Every line of code is worth the SWEat

Specifica Tecnica

27 marzo 2025

Uso	Esterno
Destinatari	Gruppo SWE@
	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Sync Lab S.r.L.
Responsabile	Andrea Perozzo
Redattori	Andrea Perozzo
	Andrea Precoma
	Davide Picello
	Klaudio Merja
	Riccardo Milan
Verificatori	Andrea Perozzo
	Andrea Precoma
	Davide Marin
	Davide Picello
	Klaudio Merja
	Riccardo Milan

Registro delle modifiche

Ver.	Data	Redattori	Verificatori	Descrizione
1.0.0	27/03/2025	Klaudio Merja	Andrea Precoma	• Redazione delle sezioni relative al *data broker* e *stream processor*
0.7.0	25/03/2025	Klaudio Merja	Andrea Precoma	• Redazione della sezione relativa al linguaggio Java
0.6.0	25/03/2025	Andrea Precoma	Klaudio Merja	• Redazione architetture del sistema
0.5.0	22/03/2025	Davide Picello	Riccardo Milan Andrea Precoma	• Completata sezione «Stato dei requisiti funzionali»
0.4.0	21/03/2025	Riccardo Milan	Davide Marin Davide Picello	• Aggiunta sezione Grafana
0.3.0	20/03/2025	Andrea Perozzo	Andrea Precoma Klaudio Merja	• Redatta sezione Database
0.2.0	18/03/2025	Andrea Precoma	Riccardo Milan Davide Picello	• Completate tecnologie del simulatore • Redatta sezione relativa al simulatore • Redatta sezione relativa ai *design pattern*
0.1.0	11/03/2025	Riccardo Milan	Klaudio Merja	• Costruita struttura del documento • Redazione introduzione del documento

Indice

Elenco delle immagini

1. Introduzione

1.1. Scopo del documento

Lo scopo principale del documento Specifiche Tecniche è quello di presentare in maniera dettagliata le scelte tecniche e tecnologiche effettuate dal gruppo al fine di sviluppare il prodotto oggetto del capitolato 4 *NearYou - Smart Custom Advertising Platform* dell'azienda proponente SyncLab S.r.l..

Il documento espone una descrizione dettagliata delle tecnologie, delle scelte architetturali e implementative e dei *design pattern* utilizzati dal gruppo per realizzare l'infrastruttura informatica che compone il prodotto *software* NearYou.

Il documento riporta inoltre il tracciamento dei requisiti funzionali soddisfatti come conseguenza dello sviluppo del prodotto corredate da grafici che ne attestano la copertura di questi ultimi.

1.2. Scopo del progetto

Il prodotto NearYou - Smart custom advertising platform è una piattaforma che sfrutta la GenAI[g] per la creazione di pubblicità personalizzate da mostrare a ciascun utente, sfruttando dati come la posizione trasmessi in tempo reale, le informazioni personali e i dati di profilazione, in maniera tale da migliorare l'esperienza finale dell'utente e aumentando contemporaneamente il ROI[g] e l'efficacia delle campagne pubblicitarie.

1.3. Glossario

Per evitare eventuali ambiguità e incomprensioni sulla terminologia adottata nella documentazione redatta dal gruppo, viene fornito un glossario. La prima occorrenza di un termine definito all'interno del glossario presente all'interno di un documento viene sottolineato e seguito dalla lettera «g» posta ad apice (e.g. termine[g]).

1.4. Riferimenti

1.4.1. Riferimenti normativi

- Norme di Progetto (v2.0.0)
  https://sweatunipd.github.io/docs/pb/norme_di_progetto_ver2.0.0.pdf
- Regolamento del progetto didattico, *slide* 23 (ultimo accesso in data 27/03/2025)
  https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf
- Capitolato C4 - Sync Lab S.r.l. (ultimo accesso in data 27/03/2025)
  https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C4.pdf

1.4.2. Riferimenti informativi

- Glossario (v2.0.0)
  https://sweatunipd.github.io/docs/pb/glossario_ver2.0.0.pdf
- Capitolato C4 - Sync Lab S.r.l. (ultimo accesso in data 27/03/2025)
  https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C4.pdf
- Guida ufficiale per l'installazione di Docker[g] (ultimo accesso in data 27/03/2025)
  https://docs.docker.com/engine/install
- Apache Flink[g] - Documentazione relativa al supporto di Flink a Java 17 (ultimo accesso in data 27/03/2025)
  https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/deployment/java_compatibility/#java-17

- Apache Flink - Documentazione relativa agli *async* I/O per l'accesso ai dati esterni (ultimo accesso in data 27/03/2025)
  https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/asyncio/#async-io-api
- Java *records* (ultimo accesso in data 27/03/2025)
  https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Record.html
- Apache Kafka[g] (ultimo accesso in data 27/03/2025)
  https://kafka.apache.org
- Documentazione della classe `ConnectionFactory` - R2DBC (ultimo accesso in data 27/03/2025)
  https://javadoc.io/doc/io.r2dbc/r2dbc-spi/latest/index.html
- Introduzione di LangChain4j - «*What if I don't have an API[g] key*» (ultimo accesso in data 27/03/2025)
  https://docs.langchain4j.dev/get-started

2. Tecnologie

In questa sezione vengono definite le tecnologie e gli strumenti adottati per lo sviluppo del prodotto *software* e le loro funzionalità all'interno del sistema. Per chiarezza, le tecnologie sono state suddivise in base al ruolo che svolgono all'interno del sistema.

2.1. Infrastruttura del sistema

2.1.1. Docker

Docker è una piattaforma *open source*[g] che permette di sviluppare, distribuire ed eseguire applicazioni. Mette a disposizione un ambiente isolato e permette di replicare l'ambiente di esecuzione in maniera tale da garantire portabilità e la replicabilità. Porta particolare vantaggio per quanto riguarda il suo utilizzo in ambito *testing*, permettendo di ridurre il tempo tra la codifica e il rilascio in produzione del prodotto.

2.1.2. Configurazione di Docker

La replicabilità del sistema viene resa possibile grazie a Docker Compose[g]. Questo strumento permette di definire e gestire applicazioni *multi-container*. In particolare, Docker Compose semplifica la creazione e la gestione dell'intero sistema, semplificando la configurazione dei servizi, delle reti e dei volumi dei singoli tramite un *file* di configurazione in YAML.

Nel nostro caso specifico, il file YAML si trova nella *root* del progetto denominato `compose.yml`. Questo *file* contiene tutti i servizi necessari a far funzionare il sistema. Ogni servizio è definito da un blocco di configurazione che specifica l'identificativo del servizio e le varie proprietà come l'immagine Docker da utilizzare, il comando da eseguire, porte da mappare e molto altro.

Di seguito vengono elencati tutti i servizi che compongono il sistema (le relative configurazioni vengono omesse e si rimanda il lettore al file `compose.yml` per ulteriori dettagli):

2.1.3. Servizi Docker implementati

- **kafka**
  - **Immagine**: apache/kafka
  - **Versione dell'immagine**: 4.0.0
  - **Riferimento**: https://hub.docker.com/r/apache/kafka (ultimo accesso in data 27/03/2025)
- **postgis**
  - **Immagine**: postgis/postgis
  - **Versione dell'immagine**: 17-3.5
  - **Riferimento**: https://hub.docker.com/r/postgis/postgis (ultimo accesso in data 27/03/2025)
- **grafana**
  - **Immagine**: rmilan/grafana-rm
  - **Riferimento**: https://hub.docker.com/r/rmilan/grafana-rm (ultimo accesso in data 27/03/2025)
- **jobmanager**
  - **Immagine**: flink
  - **Versione dell'immagine**: 1.20.1-scala_2.12-java17
  - **Riferimento**: https://hub.docker.com/_/flink (ultimo accesso in data 27/03/2025)
- **taskmanager**
  - **Immagine**: flink
  - **Versione dell'immagine**: 1.20.1-scala_2.12-java17
  - **Riferimento**: https://hub.docker.com/_/flink (ultimo accesso in data 27/03/2025)

- **simulator**
  - **Immagine**: *Build* in locale con Dockerfile
  - ***Path* del Dockerfile**: `./client` (relativo alla *root* del progetto)
  - **Riferimento**: **https://github.com/SWEatUNIPD/NearYou/tree/main/client**

2.2. Linguaggi di sviluppo

2.2.1. TypeScript

TypeScript è un *superset* di JavaScript che supporta sia la programmazione orientata agli oggetti con classi, interfacce ed ereditarietà, sia la programmazione procedurale basata su funzioni e blocchi di istruzioni.

2.2.1.1. Utilizzo nel progetto

TypeScript è stato utilizzato per la realizzazione del simulatore dei sensori, il quale si occupa di generare i dati di localizzazione di un percorso casuale per inviarli ad Apache Kafka mediante le API fornite dalla libreria Kafkajs.

2.2.1.2. Versione

La versione di TypeScript utilizzata per lo sviluppo del simulatore è la 5.7.2.

2.2.1.3. Librerie e framework

La seguente lista elenca le dipendenze utilizzate nel simulatore.

- **@mapbox/polyline**
  - **Documentazione**: https://www.npmjs.com/package/@mapbox/polyline (ultimo accesso in data 27/03/2025)
  - **Versione**: 1.2.1
  - **Descrizione**: Implementa un formato dell'algoritmo di Google per comprimere i dati di coordinate geografiche di un percorso.
- **Dotenv**
  - **Documentazione**: https://www.npmjs.com/package/dotenv (ultimo accesso in data 27/03/2025)
  - **Versione**: 16.4.7
  - **Descrizione**: Modulo che carica le variabili d'ambiente da un *file* `.env` in `process.env`. In questo modo si possono configurare dei parametri del sistema in modo agile.
- **Inversify**
  - **Documentazione**: https://inversify.io (ultimo accesso in data 27/03/2025)
  - **Versione**: 7.1.0
  - **Descrizione**: *Tool* utilizzato per gestire la *dependency injection* in applicativi sviluppati in JavaScript e TypeScript.
- **Kafkajs**
  - **Documentazione**: https://kafka.js.org/docs/introduction (ultimo accesso in data 27/03/2025)
  - **Versione**: 2.2.4
  - **Descrizione**: Usata per agevolare le operazioni di produzione e consumo di messaggi attraverso Apache Kafka.

Per effettuare i *test* e le analisi statiche del codice invece sono state utilizzate le seguenti librerie:
- **Vitest** per i *test* di unità
- **ESLint** per l'analisi statica del codice

2.2.2. Java

Java è un linguaggio di programmazione orientato agli oggetti che nasce con lo scopo di creare applicazioni indipendenti dalla piattaforma, grazie alla sua capacità di compilare il codice in *bytecode* ed eseguirlo su una JVM.

2.2.2.1. Utilizzo nel progetto

Nel nostro specifico caso, viene adottato per la creazione del servizio di *stream processing*[g] (denominato anche *job*) per Apache Flink. Questo si occupa di elaborare i dati di localizzazione in tempo reale provenienti dai sensori, garantendone la persistenza all'interno del *database*[g], e di arricchire tali dati con le informazioni necessarie a creare il *prompt*[g] da inviare alla LLM[g] per poter generare un annuncio il più personalizzato possibile.

2.2.2.2. Versione

Per soddisfare i requisiti di Apache Flink è stata adottato Java 17 LTS[g].

La documentazione di Apache Flink riporta l'introduzione in maniera sperimentale di Java 17 dalla versione 1.18 di Flink. Tuttavia il prodotto è stato validato e le funzionalità fondamentali che hanno un impatto dovuto a questa scelta, ovvero i *records* in Java, funzionano correttamente.

2.2.2.3. Librerie e framework

Per la gestione del progetto e l'automazione delle operazioni di *build* e *test* è stato utilizzato Apache Maven[g]. Per avere una visione nel dettaglio di tutte le librerie utilizzate all'interno del nostro sistema, è possibile visionare il *file* `pom.xml` presente all'interno della cartella `job` del nostro progetto.

La seguente lista rappresenta le dipendenze più rilevanti presenti all'interno del progetto e non vuole essere un mero elenco di tutte le dipendenze e librerie presenti all'interno del nostro progetto.

