MANUALE DEL MANUTENTORE

ThiReMa Project

www.redroundrobin.site — redroundrobin.site@gmail.com

Informazioni sul documento

Versione	2.0.0+b1.0
Uso	Esterno
Stato	Approvato
Destinatari	SanMarco Informatica
	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Red Round Robin
Redattori	Nicolò Frison
	Mariano Sciacco
Verificatori	Giovanni Vidotto
	Fouad Mouad
Approvazione	Lorenzo Dei Negri

Registro delle modifiche

Versione	Descrizione	Data	Autore	Ruolo
2.0.0+b1.0	Approvazione del documento	2020-05-15	Lorenzo Dei Negri	Responsabile
1.0.5+b0.18	Aggiunta e verifica della nuova appendice §A	2020-05-14	Mariano Sciacco e Fouad Mouad	Programmatore e verificatore
1.0.4+b0.18	Aggiornamento e verifica della sezione §5	2020-05-13	Mariano Sciacco e Fouad Mouad	Programmatore e verificatore
1.0.3+b0.18	Aggiornamento e verifica della sezione §3	2020-05-12	Nicolò Frison e Giovanni Vidotto	Programmatore e verificatore
1.0.2+b0.17	Aggiornamento e verifica della sezione §2	2020-05-08	Nicolò Frison e Giovanni Vidotto	Programmatore e verificatore
1.0.1+b0.17	Aggiornamento e verifica della sezione §1	2020-05-07	Nicolò Frison e Giovanni Vidotto	Programmatore e verificatore
1.0.0+b0.15	Approvazione del documento	2020-04-13	Alessandro Tommasin	Responsabile
0.2.2+b0.14	Stesura e verifica dell'appendice §A	2020-04-11	Lorenzo Dei Negri e Mariano Sciacco	Programmatore e verificatore
0.2.1+b0.14	Stesura e verifica della sezione §4	2020-04-10	Lorenzo Dei Negri e Mariano Sciacco	Programmatore e verificatore
0.2.0+b0.14	Approvazione del documento	2020-04-06	Giovanni Vidotto	Responsabile
0.1.2+b0.13	Stesura e verifica della sezione §3	2020-04-03	Nicolò Frison e Lorenzo Dei Negri	Programmatore e verificatore
0.1.1+b0.13	Stesura e verifica della sezione §2	2020-04-02	Nicolò Frison e Lorenzo Dei Negri	Programmatore e verificatore
0.1.0+b0.12	Approvazione del documento	2020-03-23	Lorenzo Dei Negri	Responsabile
0.0.2+b0.11	Stesura e verifica della sezione §1	2020-03-22	Alessandro Tommasin e Fouad Mouad	Programmatore e verificatore
0.0.1+b0.11	Creazione documento	2020-03-21	Alessandro Tommasin	Programmatore

Indice

Elenco delle figure

1 Introduzione

1.1 Premessa

Il nome *ThiReMa project* è il nome in codice interno usato dai fornitori per identificare il prodotto *RIoT*. Qualunque riferimento a *ThiReMa project* è un riferimento diretto a *RIoT*.

1.2 Scopo del documento

Questo documento si propone come guida per gli sviluppatori che andranno ad estendere e/o manutenere il prodotto RIoT.
Nelle sezioni successive del documento sarà possibile trovare una descrizione dei linguaggi, delle tecnologie e degli strumenti utilizzati durante lo sviluppo del progetto, oltre che un'analisi dell'architettura e delle scelte progettuali fatte per il prodotto.

1.3 Scopo del prodotto

Per grandi e medie aziende, ma non solo, la gestione e l'analisi di grosse moli di dati sta diventando sempre di più una realtà concreta.
Il progetto RIoT si prefigge come obiettivo la creazione di una WEB APPLICATION$_G$, la quale permetta di analizzare ingenti moli di dati, ricevuti da più sensori eterogenei tra loro. Tale applicazione metterà a disposizione un'interfaccia intuitiva che permetterà di visualizzare più dati di interesse od eventuali correlazioni tra gli stessi. Infine, per ogni tipologia di dato sarà possibile assegnarne il monitoraggio ad un particolare ENTE$_G$.

1.4 Glossario

Per evitare possibili ambiguità relative ad alcuni termini usati nel documento, questi verranno indicati in MAIUSCOLETTO con una G al pedice e saranno riportati nel glossario presente nell'appendice §A.

