sui dati estratti, ed eventualmente una sorgente dati (generalmente una base dati) deputata alla gestione della persistenza dei dati stessi. 14

CSS3 è un linguaggio che stabilisce le regole secondo cui un sito dovrà mostrare le informazioni, a livello di design e presentazione delle pagine web. 37

data source In informatica, è il luogo digitale in cui vengono salvati o creati dei dati. Spesso coincide con un file, una base dati o dati sul web. 12

Eclipse E' un ambiente di sviluppo integrato multi-linguaggio e multipiattaforma. 7

endpoint In informatica indica un luogo digitale esposto tramite un canale di comunicazione. Corrisponde al punto di un API dove avviene la comunicazione e lo scambio di risorse. 5, 7, 13–15, 20, 34

framework In informatica e nello sviluppo software, si intende un'architettura logica di supporto, spesso un'implementazione logica di un particolare design pattern, sulla quale un software può essere progettato e realizzato, generalmente facilitandone lo sviluppo da parte del programmatore. 2, 5, 9–11

front end In informatica denota la parte visibile all'utente di un programma e con cui può interagire. Nello sviluppo dei siti web, corrisponde alla parte visibile da chiunque e raggiungibile all'indirizzo web del sito. 2, 4, 5, 20

HTML5 è un linguaggio di formattazione strutturato ad albero che permette di strutturare pagine collegate fra di loro attraverso link. Dall'ultima versione supporta contenuti multimediali, geolocalizzazione e la creazione di applicazioni web complesse. 37

IoT E' un concetto che descrive oggetti fisici con sensori, capacità di elaborazione, software e altre tecnologie che si collegano e scambiano dati con altri dispositivi e sistemi attraverso il collegamento ad internet o altri network. 37

JSON formato per lo scambio di dati, in particolare per applicazioni client/server. 37

JUnit4 Un framework di testing di unità per il linguaggio Java. 30

Kanban In ambito di sviluppo software si fa riferimento ad una metodologia di sviluppo software dove i singoli compiti vengono considerati delle piccole fasi e portati a termine in sequenza, uno alla volta. 6

Node.js Un runtime system open-source e multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. 5

NoSql Un movimento che promuove sistemi software dove la persistenza dei dati è in generale caratterizzata dal fatto di non utilizzare il modello relazionale, di solito usato dalle basi di dati tradizionali. 5

REST In informatica, rappresenta un sistema di trasmissione di dati su HTTP senza ulteriori livelli, quali ad esempio SOAP. I sistemi REST non prevedono il concetto di sessione, ovvero sono stateless. Il funzionamento prevede una struttura degli URL ben definita che identifica univocamente una risorsa o un insieme di risorse e l'utilizzo dei metodi HTTP specifici. 2, 5, 7, 13–16, 20, 34

SCRUM In ambito di sviluppo software, è un framework agile per la gestione del ciclo di sviluppo, iterativo ed incrementale, concepito per gestire progetti e prodotti software o applicazioni di sviluppo. 2

tecniche di partizione In informatica, si intende le strategie per assegnare ad una partizione di memoria una determinata task. In questo modo, è possibile sfruttare in maniera più efficace la memoria a disposizione. 15

unmarshaller componente utilizzato per trasformare dati da un formato per salvataggio e trasmissione,ad esempio XML o JSON, in una rappresentazione attraverso oggetti. 27

URI In informatica, è una sequenza di caratteri che identifica universalmente ed univocamente una risorsa. Sono esempi di URI: un indirizzo web (URL), un documento, un indirizzo di posta elettronica. 37

XML In informatica, 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. 37

Acronimi

API Application Programming Interface (API). 5, 7, 15

CRUD Create, Remove, Update, Delete. 5, 12, 13, 20, 24

CSS3 Cascade Style Sheets 3 (CSS3). 2

DBMS Database Management System. 8

DI Dependency Injection. 9, 10, 34

DTO Data Transfer Object. 11–13, 20, 22, 24–26, 29, 30, 34

HTML5 HyperText Markup Language 5 (HTML5). 2

ICT Information and Communication Technologies. 2

IDE Integrated Development Environment. 7

IoC Inversion of Control. 9, 10

IoT Internet of Things (IoT). 2

JPA Jakarta Persistence API. 11–13, 19, 25, 33

JSON JavaScript Object Notation (JSON). 2

ORM Object-Relational Mapping. 11, 33

POM Project Object Model. 7

URI Uniform Resource Identifier (URI). 15

XML eXtensible Markup Language (XML). 3, 7, 21, 22, 27, 28, 34

Bibliografia

[1] Loredana Crusoveanu. *Intro to Inversion of Control and Dependency Injection with Spring*. URL: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring (cit. a p. 9).

[2] Matthew Tyson. *What is JPA? Introduction to Java persistence*. URL: https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html (cit. a p. 11).

[3] *Maven*. URL: https://maven.apache.org/index.html (cit. a p. 7).

[4] *Postman*. URL: https://www.postman.com/ (cit. a p. 7).

[5] *Smart city sim repository*. URL: https://github.com/SyncMonitor/SmartCitySimulator (cit. alle pp. 2, 3).

[6] *Smart city sim*. URL: https://syncmonitor.altervista.org/smartcityPadova/user (cit. a p. 4).

[7] Luciana Maci. *Che cos'è lo smart parking: le soluzioni e le tecnologie per il parcheggio intelligente*. 2020. URL: https://www.economyup.it/mobilita/che-cose-lo-smart-parking-le-soluzioni-e-le-tecnologie-per-il-parcheggio-intelligente/ (cit. a p. 2).

[8] *Spring*. URL: https://spring.io/projects (cit. alle pp. 9–11, 15).

[9] *Spring Batch - Reference Documentation*. URL: https://docs.spring.io/spring-batch/docs/current/reference/html/index.html (cit. alle pp. 15, 18).

[10] *Sync Lab*. URL: https://www.synclab.it/ (cit. a p. 1).

[11] *Sync Monitor*. URL: https://syncmonitor.altervista.org/ (cit. a p. 3).