- **Apache Flink**
  - **Documentazione**: https://nightlies.apache.org/flink/flink-docs-release-1.20 (ultimo accesso in data 27/03/2025)
  - **Versione**: 1.20.1
  - **Descrizione**: *Framework* ed *engine* per effettuare operazioni *stateful*[g] su un flusso di dati (nel nostro caso dati di localizzazione), elevato o meno che sia, in tempo reale in maniera reattiva, scalabile e affidabile.
- **PostgreSQL JDBC Driver**
  - **Documentazione**: https://jdbc.postgresql.org/documentation (ultimo accesso in data 27/03/2025)
  - **Versione**: 42.7.5
  - **Descrizione**: *Driver*[g] per la connessione a un database PostgreSQL[g].
- **PostgreSQL R2DBC Driver**
  - **Documentazione**: https://github.com/pgjdbc/r2dbc-postgresql (ultimo accesso in data 27/03/2025)
  - **Versione**: 1.0.7.RELEASE
  - **Descrizione**: *Driver* per la connessione a un *database* PostgreSQL, sfruttando l'API della programmazione reattiva (per maggiori informazioni: Reactive Manifesto - ultimo accesso in data 27/03/2025). Permette di effettuare richieste asincrone e non bloccanti al *database*, come richiesto dalla documentazione di Flink per le operazioni di I/O asincrone.
- **R2DBC Pool**
  - **Documentazione**: https://github.com/r2dbc/r2dbc-pool (ultimo accesso in data 27/03/2025)
  - **Versione**: 1.0.2
  - **Descrizione**: Implementazione della *connection pool* per R2DBC.
- **Project Reactor**
  - **Documentazione**: https://projectreactor.io/docs/core/release/reference (ultimo accesso in data 27/03/2025)

  - **Versione**: 3.7.4
  - **Descrizione**: Libreria di programmazione reattiva, completamente non bloccante. Viene utilizzato all'interno del nostro prodotto come *client* per le operazioni asincrone di *data enrichment*. Nel nostro caso sono quella di richiesta del punto di interesse più vicino alla posizione del sensore che soddisfi gli interessi dell'utente e quella di generazione dell'annuncio tramite LLM sfruttando la profilazione dell'utente e i dati del punto di interesse registrati nel *database*.
- **SLF4J**
  - **Documentazione**: https://www.slf4j.org/docs.html
  - **Versione**: 2.0.17
  - **Descrizione**: Libreria che serve da astrazione (*facade*[g]) per i vari *framework* di *log*[g] in Java, permettendo di cambiare *framework* senza dover modificare il codice.
- **LogBack**
  - **Documentazione**: https://logback.qos.ch/documentation.html
  - **Versione**: 1.5.17
  - **Descrizione**: *Framework* di *logging* per Java, utilizzato per la configurazione dei *logger* all'interno del nostro servizio di *stream processing*.
- **LangChain4j**
  - **Documentazione**: https://docs.langchain4j.dev/intro
  - **Versione**: 1.0.0-beta1
  - **Descrizione**: Libreria per LLM che permette, nel nostro caso, la generazione di annunci personalizzati in base ai dati di profilazione dell'utente e del punto di interesse.

Per effettuare i *test* e le analisi statiche del codice invece sono state utilizzate le seguenti librerie:
- **JUnit** per i *test* di unità
- **TestContainers** per i test di *integrazione*
- **Mockito** per effettuare il *mocking* delle dipendenze
- **CheckStyle** per l'analisi statica del codice

2.3. Data broker

Il *data broker*[g] svolge un ruolo fondamentale all'interno del nostro sistema in quanto si occupa di ricevere i dati e inoltrarli ai servizi che ne fanno uso in maniera efficiente e scalabile. Nel nostro caso, il *data broker* riceve i dati di localizzazione dai sensori e li inoltra successivamente al servizio di *stream processing*.

2.3.1. Apache Kafka

Apache Kafka è una piattaforma di *streaming* di dati. Progettato per essere scalabile, *fault-tolerant* e ad avere elevate prestazioni, viene utilizzato per la gestione dei dati in tempo reale. In particolare nel nostro caso, Kafka viene utilizzato per creare le *pipeline*[g] di dati tra il simulatore e il servizio di *stream processing*.

Tuttavia non tutte le funzionalità che Apache Kafka fornisce ai programmatori sono state sfruttate all'interno del nostro progetto come la replicazione dei dati su più *broker*. Per la natura dimostrativa del progetto, è stato scelto di utilizzare un singolo *broker* Kafka, ma ciò non preclude l'utilizzo di questa funzionalità, che può avere impatti positivi sulla tolleranza agli errori del sistema.

2.3.2. Formato dei dati di localizzazione

Ogni dato di localizzazione emesso da un sensore (o simulatore nel nostro caso specifico) è composto da un oggetto JSON con le seguenti proprietà:

```JSON
{
  "rent_id": 1,
  "latitude": 45.123456,
  "longitude": 11.123456,
  "timestamp": 12345678910
}
```

- `rent_id`: ID del noleggio che ha emesso il dato di localizzazione (si rimanda il lettore alla sez. 2.6 per maggiori dettagli riguardo l'entità `Rent`).
- `latitude`: latitudine del noleggio che ha emesso il dato di localizzazione.
- `longitude`: longitudine del noleggio che ha emesso il dato di localizzazione.
- `timestamp`: timestamp del dato di localizzazione in millisecondi. Questo campo è fondamentale per la gestione della persistenza dei dati all'interno del *database* e per la generazione degli annunci, in quanto permette di evitare conflitti tra più dati di localizzazione emessi dallo stesso noleggio.

2.4. Stream processor

Lo *stream processor* è il cuore pulsante dell'intero sistema sviluppato dal gruppo. Esso si occupa dell'ingestione dei dati di localizzazione, di arricchirli di informazioni necessarie alla creazione del *prompt* da inviare all'LLM e di persistere questi ultimi all'interno del *database*.

2.4.1. Apache Flink

Apache Flink è un *framework* ed *engine* di *processing* distribuito che permette di eseguire delle operazioni definite *stateful* su uno *stream* di dati in entrata, limitati o meno che siano. È progettato per un funzionamento in modo continuo e con tempi di inattività e di risposta molto bassi.

Nel nostro caso Flink viene utilizzato per elaborare i dati di localizzazione in tempo reale provenienti dai sensori, garantendone la persistenza all'interno del *database* e, a partire da questi dati, trovare più informazioni possibili al fine di generare l'annuncio più adatto da spedire all'utente finale.

2.5. Generazione annunci

Il capitolato prevede l'utilizzo di LLM per la generazione degli annunci utilizzando come *prompt* gli interessi dell'utente finale, la categoria commerciale e l'offerta del punto di interesse più vicino alla posizione del sensore. Si fa notare che il peso maggiore dei dati è risieduto nei campi di testo libero (interessi dell'utente e offerta del punto di interesse) poiché le LLM sono specializzate proprio a interpretare queste tipologie di *input*.

2.5.1. LangChain4j

LangChain4j è una libreria Java che semplifica l'integrazione di LLM con applicazioni Java. Fornisce una serie di strumenti per lavorare con LLM, tra cui la creazione di *prompt* e la generazione di risposte tramite la LLM stessa. LangChain4j supporta diversi modelli di LLM: uno tra questi, da noi utilizzato, è GPT-4o-mini di OpenAI. Questo modello è stato utilizzato per un semplice motivo: permettere al gruppo di sviluppare il progetto in locale senza dover pagare l'API *key* di OpenAI, sfruttando la chiave `demo` fornita da LangChain[g] stessa. Il modello può essere comunque modificato in un secondo momento per permettere l'utilizzo di altri modelli grazie anche in parte alla modularità della libreria.

2.6. Database

2.6.1. PostgreSQL

Per la gestione dei dati relazionali è stato scelto PostgreSQL, un DBMS[g] che offre affidabilità e una certa flessibilità per l'estensione tramite *plugin* ed estensioni. Nel nostro contesto, all'avvio viene eseguito automaticamente lo *script* `create.sql` che crea lo schema del *database* (tabelle, relazioni, ecc.) secondo le esigenze del progetto e popola le tabelle di dati necessari a provare il funzionamento del nostro sistema.

2.6.2. PostGIS

Per l'elaborazione e l'archiviazione di dati geografici si fa uso dell'estensione PostGIS[g], la quale aggiunge a PostgreSQL il supporto per tipi, funzioni e indici spaziali.

In particolare l'immagine Docker utilizzata è postgis/postgis. Oltre a PostgreSQL questa contiene già la libreria PostGIS e le relative dipendenze. Questo *setup* permette, nel nostro caso, di:

- Persistere le coordinate geografiche (latitudine e longitudine) dei punti di interesse e delle posizioni trasmesse in tempo reale da ogni noleggio attivo.
- Effettuare *query* geospaziali all'interno del *database* per individuare i potenziali punti di interesse rispetto ad una determinata posizione ed entro un determinato *range*.

2.6.3. Struttura del database

2.6.3.1. Diagramma ER

Di seguito viene mostrata la struttura del *database*:

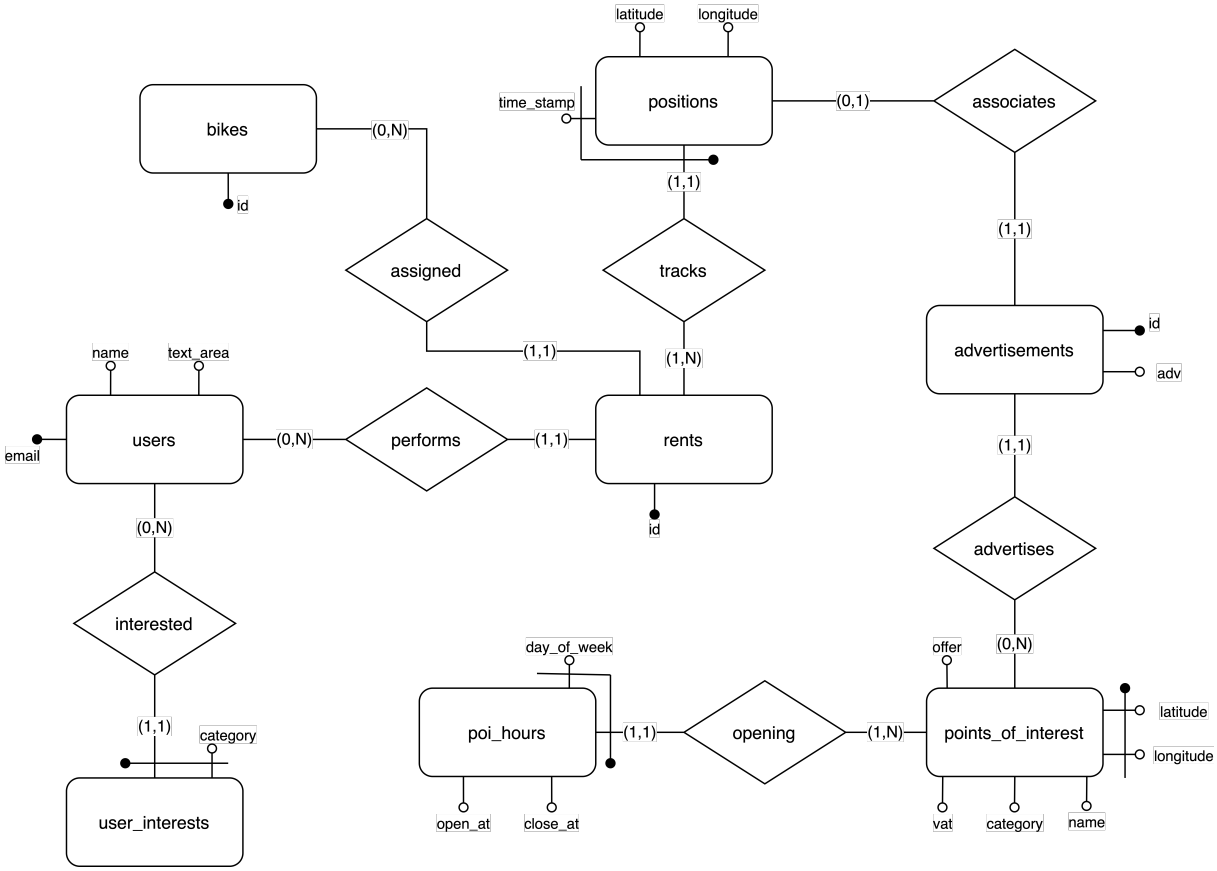

Figura 1: Diagramma ER

2.6.3.2. Scelte progettuali

Alcune scelte progettuali, apparentemente ridondanti, sono state adottate per soddisfare specifiche esigenze, in particolare per strumenti come Grafana[g].

- **Chiavi primarie composte:** La tabella `positions` utilizza una chiave primaria composta da `time_stamp` e `rent_id`. Questo garantisce l'univocità di ogni posizione registrata per un noleggio. Nella tabella `advertisements`, `position_time_stamp` e `position_rent_id` fungono da chiavi esterne per collegare un annuncio alla posizione di un noleggio.
- **Scelta delle chiavi primarie:** La tabella `points_of_interest` utilizza `latitude` e `longitude` come chiavi primarie per garantire che ogni punto di interesse sia univocamente identificabile in base alla sua posizione. Questo evita la creazione di ID artificiali e semplifica l'integrazione con strumenti GIS e di analisi spaziale. Tuttavia, in altre tabelle come `rents` o `advertisements`, è stato mantenuto un ID univoco separato per facilitare la visualizzazione e l'interazione con i dati nell'interfaccia di Grafana.

2.7. Interfaccia amministratore

L'interfaccia fornita dal *software* deve permettere all'amministratore di visualizzare la mappa con i punti di interesse, i sensori che si muovono e gli eventuali annunci generati. Deve disporre di una visualizzazione per lo storico degli annunci e una per entrare nel dettaglio di un singolo annuncio. Infine si mette a disposizione una sezione dedicata a dei grafici di analisi dei dati, utile all'amministratore per monitorare l'efficienza del servizio di noleggio.