2 Tecnologie interessate

Di seguito vengono descritte le diverse tecnologie che, dopo una prima fase di analisi, sono state scelte per lo sviluppo del prodotto.

2.1 Linguaggi

In questa sezione vengono trattati i linguaggi utilizzati per l'implementazione di RIoT.

2.1.1 Java

Il linguaggio è stato scelto perché permette di interfacciarsi con facilità con la piattaforma KAFKA$_G$ richiesta dal capitolato. JAVA$_G$ permette inoltre di gestire le dipendenze in maniera automatica tramite Maven e di scrivere ed eseguire suite di test in maniera facile e veloce tramite JUNIT$_G$.
Il linguaggio è inoltre ben conosciuto e facile da apprendere per uno sviluppatore con una qualche esperienza nella programmazione orientata agli oggetti.

2.1.2 PHP

Questo linguaggio è stato scelto per lo sviluppo dell'applicazione web perché utilizzabile assieme a framework che permettono la realizzazione di componenti e pagine web in maniera veloce ed intuitiva, utilizzando sempre una sintassi elegante.

2.1.3 JavaScript

JAVASCRIPT$_G$ è stato scelto perché permette la realizzazione della componente BOT TELEGRAM$_G$ con un numero limitato di righe di codice, utilizzando dei moduli presenti per il suddetto linguaggio.
Un altro punto a suo favore è che il suo tempo di apprendimento è ragionevolmente breve.

2.2 Strumenti

In questa sezione vengono trattati gli strumenti utilizzati per l'implementazione di RIoT.

2.2.1 Docker

Questo strumento è stato utilizzato sia perché fortemente consigliato dal capitolato, sia perché permette il rilascio delle varie componenti in ambienti isolati tra loro, detti CONTAINER$_G$, che simulano degli ambienti virtuali dov'è possibile eseguire e testare le proprie applicazioni.
In questo modo è possibile simulare l'esecuzione di più sistemi operativi su una stessa macchina fisica, condividendone le risorse, in modo da ridurre i costi e aumentare l'efficienza generale.
A differenza delle macchine virtuali, i CONTAINER$_G$ risultano essere più leggeri, in quanto vengono fatti su misura per le applicazioni che devono contenere; in questo modo occupano meno memoria sul

disco e impiegano meno risorse hardware.
I container vengono configurati tramite dei DOCKERFILE$_G$, dove vengono specificate le operazioni da eseguire all'avvio del container stesso, oltre che informazioni e parametri specifici dell'ambiente, come il sistema operativo da utilizzare. Infine tramite dei DOCKER-COMPOSE$_G$ vengono assemblate le diverse componenti, permettendo in pochi comandi di costruire l'intera architettura.

2.3 Framework e librerie

Le librerie esterne utilizzate sono state gestite tramite Maven per le componenti sviluppate in JAVA$_G$ (gateway, data connector ed API$_G$); per i moduli del bot Telegram (sviluppato in JAVASCRIPT$_G$) è richiesto invece il gestore di pacchetti di NODE.JS$_G$ NPM$_G$. Infine per la componente web è necessario, oltre al già menzionato *npm*, anche il gestore di pacchetti PHP *Composer*.

2.3.1 Spring

Il framework SPRING$_G$ è stato utilizzato per lo sviluppo della componente API$_G$. Più precisamente dell'intera suite sono stati utilizzati Spring Boot, Spring Security, Spring Kafka e Spring Data JPA.

- **Spring Boot** viene utilizzato per la creazione della componente API in quanto permette di creare un microservizio che è quindi indipendente dalle altre componenti del sistema; è inoltre fornito di ottima documentazione per creare un servizio web RESTful;
- **Spring Security** viene utilizzato per gestire la sicurezza e le richieste fatte alle API dalle altre componenti del sistema; è stato scelto perché è lo standard de-facto per gestire la sicurezza di applicazioni basate su Spring. Questo framework permette inoltre un'alta personalizzazione dei servizi di autenticazione da fornire, nel nostro caso nella componente API. La documentazione è disponibile al seguente indirizzo, dove è possibile trovare anche alcune guide molto utili;
- **Spring Kafka** viene utilizzato per gestire ad alto livello la comunicazione con uno o più broker kafka, permettendo quindi di inserire o reperire dei dati da uno o più topic. Nello specifico è stato impiegato per configurare sia per la connessione con il nostro servizio Kafka, che per inviare nuove configurazioni ai gateway, utilizzando un KafkaTemplate che permette di inviare messaggi ad un topic con una sintassi ad alto livello. Una guida utile ad un'eventuale estensione di questo servizio è disponibile al seguente puntatore;
- **Spring Data JPA** viene utilizzato all'interno del pacchetto repositories della componente API per gestire l'accesso al layer contenente i dati. Più nello specifico permette l'implementazione dell'accesso ai dati in maniera semplice, senza richiedere la scrittura di una grande quantità codice boilerplate. Per un'eventuale estensione del servizio o per eventuali chiarimenti si consiglia la seguente guida.