2.7.1. Grafana

Grafana non è un sistema «reattivo», cioè non reagisce agli eventi, bensì si recupera i dati con delle *query* indipendentemente da cosa venga aggiornato nel *database*. Per questo motivo non è propriamente corretto parlare di «interfaccia *real time*», tuttavia le *query* vengono effettuate a intervalli molto ravvicinati simulando quindi con elevata accuratezza il tipo di interfaccia desiderata. Le funzionalità principali di Grafana nel nostro sistema sono:

- **Monitoraggio in tempo reale**: Grafana raccoglie in tempo reale i dati dei sensori registrati nel sistema, ovvero identificativo del sensore, noleggio associato ad esso, latitudine e longitudine.
- **Visualizzazione dei dati in tempo reale**: i dati dei sensori raccolti in tempo reale vengono infatti mostrati in una *dashboard*[g] di tipo *geomap* interattiva, nella quale le posizioni dei sensori sono rappresentate da *layer*[g] di tipo *route*[g] e i punti di interesse e gli annunci con *layer* di tipo *marker*[g].
- **Visualizzazione dei dati statici**: viene messa a disposizione dell'amministratore una *dashboard* che raccoglie lo storico degli annunci generati nel tempo; interagendo con i dati nel *database*, Grafana ci permette di mostrare tutti i dati rilevanti legati ad ogni annuncio.

3. Architettura del sistema

L'architettura definisce come i componenti di un'applicazione vengono distribuiti ed eseguiti su diversi ambienti. Nel caso di un sistema in *real time* si possono individuare separati servizi che comunicano reattivamente per inviare e processare i dati. Nel nostro progetto inoltre è presente un'interfaccia grafica che non agisce in seguito a una notifica, bensì a intervalli piccoli e regolari di tempo recupera i dati dal *database*. Abbiamo quindi optato per la K-*architecture* in quanto soddisfa tutte le caratteristiche del prodotto.

3.1. K-architecture

La K-*architecture* è un modello architetturale per l'elaborazione di dati in *streaming*. Derivante dalla λ-*architecture* la sua particolarità è l'eliminazione del *batching* mantenendo un flusso costante di dati in *real time*.

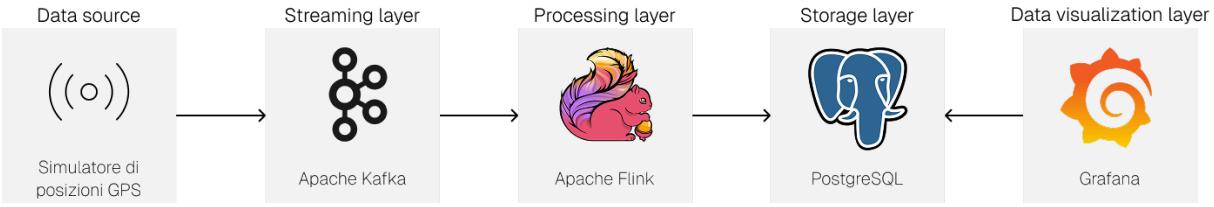

Figura 2: K-*architecture*

- ***Data source***: la sorgente di dati è costituita dal simulatore che imita l'attivazione dei noleggi e lo spostamento degli utenti sui mezzi. I sensori inviano quindi a intervalli regolari la posizione GPS[g] e ricevono l'eventuale annuncio.
- ***Streaming layer***: questo livello gestisce la trasmissione in tempo reale dei dati che vengono inoltrati al *processing layer*.
- ***Processing layer***: i dati ricevuti dallo *streaming layer* vengono processati in tempo reale prima di essere memorizzati in *database*.
- ***Storage layer***: la persistenza è gestita da un *database* relazionale che archivia i dati in arrivo dal *processing layer*. Lo *storage layer* è costituito da PostgreSQL affiancato da PostGIS, una estensione che facilita l'elaborazione di dati geospaziali.
- ***Data visualization*[g] *layer***: i dati archiviati nello *storage layer* vengono resi disponibile tramite una interfaccia grafica. Costituito da Grafana questo *layer* recupera le informazioni dal *database* a intervalli regolari in modo da aggiornare rapidamente l'interfaccia ai nuovi cambiamenti.

3.2. Flusso dei dati

Il diagramma sottostante descrive il percorso dei dati tra i *layer* del sistema e le relative elaborazioni.

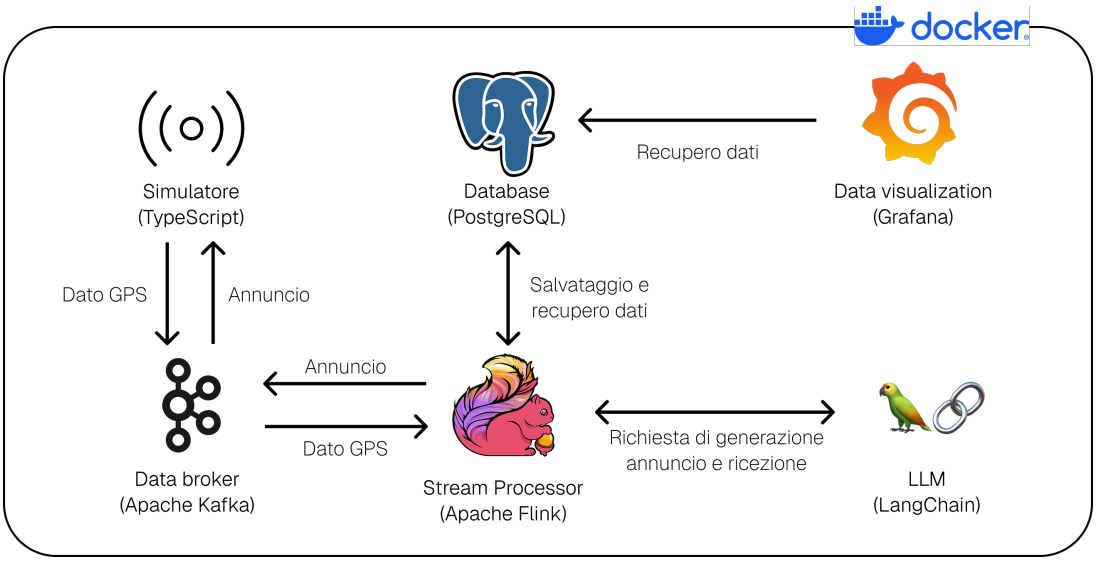

Figura 3: Flusso dei dati

1. **Generazione dei dati**: il simulatore genera in tempo reale dei percorsi tramite una chiamata API al servizio OpenStreetMap.
2. **Invio dei dati**: ogni sensore attivo del simulatore invia a intervalli regolari la posizione GPS in un Kafka *topic*[g].
3. ***Processing* dei dati**: lo *stream processor* è iscritto al *topic* delle posizioni GPS e riceve i messaggi dei sensori. Elabora quindi i dati ricevuti nel seguente modo:
   1. **Salvataggio in *database***: salva le posizioni nel *database*.
   2. **Ricerca del punto di interesse**: viene fatta la ricerca del punto di interesse più vicino alla posizione del sensore. Il punto di interesse dovrà soddisfare i seguenti requisiti:
      - Deve trovarsi entro un raggio di 100 metri dalla posizione del sensore.
      - La categoria commerciale del punto deve corrispondere a una delle categorie di interesse dell'utente, estratte dalla tabella `user_interests` del *database*.
      - Il punto di interesse deve essere aperto.

      Una volta trovato il punto di interesse, vengono estratte tutte le informazioni relative ad esso tra cui, la fondamentale, l'offerta del punto di interesse.
   3. **Richiesta di generazione dell'annuncio**: se la ricerca del punto di interesse ha avuto successo, vengono estratti gli interessi dell'utente a cui è destinato l'annuncio, in particolar modo la stringa libera che descrive i suoi interessi. Le informazioni del punto di interesse e dell'utente vengono successivamente utilizzate per costruire il *prompt* da inviare alla LLM.
   4. **Ricezione della risposta della LLM**: una volta processata la risposta da parte della LLM, il risultato viene inviato a una coda di Kafka denominata *adv-data* che salva questa generazione all'interno della tabella `advertisements`. Nel caso in cui la LLM ritornasse una stringa vuota, l'annuncio non viene mandato all'utente finale, ma viene gestita la sua persistenza a fini analitici per l'amministratore.

4. **Ricezione dell'eventuale annuncio**: il sensore riceve l'annuncio se questo è stato generato. In uno scenario reale l'annuncio verrebbe visualizzato dall'utente, ma il capitolato non prevedeva lo sviluppo dell'applicazione lato *client*.
5. **Visualizzazione grafica**: Grafana recupera le informazioni dal *database* a intervalli regolari ravvicinati per aggiornare costantemente la visuale dell'amministratore. Se questo richiede informazioni specifiche, ad esempio i dettagli di un annuncio, viene effettuata una *query* per recuperare i dati.

3.3. Diagramma delle classi

Il diagramma delle classi del sistema descrive le classi implementate e le loro relazioni. Vengono riportate tutte le classi di libreria, di cui si omettono attributi e metodi per non appesantire il diagramma, fatta eccezione per le classi ereditate o interfacce implementate da altre classi del *job* appartenenti alle librerie utilizzate.

Per migliorare la leggibilità sono omessi attributi e metodi delle classi di libreria. Inoltre il diagramma è stato diviso per sezioni a seconda del loro ruolo all'interno del sistema.

3.3.1. Infrastruttura

Questa sezione rappresenta le classi impiegate nella gestione dell'infrastruttura del sistema, ovvero quelle che si occupano della connessione al *database*, della serializzazione e deserializzazione dei dati e della creazione delle code di Kafka.

3.3.1.1. Struttura delle classi

3.3.1.1.1. GPSDataDto

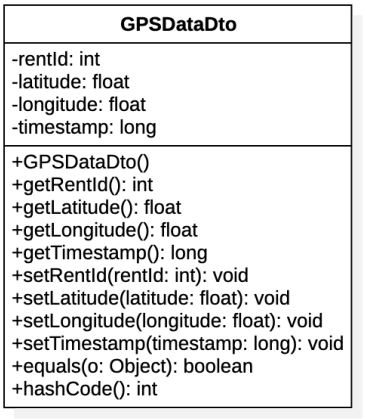

Figura 4: Diagramma della classe `GPSDataDto`

La classe `GPSDataDto` rappresenta il *data transfer object* (DTO) per i dati di localizzazione. Viene utilizzata per trasferire i dati di localizzazione tra il simulatore e il servizio di *stream processing*. Il dato viene inviato in formato JSON serializzato dal simulatore, per poi essere deserializzato dallo *stream processor* tramite la classe di utilità `GPSDataDeserializationSchema`, che estende la classe astratta `AbstractDeserializationSchema` di Flink.

3.3.1.1.1.1. Attributi

- `-rentId: int`: identificativo del noleggio associato al sensore.
- `-latitude: float`: latitudine della posizione GPS.
- `-longitude: float`: longitudine della posizione GPS.
- `-timestamp: long`: *timestamp* della posizione GPS.

3.3.1.1.1.2. Costruttori

Il costruttore della classe `GPSDataDto` è un costruttore di *default*. Non viene utilizzato il costruttore di *default* implicito fornito da Java in quanto la libreria di serializzazione/deserializzazione Jackson richiede un costruttore esplicito per la creazione di oggetti a partire da un JSON.

3.3.1.1.1.3. Metodi

- `+getRentId(): int`: restituisce l'identificativo del noleggio associato al sensore.
- `+getLatitude(): float`: restituisce la latitudine della posizione GPS.

- `+getLongitude(): float`: restituisce la longitudine della posizione GPS.
- `+getTimestamp(): long`: restituisce il *timestamp* della posizione GPS.
- `+setRentId(rentId: int): void`: imposta l'identificativo del noleggio associato al sensore.
- `+setLatitude(latitude: float): void`: imposta la latitudine della posizione GPS.
- `+setLongitude(longitude: float): void`: imposta la longitudine della posizione GPS.
- `+setTimestamp(timestamp: long): void`: imposta il *timestamp* della posizione GPS.
- `+equals(o: Object): boolean`: *overriding* del metodo della classe `Object` di Java, confronta l'uguaglianza tra l'oggetto `GPSDataDto` da cui viene invocato il metodo e l'oggetto posto a parametro e restituisce `true` se sono uguali, `false` altrimenti.
- `+hashCode(): int`: *overriding* del metodo della classe `Object` di Java, restituisce il codice *hash* dell'oggetto `GPSDataDto` da cui viene invocato il metodo.

Nel nostro caso, i *setters* della classe `GPSDataDto` sono stati implementati per garantire la deserializzazione del JSON che descrive il dato di localizzazione che viene spedito dal simulatore.

3.3.1.1.2. KafkaTopicService

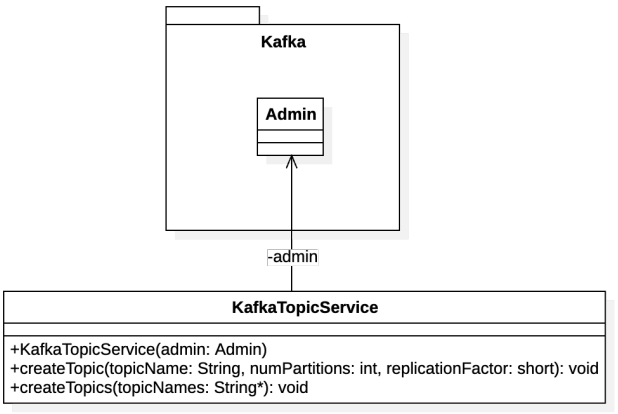

Figura 5: Diagramma della classe `KafkaTopicService`

La classe `KafkaTopicService` rappresenta il servizio di creazione dei *topic* di Kafka. Viene impiegato dal *job* per la creazione dei topic «gps-*data*» e «*adv-data*» se non già creati, utilizzati rispettivamente per la ricezione dei dati di localizzazione e l'invio degli annunci generati dalla LLM.

3.3.1.1.2.1. Attributi

- `-admin: Admin`: oggetto fornito dalla libreria Kafka per la gestione dei *topic*. Nel nostro caso è costante per ogni istanza dell'oggetto.

3.3.1.1.2.2. Costruttori

- `+KafkaTopicService(admin: Admin)`: costruttore della classe `KafkaTopicService` che inizializza l'attributo `admin` con il valore passato come parametro.

3.3.1.1.2.3. Metodi

- `+createTopic(topicName: String, numPartitions: int, replicationFactor: short): void`: metodo che crea un *topic* di Kafka con il nome e le caratteristiche specificate. Se il *topic* esiste già, non viene creato nuovamente.
- `+createTopics(topicNames: String*): void`: metodo che crea più *topic* di Kafka con i nomi specificati all'interno dell'*array* con numero di partizioni e di *replication factor* pari a 1. Se i *topic* esistono già, non vengono creati nuovamente.

3.3.1.1.3. GPSDataDeserializationSchema

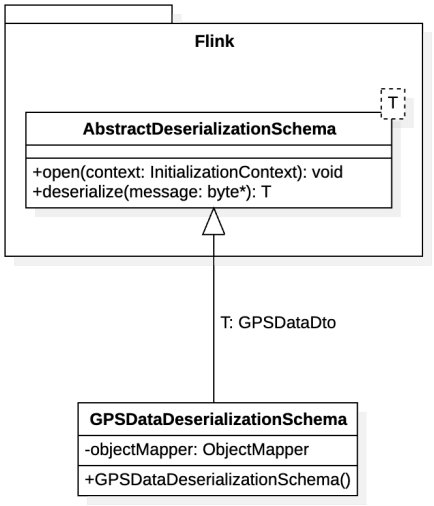

Figura 6: Diagramma della classe `GPSDataDeserializationSchema`

La classe `GPSDataDeserializationSchema` estende la classe astratta parametrica `AbstractDeserializationSchema<T>` di Flink e ridefinisce i metodi `deserialize` e `open`, dove T rappresenta la classe in cui viene deserializzato il JSON. In questo caso, i dati in arrivo sono in formato JSON e vengono deserializzati in un oggetto `GPSDataDto`, nonché parametro della classe estesa.

3.3.1.1.3.1. Attributi

- `-objectMapper: ObjectMapper`: oggetto fornito dalla libreria Jackson (interno alla dipendenza di Flink) per la serializzazione e deserializzazione di oggetti in formato JSON.

3.3.1.1.3.2. Costruttori

Viene mantenuto il costruttore di *default* fornito da Java.

3.3.1.1.3.3. Metodi

- `+open(context: InitializationContext): void`: metodo che viene invocato all'avvio del processo di deserializzazione. Viene invocato per effettuare il *setup* dell'oggetto. In questo caso, viene inizializzato l'oggetto `objectMapper` per la deserializzazione del JSON.
- `+deserialize(message: byte[]): GPSDataDto`: metodo che viene invocato per deserializzare il messaggio in arrivo. In questo caso, il messaggio viene deserializzato in un oggetto `GPSDataDto` tramite l'utilizzo dell'oggetto `objectMapper`.

3.3.1.1.4. AdvertisementSerializationSchema

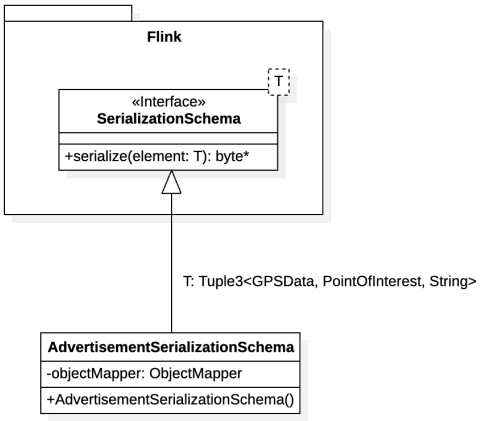

Figura 7: Diagramma della classe `AdvertisementSerializationSchema`

La classe `AdvertisementSerializationSchema` implementa l'interfaccia parametrica `SerializationSchema<T>` di Flink e ridefinisce il metodo `serialize`, dove T rappresenta la classe oggetto della serializzazione. Nel nostro caso, T corrisponde a una classe `Tuple3<GPSData, PointOfInterest, String>`, che rappresenta l'annuncio generato dalla LLM, comprensivo dei dati del punto di interesse e di localizzazione. La classe `AdvertisementSerializationSchema` viene utilizzata per serializzare l'annuncio in un messaggio JSON da inviare al *topic* «*adv-data*» di Kafka che simula la comunicazione del messaggio all'utente.

3.3.1.1.4.1. Attributi

- `-objectMapper: ObjectMapper`: oggetto fornito dalla libreria Jackson (interno alla dipendenza di Flink) per la serializzazione e deserializzazione di oggetti in formato JSON.

3.3.1.1.4.2. Costruttori

Viene mantenuto il costruttore di *default* fornito da Java.

3.3.1.1.4.3. Metodi

- `+serialize(adv: Tuple3<GPSData, PointOfInterest, String>): byte*`: metodo che viene invocato per serializzare l'annuncio.

3.3.1.1.5. DatabaseConnectionSingleton

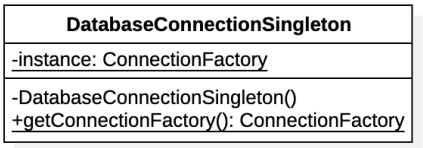

Figura 8: Diagramma della classe `DatabaseConnectionSingleton`

La classe `DatabaseConnectionSingleton` rappresenta il *singleton* della `ConnectionFactory` fornita dalla libreria R2DBC. Essa viene utilizzata per istanziare in maniera univoca durante tutto il processo una *factory* di connessioni, creabili attraverso il metodo `create()`.

3.3.1.1.5.1. Attributi

- `-instance: ConnectionFactory`: istanza della `ConnectionFactory` che viene creata globalmente per l'intero processo grazie all'implementazione del *design pattern singleton*.

3.3.1.1.5.2. Costruttori

Il costruttore viene reso inutilizzabile attraverso la ridefinizione del costruttore di *default*, ponendo l'accesso privato per implementare il *design pattern singleton*.

3.3.1.1.5.3. Metodi

- `+getConnectionFactory(): ConnectionFactory`: metodo statico che restituisce l'istanza della `ConnectionFactory`. Se l'istanza non è ancora stata creata, viene creata e restituita. Il metodo è *synchronized* per garantire che venga creata una sola istanza della `ConnectionFactory` durante l'intero processo.

3.3.2. Entità

Questa sezione rappresenta le entità del sistema, ovvero le classi che rappresentano i dati persistenti all'interno del *database*.

3.3.2.1. Struttura delle classi

3.3.2.1.1. GPSData

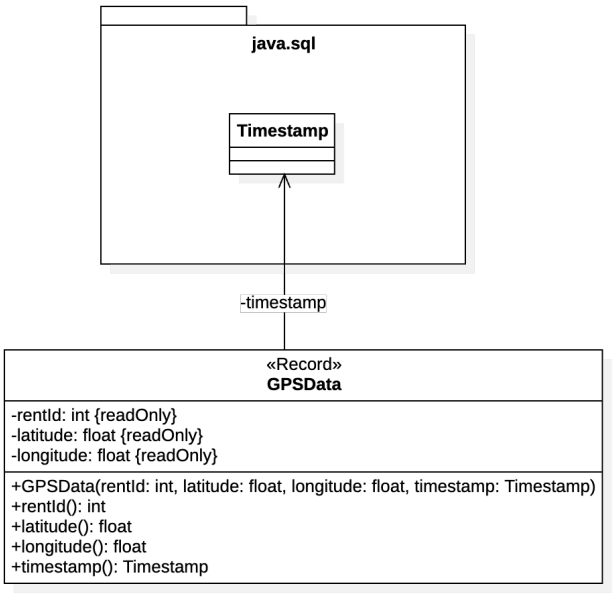

Figura 9: Diagramma della classe `GPSData`

La classe `GPSData` rappresenta il dato di localizzazione GPS, pressoché simile al DTO `GPSDataDto`, ma permette di rappresentare l'attributo `timestamp` tramite l'oggetto `Timestamp` di Java (`java.sql.Timestamp`), facilitando così le operazioni da effettuare tramite il *database*. All'interno della *codebase*[g], l'entità `GPSData` non è rappresentata tramite una classe, ma tramite i *record* in Java. Questi permettono di definire delle classi immutabili, ovvero non modificabili una volta create. Tuttavia, per mantenere la rappresentazione UML[g], il gruppo ha deciso di rappresentare la classe come un oggetto aventi attributi costanti e metodi *getter*.

3.3.2.1.1.1. Attributi

In quanto *record*, gli attributi della classe `GPSData` sotto elencati sono costanti data l'immutabilità della classe per definizione.

- `-rentId: int`: identificativo del noleggio associato al sensore.
- `-latitude: float`: latitudine della posizione GPS.
- `-longitude: float`: longitudine della posizione GPS.
- `-timestamp: Timestamp`: *timestamp* della posizione GPS.

3.3.2.1.1.2. Costruttori

- `+GPSData(rentId: int, latitude: float, longitude: float, timestamp: Timestamp)`: costruttore della classe `GPSData` che inizializza gli attributi `rentId`, `latitude`, `longitude` e `timestamp` con i valori passati come parametro.

3.3.2.1.1.3. Metodi

I metodi *getter*, nella reale implementazione della classe, vengono omessi in quanto creati dal *record*.

- `+rentId(): int`: restituisce l'identificativo del noleggio associato al sensore.
- `+latitude(): float`: restituisce la latitudine della posizione GPS.
- `+longitude(): float`: restituisce la longitudine della posizione GPS.
- `+timestamp(): Timestamp`: restituisce il *timestamp* della posizione GPS.
- `+equals(o: Object): boolean`: *overriding* del metodo della classe `Object` di Java, confronta l'uguaglianza tra l'oggetto `GPSData` da cui viene invocato il metodo e l'oggetto posto a parametro e restituisce `true` se sono uguali, `false` altrimenti.
- `+hashCode(): int`: *overriding* del metodo della classe `Object` di Java, restituisce il codice *hash* dell'oggetto `GPSData` da cui viene invocato il metodo.

3.3.2.1.2. PointOfInterest

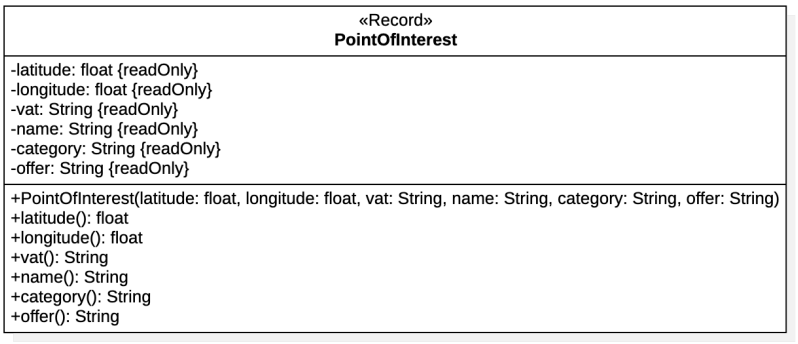

Figura 10: Diagramma della classe `PointOfInterest`

La classe `PointOfInterest` rappresenta un qualsiasi punto di interesse presente all'interno del *database*. Come per la classe `GPSData`, anche in questo caso è stato scelto di rappresentare l'entità tramite un *record* in Java.

3.3.2.1.2.1. Attributi

In quanto *record*, gli attributi della classe `PointOfInterest` sotto elencati sono costanti.

- `-latitude: float`: latitudine del punto di interesse.
- `-longitude: float`: longitudine del punto di interesse.
- `-vat: String`: partita IVA del punto di interesse.
- `-name: String`: nome del punto di interesse.
- `-category: String`: categoria del punto di interesse.
- `-offer: String`: offerta del punto di interesse, ovvero ciò che il punto mette a disposizione per i clienti e che possa essere fondamentale per il contesto degli annunci pubblicitari.

3.3.2.1.2.2. Costruttori

- `+PointOfInterest(latitude: float, longitude: float, vat: String, name: String, category: String, offer: String)`: costruttore della classe `PointOfInterest` che inizializza gli attributi `latitude`, `longitude`, `vat`, `name`, `category` e `offer` con i valori passati come parametro.

3.3.2.1.2.3. Metodi

I metodi *getter*, nella reale implementazione della classe, vengono omessi in quanto creati dal *record*.