2.3.2 Jwt

Questa libreria permette di utilizzare lo standard JWT per trasmettere informazioni tra due componenti in formato JSON$_G$. Viene utilizzata dalle componenti API per creare e gestire dei token personalizzati che permettono l'accesso (sia normale che a 2FA) e lo scambio di dati alla web app o al bot Telegram.

2.3.3 Laravel

Questo framework permette di realizzare applicazioni web utilizzando una sintassi espressiva ed elegante. Nel progetto è stato utilizzato per sviluppare la componente WEB APP$_G$. Laravel è stato scelto anche per l'ottima documentazione e le numerose guide che sono disponibili nel sito ufficiale.

2.3.4 Vue.js

Questo framework permette di scrivere pagine web reattive e dalla sintassi elegante. Nel progetto è stato utilizzato per sviluppare la componente WEB APP$_G$. Più nello specifico Vue permette di suddividere in componenti una pagina HTML che può quindi variare la sua struttura in maniera dinamica in base al comportamento dell'utente. Un'altra funzionalità fornita da Vue è la possibilità di rendere la pagina reattiva collegando il DOM con i dati che dovranno essere renderizzati dal browser. Come i framework precedenti anche Vue è dotato di una ricca documentazione e di numerose guide che ne facilitano l'implementazione.

2.3.5 Boostrap

Questo framework permette di sviluppare siti web e WEB APPLICATION$_G$ responsive. Nel progetto è stato utilizzato nella creazione delle view della nostra web app. Più nel dettaglio è stata sfruttata la possibilità fornita da Bootstrap di creare tabelle e cards responsive che hanno permesso la gestione del contenuto delle pagine HTML in maniera ottimale.

2.3.6 Telegraf

Questo modulo JavaScript viene usato all'interno della componente BOT TELEGRAM$_G$ per creare ed interfacciarsi con le API ufficiali di Telegram. Più nello specifico questo modulo è stato impiegato per creare la variabile "bot", grazie alla quale è stato possibile implementare i comandi necessari al corretto funzionamento della componente bot Telegram. Telegraf mette inoltre a disposizione vari tipi e sottotipi di dato (quali ad esempio message, callback_query, text o audio) che permettono una buona gestione delle risposte sulla base degli input dell'utente. Per questo modulo sono disponibili alcune guide direttamente sul sito ufficiale, dove è presente anche la documentazione delle API sia di Telegraf che di Telegram.

2.4 Requisiti

2.4.1 Requisiti hardware minimi (sviluppo)

I requisiti hardware minimi per lo sviluppo identificano le risorse minime per usare un server di test o per poter sviluppare il prodotto nel proprio computer. Questi requisiti non sono adatti a un server di produzione e vanno considerati come minimo indispensabile per poter provare il prodotto in locale.

- **sistema operativo:**
  - Ubuntu 18.04.3 LTS a 64bit;
  - Windows 10 PRO (Hyper-V enabled, Linux containers enabled) Build 1903 a 64bit;
  - Windows Server 2019 (Hyper-V enabled, Linux containers enabled);
  - macOS 10.15 Catalina;
- **processore:** CPU Dual-Core da 2.6GHz o superiore;
- **RAM:** 6GB DDR3 o superiore;
- **disco rigido:** 32GB di spazio o superiore;
- **connessione internet:** attiva (20 Mb/s).

2.4.2 Requisiti hardware minimi (produzione)

I requisiti hardware minimi per la produzione identificano le risorse minime necessarie per mettere in funzione l'applicazione e abilitare il servizio al lavoro effettivo. Si sconsiglia di utilizzare requisiti minori di quelli riportati o si rischia di avere instabilità del sistema e possibili malfunzionamenti a carico elevato.