- `+latitude(): float`: restituisce la latitudine del punto di interesse.
- `+longitude(): float`: restituisce la longitudine del punto di interesse.
- `+vat(): String`: restituisce la partita IVA del punto di interesse.
- `+name(): String`: restituisce il nome del punto di interesse.
- `+category(): String`: restituisce la categoria del punto di interesse.
- `+offer(): String`: restituisce l'offerta del punto di interesse.
- `+equals(o: Object): boolean`: *overriding* del metodo della classe `Object` di Java, confronta l'uguaglianza tra l'oggetto `PointOfInterest` da cui viene invocato il metodo e l'oggetto posto a parametro e restituisce `true` se sono uguali, `false` altrimenti.
- `+hashCode(): int`: *overriding* del metodo della classe `Object` di Java, restituisce il codice *hash* dell'oggetto `PointOfInterest` da cui viene invocato il metodo.

3.3.3. Richieste asincrone

Questa sezione rappresenta le classi che si occupano di effettuare le richieste asincrone per arricchire i dati al fine di migliorare il *prompt* da inviare all'LLM. Esse estendono la classe astratta `RichAsyncFunction` della libreria di Flink per eseguire queste richieste su ogni dato dello *stream*.

3.3.3.1. Struttura delle classi

3.3.3.1.1. NearestPOIRequest

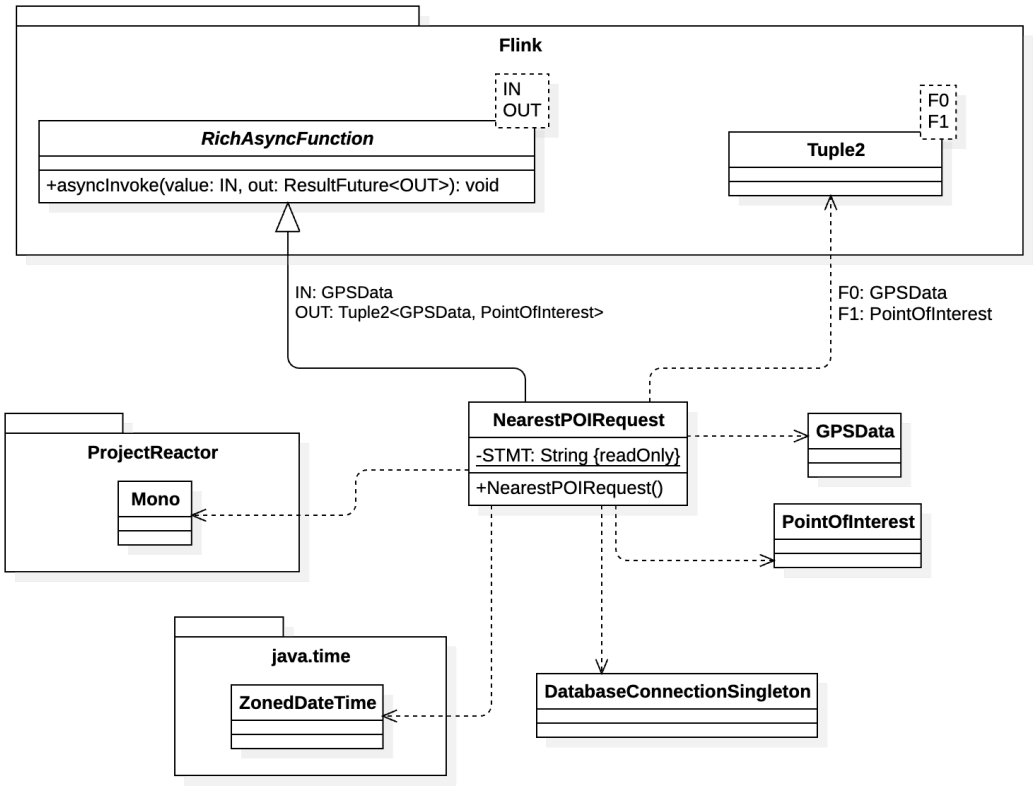

Figura 11: Diagramma della classe `NearestPOIRequest`

La classe `NearestPOIRequest` rappresenta la richiesta asincrona di ricerca del punto di interesse più vicino alla posizione del sensore. La classe estende la classe astratta parametrica `RichAsyncFunction<IN, OUT>` fornita dalla libreria di Flink con parametri `IN:GPSData` e `OUT:Tuple2<GPSData,PointOfInterest>`, che permette di eseguire operazioni asincrone all'interno di

un *job* in Flink. In questo caso, viene fatta una richiesta al *database* per la ricerca del punto di interesse più vicino, sfruttando le *query* geospaziali fornite dall'estensione PostGIS. Come menzionato in precedenza, in quanto Flink richiede un *client* asincrono per le operazioni con il *database*, è stato scelto di utilizzare le libreria R2DBC e Project Reactor per effettuare richieste non bloccanti al *database*.

3.3.3.1.1. Attributi

- `-STMT: String`: *statement* SQL da eseguire per la ricerca del punto di interesse più vicino alla posizione del sensore. L'attributo è statico e costante.

3.3.3.1.2. Costruttori

Viene mantenuto il costruttore implicito di *default* fornito da Java, in quanto non sono necessari costruttori specifici per la classe `NearestPOIRequest`.

3.3.3.1.3. Metodi

In quanto classe che estende `RichAsyncFunction`, la classe `NearestPOIRequest` ridefinisce solamente il metodo `asyncInvoke`, mantenendo l'implementazione *default* della classe estesa per i metodi `open` e `close`.

- `+asyncInvoke(gpsData: GPSData, resultFuture: ResultFuture<Tuple2<GPSData, PointOfInterest>>): void`: metodo che viene invocato in modo asincrono per eseguire la ricerca del punto di interesse più vicino alla posizione del sensore. Il risultato della ricerca viene restituito tramite il parametro `resultFuture`, che rappresenta il risultato della richiesta asincrona.

3.3.3.1.2. AdvertisementGenerationRequest

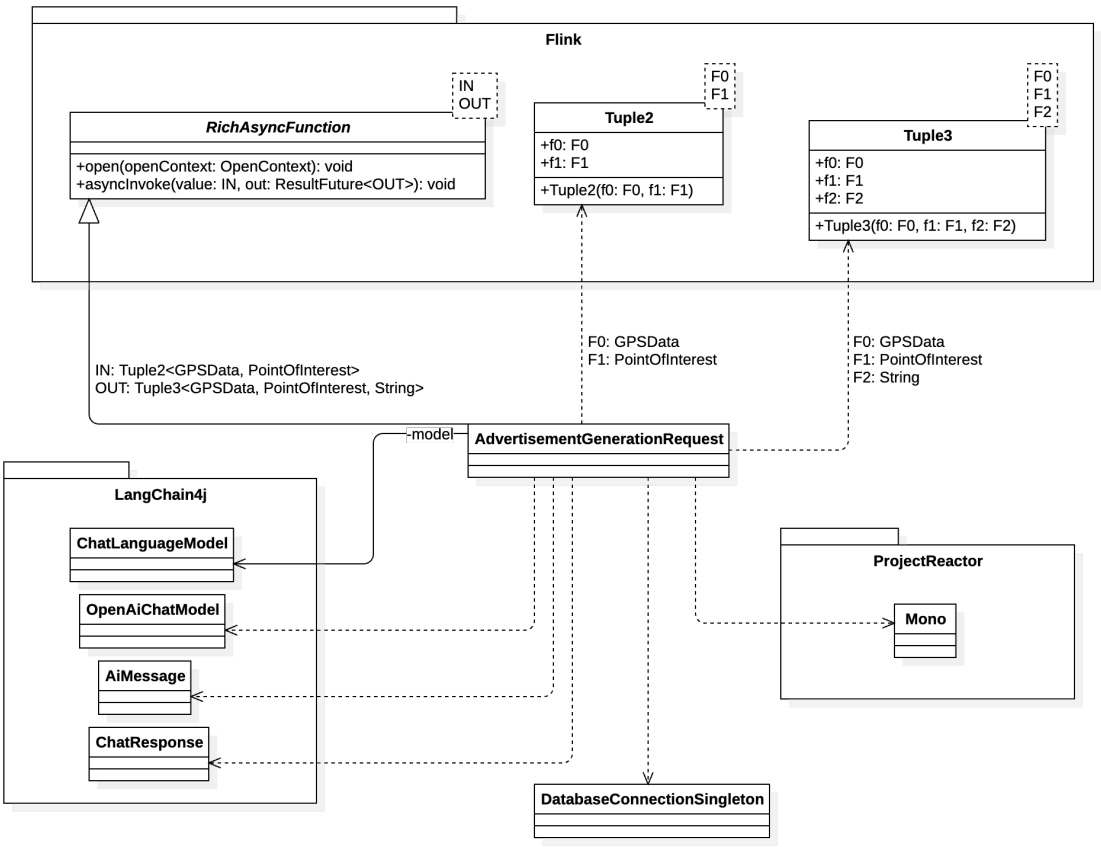

Figura 12: Diagramma della classe `AdvertisementGenerationRequest`

La classe `AdvertisementGenerationRequest` rappresenta la richiesta asincrona di generazione dell'annuncio da inviare alla LLM. Estende la classe astratta `RichAsyncFunction<IN, OUT>` fornita

dalla libreria di Flink con parametri `IN:Tuple2<GPSData,PointOfInterest>` e `OUT:Tuple3<GPSData, PointOfInterest, String>`, che permette di eseguire operazioni asincrone all'interno di un *job* in Flink. In questo caso viene fatta una richiesta alla LLM per la generazione dell'annuncio, sfruttando le API fornite dalla libreria LangChain4j.

3.3.3.1.2.1. Attributi

- `-model: ChatLanguageModel`: rappresenta il modello LLM utilizzato per effettuare le richieste di generazione degli annunci.

3.3.3.1.2.2. Costruttori

Viene mantenuto il costruttore implicito di *default* fornito da Java, in quanto non sono necessari costruttori specifici per la classe `AdvertisementGenerationRequest`.

3.3.3.1.2.3. Metodi

In quanto classe che estende `RichAsyncFunction`, la classe `AdvertisementGenerationRequest` ridefinisce solamente il metodo `asyncInvoke`, mantenendo l'implementazione *default* della classe estesa per i metodi `open` e `close`.

- `+asyncInvoke(Tuple2<GPSData, PointOfInterest> input, ResultFuture<Tuple3<GPSData, PointOfInterest, String>> resultFuture): void`: metodo che viene invocato in modo asincrono per eseguire la generazione dell'annuncio. Il risultato della generazione viene restituito tramite il parametro `resultFuture` che rappresenta il risultato della richiesta asincrona.

3.3.4. Classe main

La classe *main* del *job* di Flink si occupa di eseguire il *job*, creando l'*execution environment* e avviando il processo di *stream processing*. Essa si occupa di organizzare le varie componenti del sistema, creando i *topic* di Kafka e avviando il processo di *stream processing*.

3.3.4.1. Struttura della classe: attributi, costruttori e metodi

3.3.4.2. DataStreamJob

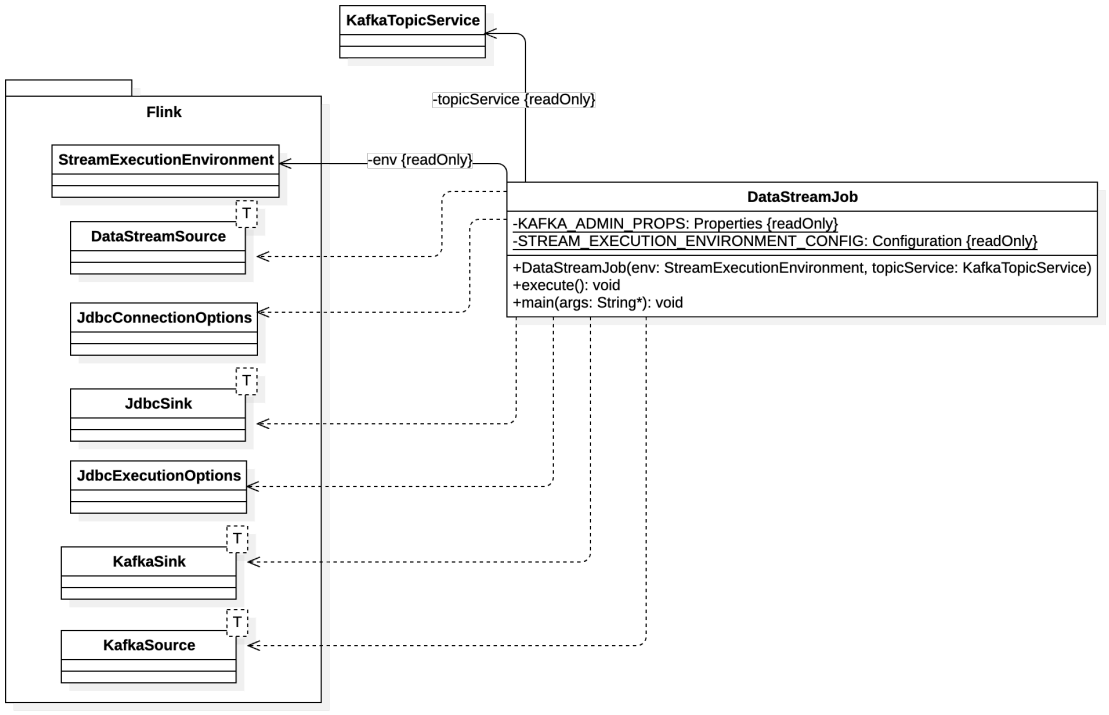

Figura 13: Diagramma della classe `DataStreamJob`

La classe `DataStreamJob` è la *main class* del *job* di Flink. Essa avvia l'esecuzione dello *stream processor*, organizzando le varie componenti di quest'ultimo e gestendo il flusso dei dati tra i vari *layer* del sistema.

3.3.4.3. Attributi

- `-env: StreamExecutionEnvironment`: oggetto fornito dalla libreria Flink per la gestione dell'ambiente di esecuzione del *job*. L'attributo è costante.
- `-topicService: KafkaTopicService`: istanza della classe `KafkaTopicService` per la creazione dei *topic* necessari al *job*. L'attributo è costante.
- `-KAFKA_ADMIN_PROPS: Properties`: oggetto che contiene tutte le configurazioni dell'*admin* di Kafka. L'attributo è statico e costante.
- `-STREAM_EXECUTION_ENVIRONMENT_CONFIG: Configuration`: oggetto che contiene tutte le configurazioni dell'*execution environment* di Flink. L'attributo è statico e costante.

3.3.4.4. Costruttori

- `+DataStreamJob(env: StreamExecutionEnvironment, topicService: KafkaTopicService)`: costruttore della classe `DataStreamJob` che inizializza gli attributi `env` e `topicService` con i valori passati come parametro.

3.3.4.5. Metodi

- `+execute(): void`: metodo che avvia l'esecuzione del *job* di Flink. Inizializza i *topic* di Kafka, crea il flusso di dati e avvia l'esecuzione del *job*.
- `+main(args: String*)`: metodo *main* della classe `DataStreamJob` che avvia l'esecuzione del *job* di Flink. Inizializza l'*execution environment*, il servizio di creazione dei *topic* e il *job* stesso.

3.4. Design pattern adottati

3.4.1. Singleton

Il *design pattern singleton* è uno dei *pattern* creazionali della GoF (*Gang of Four*) che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza.

Nel nostro caso questo *pattern* è stato adottato per garantire la creazione di una singola istanza di `ConnectionFactory`, classe di utilità fornita dalla libreria R2DBC per la creazione di connessioni al *database* PostgreSQL ottenute tramite la *connection pool* gestita internamente dal *driver* R2DBC utilizzando il metodo `create`.

3.4.1.1. Integrazione del design pattern nel progetto

Il concetto di *singleton* prevede la presenza di un'istanza univoca per l'intero sistema. Si verifica quindi l'accesso ad una risorsa condivisa da più parti del sistema, ponendo il problema dell'accesso concorrente alla risorsa.

Per garantire la soluzione a questo problema è stata fornita un'implementazione *thread-safe* del *design pattern singleton* utilizzando il meccanismo di sincronizzazione di Java all'unico punto di accesso dell'istanza globale, ovvero il metodo `getConnectionFactory`, per garantire l'accesso univoco e atomico alla risorsa.

DatabaseConnectionSingleton.java Java

```java
public class DatabaseConnectionSingleton {
  private static ConnectionFactory instance;

  private DatabaseConnectionSingleton() { }

  public static synchronized ConnectionFactory getConnectionFactory() {
    if (instance == null) {
      instance =
        ConnectionFactories.get(
          ConnectionFactoryOptions.builder()
            .option(ConnectionFactoryOptions.DRIVER, "pool")
            .option(ConnectionFactoryOptions.PROTOCOL, "postgresql")
            .option(
              ConnectionFactoryOptions.HOST,
              System.getProperty("postgres.hostname", "postgis"))
            .option(
              ConnectionFactoryOptions.PORT,
              Integer.parseInt(System.getProperty("postgres.port", "5432")))
            .option(
              ConnectionFactoryOptions.USER,
              System.getProperty("postgres.username", "admin"))
            .option(
              ConnectionFactoryOptions.PASSWORD,
              System.getProperty("postgres.password", "adminadminadmin"))
            .option(
              ConnectionFactoryOptions.DATABASE,
              System.getProperty("postgres.dbname", "admin"))
```

```
28              .build());
29      }
30      return instance;
31    }
32  }
```

4. Struttura del simulatore

Nonostante il simulatore di posizioni GPS non sia un componente centrale del sistema, ma semplicemente un *mock* per generare dati di *test* richiesto dall'azienda proponente nell'ambito del progetto didattico, il gruppo ha comunque intrapreso un approccio progettuale.

Di seguito verrà descritto nel dettaglio la struttura del simulatore presentando il diagramma delle classi, i ruoli di ogni componente all'interno del simulatore e le scelte progettuali adottate.

4.1. Diagramma delle classi

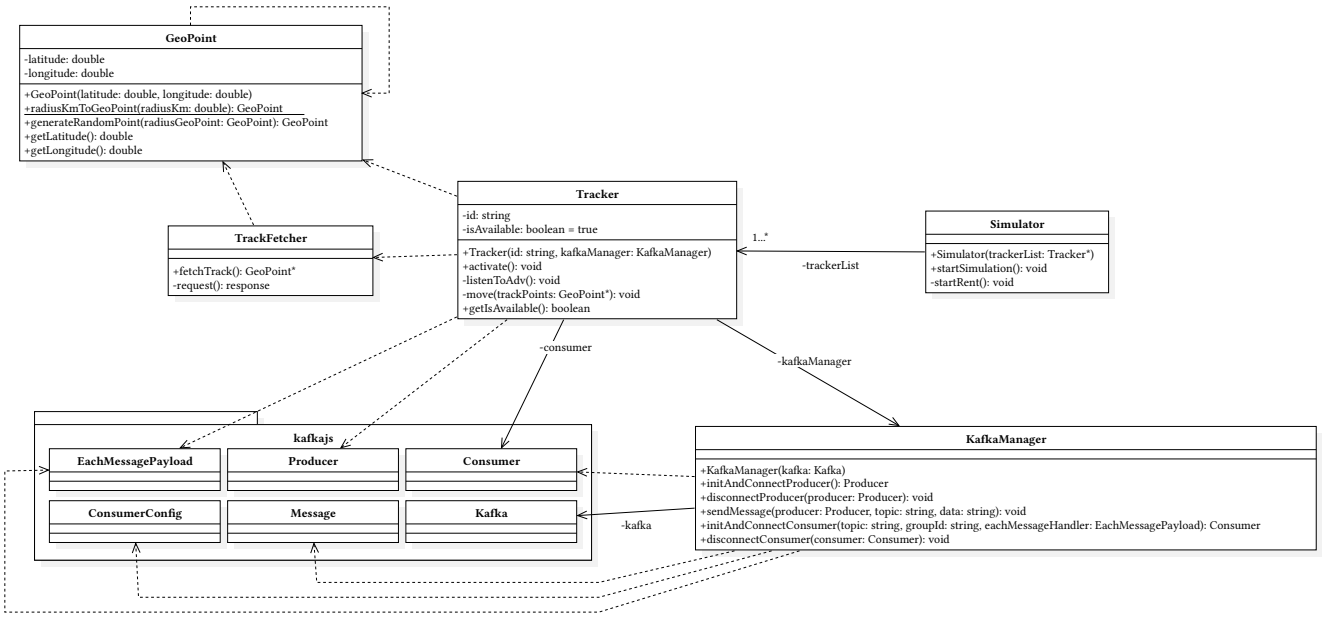

Figura 14: Diagramma delle classi del simulatore

4.1.1. Struttura delle classi

4.1.1.1. Simulator

La classe `Simulator` rappresenta il simulatore vero e proprio. Questo riceve i mezzi disponibili dal servizio di noleggio e attiva i primi «x» (valore indicato dalla variabile d'ambiente `INIT_RENT_COUNT`). Poiché è fondamentale ne esista una sola istanza viene adottato il *design pattern singleton*. Non ne viene indicata l'implementazione perché questa viene gestita dalla libreria Inversify (<u>sez. 4.2.2</u>).

4.1.1.1.1. Attributi

- `-trackerList: Tracker*`: lista dei sensori disponibili, uno per ogni mezzo. Vengono istanziati con un identificatore incrementale partendo da 1.

4.1.1.1.2. Costruttore

- `+Simulator(trackerList: Tracker*)`: costruttore della classe `Simulator` che inizializza la lista locale con i sensori passati per parametro.

4.1.1.1.3. Metodi

- `+startSimulation(): void`: avvia la simulazione attivando i primi «x» sensori.
- `-startRent(): void`: prende il primo sensore in ordine di identificativo crescente e, se disponibile, lo attiva.

4.1.1.2. Tracker

La classe `Tracker` rappresenta un sensore installato su un mezzo. Quando viene attivato richiede un percorso verosimile da seguire, manda le posizioni GPS al sistema e riceve gli eventuali annunci che vengono generati.

4.1.1.2.1. Attributi

- `-id: string`: l'identificativo univoco del sensore, e quindi anche del mezzo.
- `-isAvailable: boolean = true`: indica se il mezzo è disponibile e quindi può essere noleggiato. Per ogni istanza di `Tracker` viene sempre inizializzato a `true`.
- `-kafkaManager: KafkaManager`: riferimento al *manager* dell'istanza di Kafka.
- `-consumer!: Consumer`: una istanza di un *consumer* di Kafka che rimane in ascolto degli eventuali annunci. Viene inizializzata nel metodo `listenToAdv`, non nel costruttore.

4.1.1.2.2. Costruttore

- `+Tracker(id: string, kafkaManager: KafkaManager)`: costruttore della classe `Tracker` che riceve l'identificativo del sensore e il *manager* di Kafka.

4.1.1.2.3. Metodi

- `+activate(): void`: attiva il sensore, quindi lo rende indisponibile, richiama `listenToAdv` per rimanere in ascolto nel *topic* degli annunci, richiede un tracciato da seguire e richiama `move`.
- `-listenToAdv(): void`: utilizza `kafkaManager` per creare un *consumer* iscritto al *topic* «*adv-data*». Il metodo richiamato alla ricezione di un messaggio è vuoto perché il progetto non prevede di far visualizzare l'annuncio all'utente.
- `-move(trackerPoints: GeoPoint*): void`: riceve il tracciato da seguire sotto forma di lista di `GeoPoint`, istanzia un *producer* utilizzando `kafkaManager` e invia i dati GPS al *topic* «gps-*data*» a intervalli regolari. Quando termina il percorso disconnette il *producer* e il *consumer* terminando il noleggio, reimpostando cioè `isAvailable` a `true`.
- `+getIsAvailable(): boolean`: restituisce il valore della variabile `isAvailable`.

4.1.1.3. KafkaManager

La classe `KafkaManager` gestisce le operazioni fondamentali dei *producer* e *consumer* dell'oggetto di `Kafka` della libreria Kafkajs. Poiché è fondamentale ne esista una sola istanza viene adottato il *design pattern singleton*. Non ne viene indicata l'implementazione perché questa viene gestita dalla libreria Inversify (sez. 4.2.2).

4.1.1.3.1. Attributi

- `-kafka: Kafka`: istanza della classe `Kafka` della libreria Kafkajs.

4.1.1.3.2. Costruttore

- `+KafkaManager(kafka: Kafka)`: costruttore della classe `KafkaManager` che assegna l'istanza di Kafka all'oggetto proprio.

4.1.1.3.3. Metodi

- `+initAndConnectProducer(): Producer`: crea un *producer* di Kafka, lo connette e lo ritorna in modo che possa essere utilizzato.
- `+disconnectProducer(producer: Producer): void`: disconnette il *producer* passato per parametro.
- `+sendMessage(producer: Producer, topic: string, data: string): void`: fa inviare al *producer* passato per parametro il messaggio `data` nel *topic* specificato.
- `+initAndConnectConsumer(topic: string, groupId: string, eachMessageHandler: EachMessageHandler): Consumer`: crea un *consumer* di Kafka, lo connette al *topic* indicato, lo assegna

al «groupId» passato e lo ritorna in modo che possa essere utilizzato. Assegna inoltre la funzione `eachMessageHandler` come metodo da eseguire ogni volta che il *consumer* riceve un messaggio.
- `+disconnectConsumer(consumer: Consumer): void`: disconnette il *consumer* passato per parametro.

4.1.1.4. TrackFetcher

La classe `TrackFetcher` si occupa di inoltrare una richiesta API al servizio OpenStreetMap per recuperare un tracciato verosmile da far percorrere ai sensori.

4.1.1.4.1. Metodi

- `+fetchTrack(): GeoPoint*`: richiama il metodo `request` che restituisce la risposta della chiamata API, estrae i punti del percorso sotto forma di `GeoPoint` e ritorna una loro lista rappresentante il tracciato.
- `-request(): response`: recupera dalle variabili d'ambiente il centro della mappa e il raggio di generazione. Con l'ausilio della classe `GeoPoint` genera i punti di inizio e fine tracciato, li passa come parametri alla *API request* e ritorna la risposta.