- **sistema operativo:**
  - Ubuntu server 18.04.3 LTS a 64bit;
  - Windows 10 PRO (Hyper-V enabled, Linux containers enabled) Build 1903 a 64bit;
  - Windows Server 2019 (Hyper-V enabled, Linux containers enabled);
  - macOS 10.15 Catalina;
- **processore:** CPU Quad-Core da 2.8GHz o superiore;
- **RAM:** 8GB DDR4 o superiore;
- **disco rigido:** 64GB di spazio o superiore;
- **connessione internet:** attiva (100 Mb/s).

2.4.3 Requisiti di sistema

- **Docker** nella versione 19.x.x;
- **docker-compose** nella versione 1.25.x;

2.4.4 Requisiti client web

- FIREFOX$_G$ a partire dalla versione 69.0;
- CHROME$_G$ a partire dalla versione 75.0;
- SAFARI$_G$ a partire dalla versione 13.0;
- EDGE$_G$ a partire dalla versione 42.0.

2.4.5 Linguaggi di programmazione

- **Java** nella versione 11;
- **PHP** nella versione 7.3+;
- **JavaScript**.

2.4.6 Gestori delle dipendenze

- **npm** nella versione 6.13.x+;
- **Composer** nella versione 1.1.x;
- **Maven** nella versione 3.6.x.

3 Installazione

In questa sezione viene spiegato come effettuare l'installazione delle componenti del progetto RIoT.

3.1 Installazione e gestione tramite script BASH

L'installazione e la gestione del prodotto è stata semplificata attraverso uno script BASH denominato *mr_wolf.sh*. Questo script si basa su Docker e permette di gestire alla stessa maniera di un docker-compose tutti i servizi attivi. In particolare, è possibile controllare le seguenti operazioni:

- **help:** comandi di aiuto per lo script *mr_wolf*;
- **status:** status dei servizi RIoT;
- **init:** prima installazione e avvio di tutti i servizi RIoT;
- **start:** avvio di tutti i servizi RIoT;
- **stop:** stop di tutti i servizi RIoT;
- **remove:** stop e rimozione di tutti i servizi RIoT;

L'avvio dello script può essere effettuato nei sistemi operativi Linux e MacOs nel seguente modo:

```
$ chmod +x mr_wolf.sh
$ ./mr_wolf.sh
```

Notare che il primo comando va effettuato solo la prima volta.
Infine, l'installazione e gestione su Windows è disponibile solamente tramite Docker.

3.2 Installazione e avvio completo tramite Docker

In alternativa, è possibile eseguire l'installazione completa di tutto il prodotto sulla propria macchina tramite Docker. Per farlo, è sufficiente eseguire il file *docker-compose.riot.yml* nel proprio terminale tramite il comando:

```
$ docker-compose -f docker-compose.riot.yml up -d
```

3.3 Gateway

3.3.1 Avvio tramite Docker

Il gateway è stato predisposto con un *Dockerfile* con cui avviare direttamente tramite un Docker container l'applicazione. Per fare ciò è sufficiente eseguire i seguenti comandi:

```
$ docker build --tag thirema-gateway:1.0 .
$ docker run --publish 29092:29092 --detach --name thirema-gateway thirema-gateway:1.0
```

Verrà creata una immagine contenente il software avviato.

3.3.2 Avvio manuale

Per installare localmente la componente gateway, è necessario eseguire nel terminale il maven build lifecycle all'interno della cartella della componente tramite il comando:

```
$ mvn install
```

ed in seguito avviare il client.

3.4 Data collector

3.4.1 Avvio tramite Docker

Il data collector è stato predisposto con un *Dockerfile* con cui avviare direttamente tramite un Docker container l'applicazione. Per fare ciò è sufficiente eseguire i seguenti comandi:

```
$ docker build --tag thirema-dc:1.0 .
$ docker run --publish 29092:29092 --detach --name thirema-data-collector thirema-dc:1.0
```

Verrà creata una immagine contenente il software avviato.

3.4.2 Avvio manuale

Per installare localmente la componente data collector è necessario eseguire nel terminale il maven build lifecycle all'interno della cartella della componente tramite il comando:

```
$ mvn install
```

ed in seguito avviare il client.

3.5 API

3.5.1 Avvio tramite Docker

Le API sono state predisposte con un *Dockerfile* con cui avviare direttamente tramite un Docker container l'applicazione. Per fare ciò è sufficiente eseguire i seguenti comandi:

```
$ docker build --tag thirema-api:1.0 .
$ docker run -p 29092:29092 -p 9999:9999 -p 3000:3000 --detach --name thirema-api thirema-api:1.
```

Verrà creata una immagine contenente il software avviato.

3.5.2 Avvio manuale

Per installare localmente la componente API è necessario eseguire nel terminale il maven build lifecycle all'interno della cartella della componente tramite il comando:

```
mvn install
```

ed in seguito avviare l'eseguibile ApirestApplication.java.

3.6 Web application

3.6.1 Avvio tramite Docker

La WEB APP$_G$ è stata predisposta con un *Dockerfile* con cui avviare direttamente tramite un Docker container l'applicazione. Per fare ciò è sufficiente eseguire i seguenti comandi:

```
$ docker build --tag thirema-webapp:1.0 .
$ docker run -p 80:8000 --detach --name thirema-webapp thirema-webapp:1.0
```

Verrà creata una immagine contenente il software avviato.

3.6.2 Avvio manuale

Per installare localmente la componente WEB APPLICATION$_G$ è necessario eseguire nel terminale i seguenti comandi, a partire dalla cartella della componente:

```
$ composer install
$ npm install
$ npm run dev
$ php artisan serve
```

ed in seguito collegarsi all'indirizzo specificato nel terminale.

3.7 Bot Telegram

3.7.1 Avvio tramite Docker

Il BOT TELEGRAM$_G$ è stata predisposta con un *Dockerfile* con cui avviare direttamente tramite un Docker container l'applicazione. Per fare ciò è sufficiente eseguire i comandi riportati di seguito.

Verrà creata una immagine contenente il software avviato.

```
$ docker build --tag thirema-telegram-bot:1.0 .
$ docker run -p 3000:3000 -d --name thirema-telegram-bot thirema-telegram-bot:1.0
```

3.7.2 Avvio manuale

Per installare ed eseguire localmente il bot Telegram è sufficiente eseguire i seguenti comandi mentre ci si trova all'interno della cartella della componente:

```
$ npm install
$ node main
```

4 Test

In questa sezione viene spiegato come eseguire i test implementati per le componenti del progetto RIoT. Per ciascun componente, in base al linguaggio di programmazione con cui è stato sviluppato, sarà necessario eseguire un comando apposito che avvierà tutte le suite dei test integrati. A partire dalla cartella con tutte le componenti, sarà opportuno spostarsi sulla *working directory* in cui sono contenuti i sorgenti.

4.1 Gateway

- *ambiente:* Java (maven)
- *working directory:* `./gateway/gateway/`
- *comando:* `$ mvn verify`

4.2 Data collector

- *ambiente:* Java (maven)
- *working directory:* `./kafka-db/kafka-data-collector/`
- *comando:* `$ mvn verify`

4.3 API

- *ambiente:* Java (maven)
- *working directory:* `./api/apirest/`
- *comando:* `$ mvn verify`

4.4 Web application

- *ambiente:* PHP (composer), NodeJs (npm)
- *working directory:* `./webapp/`
- *comando per PHP:* `$ composer install && vendor/bin/phpunit -c phpunit.xml`
- *comando per NodeJs:* `$ npm run build --if-present && npm test`

4.5 Bot Telegram

- *ambiente:* NodeJs (npm)
- *working directory:* `./telegram-bot/`
- *comando:* `$ npm run build --if-present && npm test`

5 Architettura

L'architettura del prodotto è suddivisa in:

- **gateway**;
- **piattaforma Apache Kafka**;
- **data collector**;
- **database PostgreSQL e Timeseries**;
- **API REST**;
- **bot Telegram**;
- **web application**.