4.1.1.5. GeoPoint

La classe `GeoPoint` rappresenta un punto geospaziale costituito da latitudine e longitudine.

4.1.1.5.1. Attributi

- `-latitude: double`: il valore della latitudine del punto geospaziale.
- `-longitude: double`: il valore della longitudine del punto geospaziale.

4.1.1.5.2. Costruttore

- `+GeoPoint(latitude: double, longitude: double)`: costruttore della classe `GeoPoint` che inizializza la latitudine e longitudine con i valori ricevuti.

4.1.1.5.3. Metodi

- `+radiusKmToGeoPoint(radiusKm: double): GeoPoint`: metodo statico che converte il valore di un raggio in chilometri in un `GeoPoint` che dista `radiusKm` dall'equatore. All'aumentare del raggio si va a perdere precisione ma ai fini dimostrativi del progetto sono sufficienti pochi chilometri.
- `+generateRandomPoint(radiusGeoPoint: GeoPoint): GeoPoint`: genera un `GeoPoint` all'interno del raggio ricevuto e con centro l'oggetto di invocazione del metodo.
- `+getLatitude(): double`: restituisce il valore della variabile `latitude`.
- `+getLongitude(): double`: restituisce il valore della variabile `longitude`.

4.1.2. Componenti di utilità

Sfruttando l'aspetto procedurale del linguaggio TypeScript sono state create delle componenti di supporto. Queste non contengono classi o interfacce quindi diventa più efficace descriverle di seguito piuttosto che in un diagramma delle classi.

- **App**: rappresenta il punto di accesso al servizio e si occupa della creazione di una istanza del simulatore e del suo avvio.

App.ts TypeScript

```
1 const simulator = container.get(Simulator);
2 simulator.startSimulation();
```

- **EnvManager**: espone l'accesso per le variabili d'ambiente. Per utilizzarne una è sufficiente importare il modulo e richiamare `env.VAR_NAME`.

config/EnvManager.ts TypeScript

```
dotenv.config({ path: './src/config/.env' });
export const env = process.env;
```

- **InversifyType**: espone una mappa delle componenti da «iniettare» con il *design pattern dependency injection*. In questo modo si evitano i problemi di *mistyping* dei *serviceId*.

config/InversifyType.ts TypeScript

```
export const TYPES = {
  KafkaManager: Symbol.for('KafkaManager'),
  TrackerList: Symbol.for('TrackerList')
};
```

- **Inversify.config**: definisce il *container* e i *binding* per risolvere le dipendenze con la libreria Inverisfy.

config/InversifyType.ts TypeScript

```
export const container = new Container();

container
  .bind<KafkaManager>(TYPES.KafkaManager)
  .toDynamicValue(() => {
    const kafkaConfig: KafkaConfig = {
      clientId: env.CLIENT_ID,
      brokers: [env.BROKER ?? 'localhost:9094']
    };
    const kafka: Kafka = new Kafka(kafkaConfig);
    return new KafkaManager(kafka);
  })
  .inSingletonScope();

container
  .bind<Tracker[]>(TYPES.TrackerList)
  .toDynamicValue((context: ResolutionContext): Tracker[] => {
    const kafkaManager: KafkaManager =
    context.get<KafkaManager>(TYPES.KafkaManager);
    let trackerList: Tracker[] = [];
    for (let i = 1; i <= Number(env.INIT_TRACKER_COUNT); i++) {
      const id = i.toString();
      const tracker: Tracker = new Tracker(id, kafkaManager);
      trackerList.push(tracker);
    }
    return trackerList;
  });

container.bind(Simulator).toSelf().inSingletonScope();
```

4.2. Design pattern adottati

4.2.1. Dependency injection

Quando un progetto è costituito da un numero considerevole di componenti risulta fondamentale minimizzare le dipendenze. Più si riesce ad evitare debito tecnico e più semplice risulta aggiungere funzionalità perché le parti del sistema non sono fortemente accoppiate. L'obiettivo di questo *design pattern* è quindi quello togliere a un componente la responsabilità della risoluzione delle proprie dipendenze.

4.2.1.1. Implementazione della dependency injection

Il gruppo ha deciso di utilizzare la libreria Inversify per gestire la *dependency injection* nel servizio del simulatore. Possedendo delle annotazioni specifiche lo strumento di **IoC** (*Inversion of Control*) ha agevolato l'implementazione del *design pattern*. È stato infatti sufficiente contrassegnare le dipendenze con delle annotazioni (`@Injectable` e `@Inject`) e definire la risoluzione nel *file* `client/src/config/Inversify.config.ts`.

4.2.1.2. Concetti principali di Inversify ed esempio di utilizzo

Adottando il *design pattern dependency injection* le dipendenze sono dichiarate come parametri nel costruttore annotate da `@Inject('serviceId')` e le relative classi devono essere contrassegnate da `@Injectable()`. In un *file* di configurazione poi deve essere dichiarato il *container* e i *binding* tra i serviceId e le classi «iniettabili».

Tracker.ts TypeScript

```typescript
@Injectable()
class Tracker { }
```

Rent.ts TypeScript

```typescript
class Rent {
  constructor(
    @Inject('Tracker')
    private tracker: Tracker
  ) { }
}
```

Inversify.config.ts TypeScript

```typescript
export const container = new Container();
container.bind<Tracker>('Tracker').to(Tracker);
container.bind(Rent).toSelf();
```

Al momento della creazione dell'oggetto di tipo `Rent` è sufficiente la funzione `get()` del *container* e questa risolverà le dipendenze come specificato.

App.ts TypeScript

```typescript
const rent = container.get(Rent);
```

4.2.1.3. Integrazione del design pattern nel progetto

Nel servizio del simulatore sono state risolte le dipendenze tra i sensori e il *manager* di Apache Kafka, tra il simulatore e la lista di sensori. Nel *file* di configurazione è stato personalizzato il *binding* poiché entrambi richiedono delle precedenti impostazioni che non sono possibili da «iniettare» automaticamente.

Le classi e di conseguenza la *dependency injection* sono state configurate nel seguente modo. Per evitare incongruenze tra i *serviceId* delle classi «iniettabili» è stata crata una mappa univoca.

config/InversifyTypes.ts TypeScript

```
export const TYPES = {
  KafkaManager: Symbol.for('KafkaManager'),
  TrackerList: Symbol.for('TrackerList')
};
```

KafkaManager.ts TypeScript

```
@Injectable()
export class KafkaManager {
  constructor(
      private kafka: Kafka
  ) { }

  ...
}
```

Tracker.ts TypeScript

```
 @Injectable()
 export class Tracker {
   ...

   constructor(
     private id: string,
     @inject(TYPES.KafkaManager)
     private kafkaManager: KafkaManager
   ) { }

   ...
}
```

Simulator.ts TypeScript

```
export class Simulator {
  constructor(
    @inject(TYPES.TrackerList)
    private trackerMap: Tracker[]
  ) { }

  ...
}
```

Poiché i *bind* di `KafkaManager` e `Tracker` non sono immediatamente risolvibili è stato necessario definirli con `toDynamicValue()`. Per inizializzare i sensori è stato assegnato loro un identificativo incrementale coincidente con quelli in *database*.

Il *bind* di `KafkaManager` e `Simulator` è stato contrassegnato dalla funzione `inSingletonScope()` per assicurare che esista una sola istanza per tipo.

config/Inversify.config.ts TypeScript

```typescript
export const container = new Container();

container
  .bind<KafkaManager>(TYPES.KafkaManager)
  .toDynamicValue(() => {
    const kafkaConfig: KafkaConfig = {
      clientId: env.CLIENT_ID,
      brokers: [env.BROKER ?? 'localhost:9094']
    };
    const kafka: Kafka = new Kafka(kafkaConfig);
    return new KafkaManager(kafka);
  })
  .inSingletonScope();

container
  .bind<Tracker[]>(TYPES.TrackerList)
  .toDynamicValue((context: ResolutionContext): Tracker[] => {
    const kafkaManager: KafkaManager =
    context.get<KafkaManager>(TYPES.KafkaManager);
    let trackerList: Tracker[] = [];
    for (let i = 1; i <= Number(env.INIT_TRACKER_COUNT); i++) {
      const id = i.toString();
      const tracker: Tracker = new Tracker(id, kafkaManager);
      trackerList.push(tracker);
    }
    return trackerList;
  });

container.bind(Simulator).toSelf().inSingletonScope();
```

Nel *file* principale è sufficiente quindi richiedere l'istanza di `Simulator` tramite la funzione `get()` del *container*.

App.ts TypeScript

```typescript
const simulator = container.get(Simulator);
simulator.startSimulation();
```

4.2.2. Singleton

Può essere che alcune componenti debbano mantenere un'integrità per tutta l'esecuzione del prodotto, non potendo quindi esistere diverse istanze con diversi valori. Il *design pattern singleton* assicura che ovunque si acceda al componente venga restituita sempre la stessa istanza.

4.2.2.1. Implementazione del design pattern

Il gruppo è consapevole della potenziale fallacità di questo *design pattern* in quanto due processi potrebbero concorrere alla stessa risorsa e, in particolari situazioni, far generare due istanze. Nel caso del simulatore per minimizzare gli errori è stato scelto di demandare il lavoro alla libreria Inversify (spiegata nel dettaglio nelle sez. 4.2.1.1 e sez. 4.2.1.2).

4.2.2.2. Integrazione del design pattern nel progetto

Come anticipato nella sez. 4.2.1.3 è stata dichiarata la risoluzione della dipendenza nel *file* `client/src/config/Inversify.config.ts` e le componenti interessate, quindi `KafkaManager` e `Simulator`, sono state contrassegnate dalla funzione `inSingletonScope()`.

config/Inversify.config.ts TypeScript

```
1  export const container = new Container();
2
3  container
4    .bind<KafkaManager>(TYPES.KafkaManager)
5    .toDynamicValue(() => {
6      const kafkaConfig: KafkaConfig = {
7        clientId: env.CLIENT_ID,
8        brokers: [env.BROKER ?? 'localhost:9094']
9      };
10     const kafka: Kafka = new Kafka(kafkaConfig);
11     return new KafkaManager(kafka);
12   })
13   .inSingletonScope();
14
15 ...
16
17 container.bind(Simulator).toSelf().inSingletonScope();
```

5. Stato dei requisiti funzionali

Nella seguente tabella verranno riportati i requisiti funzionali individuati durante l'Analisi dei Requisiti ed il loro stato.

In particolare per ogni requisito verrà riportato:

- **Codice identificativo**

  Ogni requisito è identificato univocamente da un codice che presenta la seguente struttura:

  R[Importanza][Tipo]-[ID]

  Dove
  - **Importanza**: indica il grado di importanza di ogni requisito, che si distingue in:
    - **O**: requisito obbligatorio.
    - **D**: requisito desiderabile.
    - **F**: requisito facoltativo.
  - **Tipo**: indica la tipologia di requisito, che si distingue in:
    - **F**: requisito funzionale.
    - **Q**: requisito di qualità.
    - **V**: requisito di vincolo.
  - **ID**: numero progressivo che identifica univocamente il requisito nella sua categoria.
- **Descrizione**
- **Stato**
  - Soddisfatto
  - Non soddisfatto

Per una spiegazione più approfondita si rimanda al documento `analisi_dei_requisiti_ver2.0.0`.