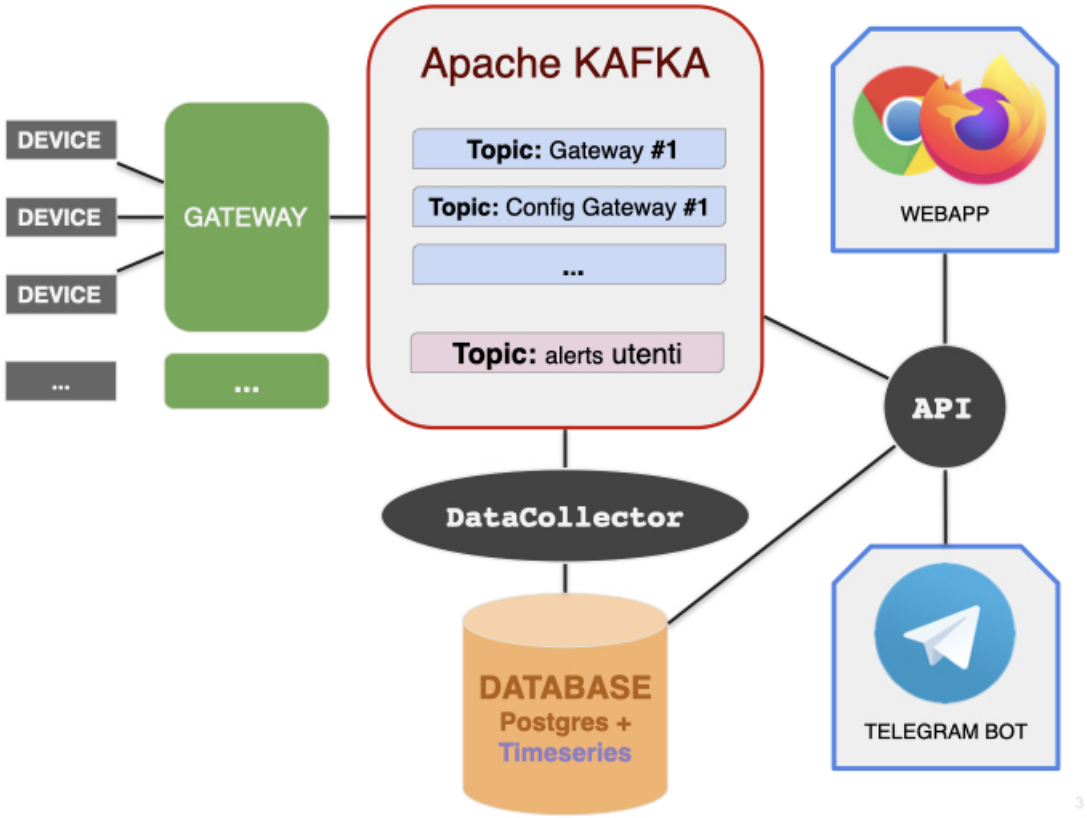

Figura 1: Schema riassuntivo dell'architettura generale del prodotto

5.1 Interazione tra i componenti

Il gateway interagisce con Kafka scrivendo, su un apposito topic di quest'ultimo, i dati che riceve dai sensori ad esso connessi, tramite un producer, in formato JSON$_G$.
Esso è connesso anche a un secondo topic, tramite un consumer, in modo da permettere agli utenti della web app, che ne abbiano i permessi, di impostarne una diversa configurazione.

La componente data collector si interpone tra i client (web app e bot Telegram), Kafka e i database, con lo scopo di tradurre i dati $JSON_G$ salvati nel topic, a cui è connesso tramite un producer, in record per le tabelle del Timeseries db.
Le API_G, che contengono la business logic, si interpongono tra i client, ovvero bot Telegram e web app, Kafka e ai database.
La connessione API-client si attua tramite richieste e risposte HTTP e il passaggio dei dati tra queste componenti avviene tramite oggetti in formato JSON; la comunicazione API-Kafka, invece, avviene tramite un producer collegato ad un topic, per permettere agli utenti che ne abbiano i permessi, di inviare nuove configurazioni ai gateway.
Infine la connessione tra API e Database avviene tramite il framework JPA, fornito da Spring.

5.2 Gateway

Un *gateway* è una componente localizzata all'interno di un'azienda che permette di rendere uniforme l'interfaccia di accesso ai dati dei singoli dispositivi configurati all'interno del gateway stesso.
Dai gateway è possibile inoltre configurare funzioni di accumulo dei pacchetti contenenti i dati dei sensori o di impostare alcuni timer, al termine dei quali deve essere effettuato l'invio dei dati all'interno dei rispettivi topic di KAFKA$_G$.
Tutte le configurazioni vengono ricevute tramite appositi topic adibiti esclusivamente a questa funzione.

- La componente è stata sviluppata in Java 11.

5.2.1 Diagramma dei package

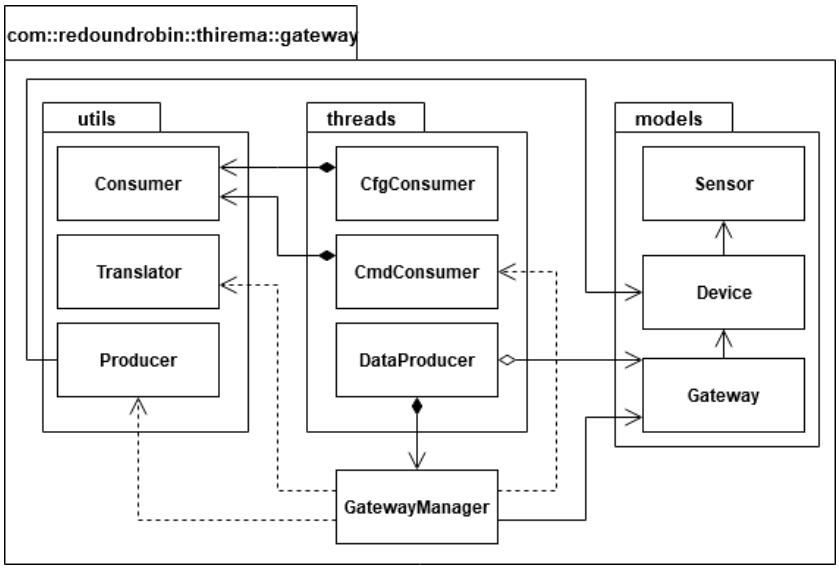

Figura 2: Diagramma dei package per la componente gateway

5.2.2 Dipendenze esterne

La componente gateway ha due dipendenze esterne:

- **KafkaProducer<K, V>**, classe concreta che implementa l'interfaccia Producer<K,V>. Svolge il compito di client per il cluster KAFKA$_G$, pubblicando dati all'interno di un topic. La classe Producer ne possiede un riferimento.
- **KafkaConsumer<K,V>**, classe concreta che implementa l'interfaccia Consumer<K,V>. Svolge il compito di client per il cluster KAFKA$_G$, consumando i messaggi resenti all'interno di uno o più topic. La classe Consumer ne possiede un riferimento.