5.1. Tracciamento dei requisiti funzionali soddisfatti

Codice	Descrizione	Stato
Obbligatori		
ROF-1	Il sensore deve trasmettere i suoi dati di identificazione e localizzazione al sistema a intervalli regolari.	Soddisfatto
ROF-2	La *dashboard* dell'amministratore, per essere accessibile solamente da quest'ultimo, deve essere protetta da un sistema di autenticazione. Per poter visualizzare la *dashboard* l'amministratore deve quindi autenticarsi con le proprie credenziali.	Soddisfatto
ROF-3	L'amministratore, per poter accedere alla *dashboard*, deve fornire l'indirizzo *e-mail* dell'*account* con cui è registrato all'interno del sistema.	Soddisfatto
ROF-4	L'amministratore, per poter accedere alla *dashboard*, deve fornire la *password* dell'*account* con cui è registrato all'interno del sistema.	Soddisfatto
ROF-5	Se l'amministratore inserisce delle credenziali non valide, come una *e-mail* o *password* non valida, il sistema deve ritornare il messaggio di errore «Credenziali errate».	Soddisfatto

Codice	Descrizione	Stato
ROF-6	L'amministratore, una volta autenticato, deve poter visualizzare la mappa geografica sulla *dashboard* del sistema.	Soddisfatto
ROF-7	L'amministratore deve poter visualizzare i punti di interesse presenti all'interno del sistema tramite dei *marker* posizionati all'interno della mappa geografica.	Soddisfatto
ROF-8	L'amministratore, per ogni noleggio attivo che viene erogato, deve poter visualizzare il tracciato percorso dal mezzo a noleggio attraverso la mappa geografica.	Soddisfatto
ROF-9	L'amministratore deve poter visualizzare un *marker* in corrispondenza di una posizione, in prossimità di un punto di interesse, che ha causato la generazione di un annuncio tramite LLM per l'utente del mezzo.	Soddisfatto
ROF-10	L'amministratore deve poter visualizzare un *marker* in corrispondenza di una posizione, in prossimità di un punto di interesse, dove la LLM non ha generato un annuncio perché ha valutato l'utente come non interessato al punto di interesse in base alla sua profilazione.	Soddisfatto
ROF-11	L'amministratore deve poter visualizzare tramite un'interazione con il *marker* (come un *hover* o un *click*) le informazioni relative al punto di interesse.	Soddisfatto
ROF-12	L'amministratore deve poter visualizzare dalle informazioni fornite tramite l'interazione con il *marker* del punto di interesse il nome dello stesso.	Soddisfatto
ROF-13	L'amministratore deve poter visualizzare, dalle informazioni fornite tramite interazione con il *marker* del punto di interesse, la categoria di esercizio commerciale (e.g. alimentare, sportivo, etc.).	Soddisfatto
ROF-14	L'amministratore deve poter visualizzare le informazioni relative all'annuncio generato tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-15	L'amministratore deve poter visualizzare il nome del punto di interesse legato all'annuncio dalle informazioni visualizzate tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-16	L'amministratore deve poter visualizzare l'*e-mail* dell'utente destinatario dalle informazioni visualizzate tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-17	L'amministratore deve poter visualizzare la data e l'ora di generazione dell'annuncio dalle informazioni visualizzate tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-18	L'amministratore deve poter visualizzare l'annuncio dalle informazioni visualizzate tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-19	L'amministratore deve poter visualizzare la categoria di esercizio commerciale del punto di interesse coinvolto nella generazione	Soddisfatto

Codice	Descrizione	Stato
	dell'annuncio visualizzato tramite un'interazione con un *marker* di generazione annuncio.	
ROF-20	L'amministratore deve poter visualizzare un messaggio con le informazioni di un annuncio non generato, tramite l'interazione con un *marker* di mancata generazione.	Soddisfatto
ROF-21	L'amministratore deve poter visualizzare il nome del punto di interesse sul messaggio con le informazioni di un annuncio non generato, tramite l'interazione con un *marker* di mancata generazione.	Soddisfatto
ROF-22	L'amministratore deve poter visualizzare l'*e-mail* dell'utente destinatario sul messaggio con le informazioni di un annuncio non generato, tramite l'interazione con un *marker* di mancata generazione.	Soddisfatto
ROF-23	L'amministratore deve poter visualizzare la data e ora di tentata generazione sul messaggio con le informazioni di un annuncio non generato, tramite l'interazione con un *marker* di mancata generazione.	Soddisfatto
ROF-24	L'amministratore deve poter chiudere la vista con le informazioni sull'annuncio generato visualizzata sulla mappa tramite l'interazione con un *marker* di generazione annuncio.	Soddisfatto
ROF-25	L'amministratore deve poter chiudere il messaggio di annuncio non generato visualizzato sulla mappa tramite l'interazione con un *marker* di mancata generazione.	Soddisfatto
ROF-26	L'amministratore deve essere in grado di interagire con la mappa per spostare il centro della visuale.	Soddisfatto
ROF-27	L'amministratore deve essere in grado di modificare l'ampiezza della visuale sulla mappa. In particolare bisogna permettere l'ampliamento e il restringimento del campo visivo che l'amministratore ha sul territorio visualizzato all'interno della mappa.	Soddisfatto
ROF-28	Creazione di un generatore di dati GPS per simulare il funzionamento di un sensore che interagisce col sistema.	Soddisfatto
ROF-29	Il generatore deve generare dei percorsi che siano realistici, ovvero che seguano le varie strade, vie e piste ciclabili che una bicicletta può percorrere.	Soddisfatto
Desiderabili		
RDF-1	L'amministratore deve essere in grado di accedere alla sezione dedicata allo storico degli annunci generati all'interno della *dashboard*.	Soddisfatto
RDF-2	L'amministratore deve essere in grado di visualizzare, nella sezione dedicata allo storico degli annunci, l'elenco degli annunci generati dal sistema.	Soddisfatto

Codice	Descrizione	Stato
RDF-3	L'amministratore deve essere in grado di visualizzare l'elenco degli annunci nello storico sotto forma di lista oppure di griglia a seconda delle preferenze dell'amministratore stesso.	Non soddisfatto
RDF-4	L'amministratore deve essere in grado di visualizzare un singolo elemento con le informazioni dell'annuncio all'interno dello storico.	Soddisfatto
RDF-5	L'amministratore deve essere in grado di visualizzare il nome del punto di interesse di ogni singolo elemento presente all'interno dello storico.	Soddisfatto
RDF-6	L'amministratore deve essere in grado di visualizzare l'*e-mail* dell'utente destinatario di ogni singolo elemento all'interno dello storico.	Soddisfatto
RDF-7	L'amministratore deve essere in grado di visualizzare data e ora relativi al tentativo di generazione di ogni singolo elemento all'interno dello storico.	Soddisfatto
RDF-8	L'amministratore deve essere in grado di poter visualizzare i dettagli di un singolo elemento all'interno dello storico.	Soddisfatto
RDF-9	L'amministratore deve essere in grado di visualizzare il nome del punto di interesse relativo ad un annuncio tramite la visualizzazione dei dettagli dell'elemento nello storico.	Soddisfatto
RDF-10	L'amministratore deve essere in grado di visualizzare l'*e-mail* dell'utente destinatario di un annuncio tramite la visualizzazione dei dettagli dell'elemento nello storico.	Soddisfatto
RDF-11	L'amministratore deve essere in grado di visualizzare data e ora del tentativo di generazione di un annuncio tramite la visualizzazione dei dettagli dell'elemento nello storico.	Soddisfatto
RDF-12	L'amministratore deve essere in grado di visualizzare il corpo dell'annuncio tramite la visualizzazione dei dettagli dell'elemento nello storico.	Soddisfatto
RDF-13	L'amministratore deve essere in grado di visualizzare la categoria del punto di interesse collegato all'annuncio tramite la visualizzazione dei dettagli dell'elemento nello storico.	Soddisfatto
RDF-14	L'amministratore deve essere in grado di chiudere la vista di visualizzazione dei dettagli di un singolo annuncio.	Soddisfatto
RDF-15	L'amministratore deve essere in grado, tramite un sistema di filtraggio, di visualizzare gli annunci dello storico per *e-mail* dell'utente destinatario dell'annuncio.	Soddisfatto
RDF-16	L'amministratore deve essere in grado, tramite un sistema di filtraggio, di visualizzare gli annunci dello storico per nome del punto di interesse.	Soddisfatto
RDF-17	L'amministratore deve essere in grado, tramite un sistema di filtraggio, di visualizzare gli annunci dello storico generati in un certo intervallo di date.	Soddisfatto

Codice	Descrizione	Stato
RDF-18	L'amministratore deve essere in grado, tramite un sistema di filtraggio, di visualizzare gli annunci dello storico generati in una determinata fascia oraria.	Soddisfatto
	Facoltativi	
RFF-1	L'amministratore deve poter visualizzare la sezione dedicata ai grafici all'interno della *dashboard* del sistema.	Soddisfatto
RFF-2	L'amministratore deve poter visualizzare un singolo grafico relativo ad una particolare analisi dati.	Soddisfatto
RFF-3	L'amministratore deve poter visualizzare il titolo di uno specifico grafico a seconda dell'analisi dati che viene rappresentata.	Soddisfatto
RFF-4	L'amministratore deve poter visualizzare in uno specifico grafico un'etichetta relativa alla tipologia di misura rappresentata sulle assi delle ascisse e delle ordinate e, infine, i relativi valori.	Soddisfatto
RFF-5	L'amministratore deve poter visualizzare, all'interno di ciascun grafico, la rappresentazione dello specifico *set* di dati previsti per quel grafico.	Soddisfatto
RFF-6	L'amministratore deve poter visualizzare un grafico che mostri il numero di annunci generati dal sistema nelle ultime 24 ore, con granularità oraria.	Soddisfatto
RFF-7	L'amministratore deve poter visualizzare un grafico raffigurante il numero medio di noleggi che vengono effettuati in ciascun mese dell'anno, risultato della media di noleggi effettuati in quel mese nel corso degli anni.	Soddisfatto
RFF-8	L'amministratore, dalla sezione dedicata ai grafici, deve poter selezionare uno specifico punto di interesse per poter poi visualizzare i grafici delle statistiche ad esso correlate.	Soddisfatto
RFF-9	L'amministratore deve poter visualizzare un grafico che mette a confronto il numero di annunci generati con il numero di annunci non generati per un certo punto di interesse nell'ultima settimana.	Soddisfatto
RFF-10	Viene richiesta la creazione di uno strumento di visualizzazione degli annunci in tempo reale per l'utente utilizzatore del servizio.	Non soddisfatto

6. Grafici riassuntivi

6.1. Requisiti funzionali obbligatori

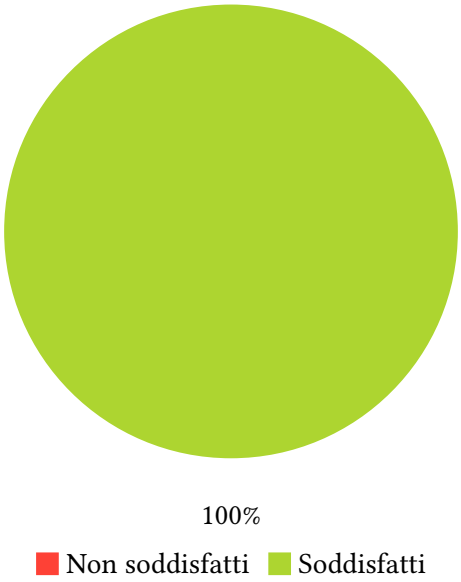

Figura 15: Grafico a torta riassunto dei requisiti funzionali obbligatori

6.2. Requisiti funzionali desiderabili

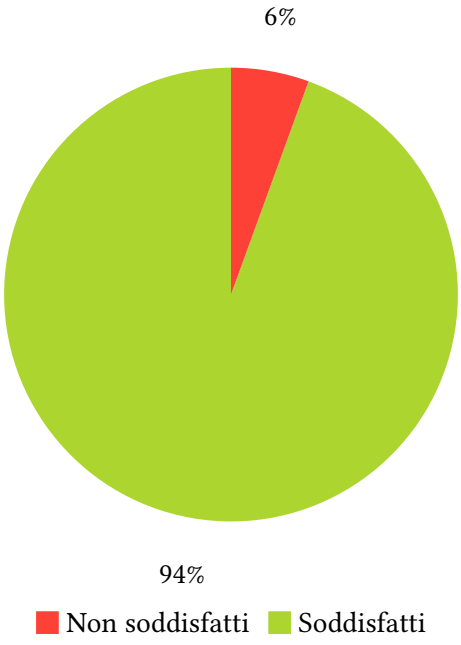

Figura 16: Grafico a torta riassunto dei requisiti funzionali desiderabili

6.3. Requisiti funzionali facoltativi

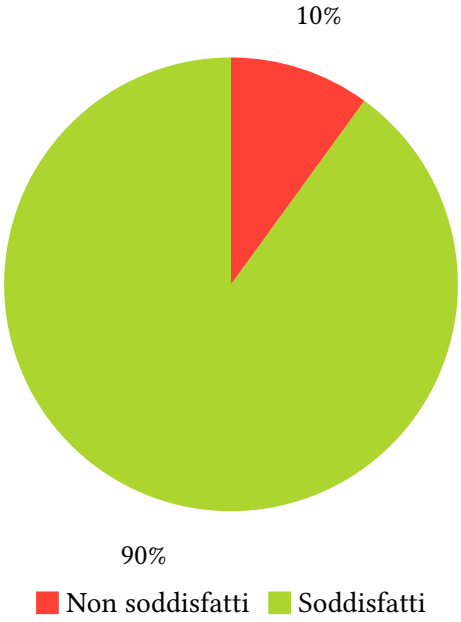

Figura 17: Grafico a torta riassunto dei requisiti funzionali facoltativi

6.4. Requisiti funzionali totali

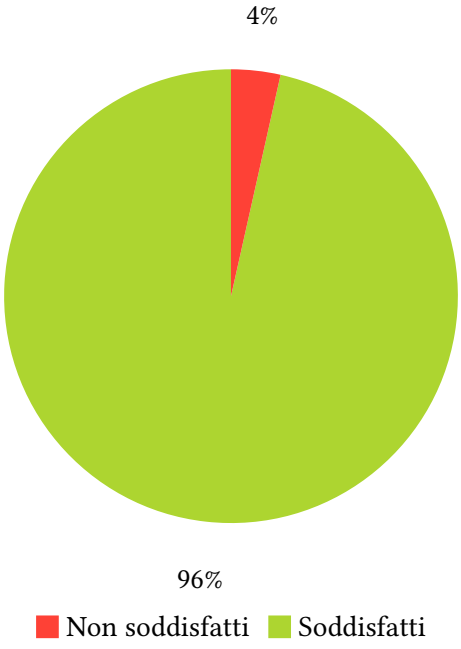

Figura 18: Grafico a torta riassunto dei requisiti funzionali totali