5.2.3 Diagramma delle classi

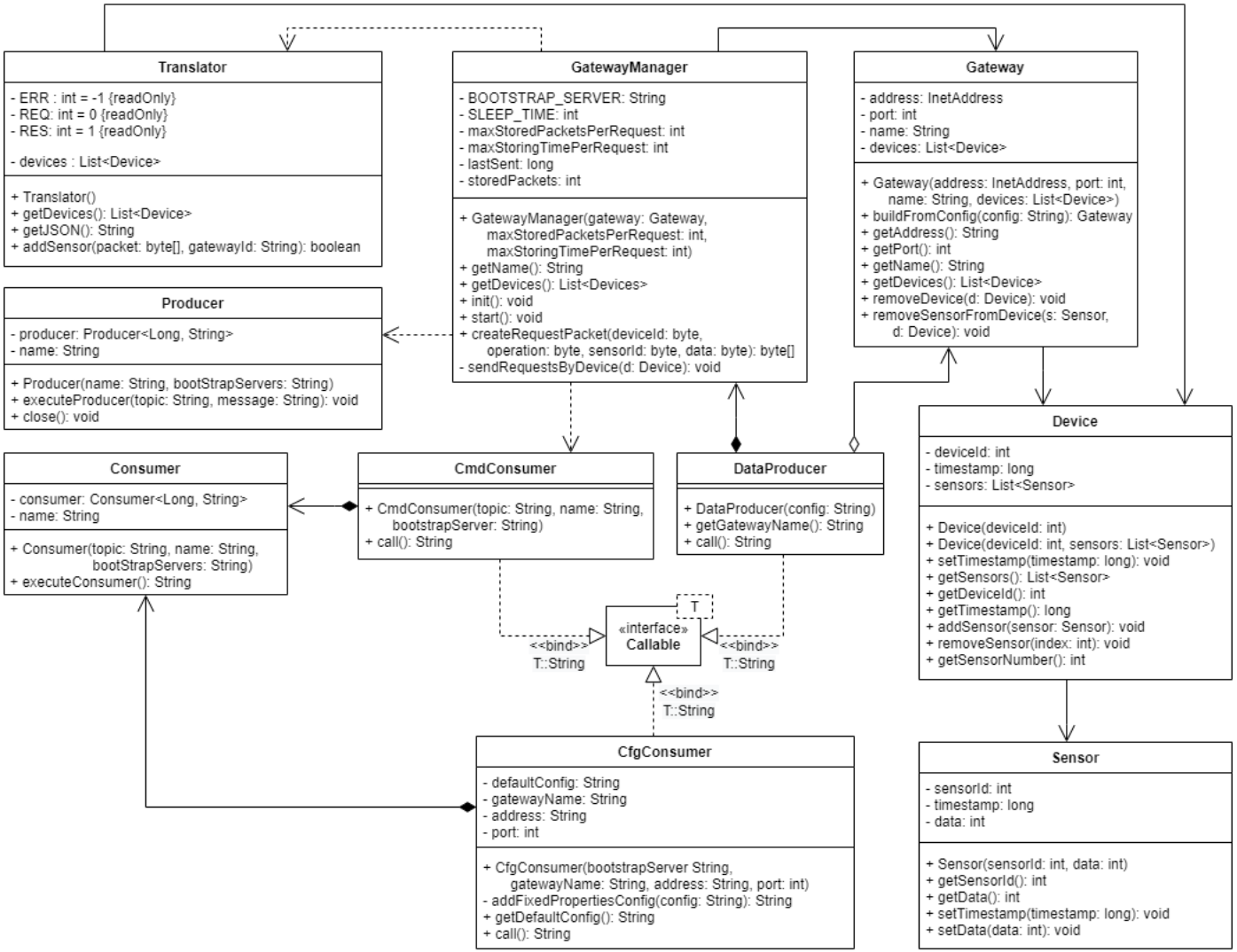

Figura 3: Diagramma delle classi per la componente gateway

Come si evince dal diagramma sono presenti le seguenti classi:

- **Consumer**: rappresenta un consumer di Kafka, che è infatti contenuto al suo interno. La classe viene utilizzata per "consumare" messaggi da uno o più topic di Kafka (in questo caso dal topic delle configurazioni);
- **Producer**: rappresenta un producer di Kafka, che è infatti contenuto al suo interno. La classe viene utilizzata per "produrre" messaggi contenenti le rilevazioni del valori dei sensori all'interno di un topic Kafka;
- **CfgConsumer**: classe che contiene al suo interno un Consumer per poter reperire da un topic KAFKA_G le configurazioni per il gateway. Appena ne riceve una, viene sostituita alla vecchia configurazione e viene ripresa l'attività di fetching dei dati;
- **DataProducer**: classe che contiene un riferimento a Gateway, in cui si trova la business logic per la richiesta al simulatore e l'invio dei dati a Kafka. Per poter fare ciò necessita della classe Translator per convertire i dati in JSON e delle classi Device e Sensor per costruire la lista di dispositivi a cui dovrà richiedere i dati;
- **CmdConsumer**: classe che contiene al suo interno un Consumer per poter reperire da un topic KAFKA_G i comandi per i dispositivi del gateway. Appena ne riceve uno, viene inviato al GatewayManager che lo reindirizza al dispositivo corretto;
- **GatewayManager**: classe che contiene buona parte della logica della componente Gateway: al suo interno infatti è presente un ciclo che itera tutti i dispositivi di tutti i sensori richiedendo ad ognuno dei dati. Quando ne raccoglie un certo numero o passano n secondi (in base alla configurazione del gateway) questi dati vengono inviati ad un topic ed il ciclo si ripete;
- **Translator**:classe che, come detto brevemente in precedenza, viene utilizzata per accumulare e tradurre ciò che viene restituito dai sensori in un pacchetto in formato JSON che poi verrà inviato in un topic;
- **Gateway**: rappresenta il gateway; è infatti presente un identificativo (name), una lista di device (devices) e l'indirizzo (address) e la porta (port) a cui inviare le richieste dei dati dei dispositivi;
- **Sensor**: rappresenta il dato di un singolo sensore; è infatti presente un identificativo (sensorId), un valore (data) ed infine il timestamp legato a quando è stata fatta la rilevazione del valore;
- **Device**: rappresenta un dispositivo; al suo interno è presente una lista di Sensor (sensors), un identificativo (deviceId) ed un timestamp legato a quando è stata fatta l'ultima rilevazione di uno qualsiasi dei suoi sensori.

5.2.4 Diagramma di sequenza

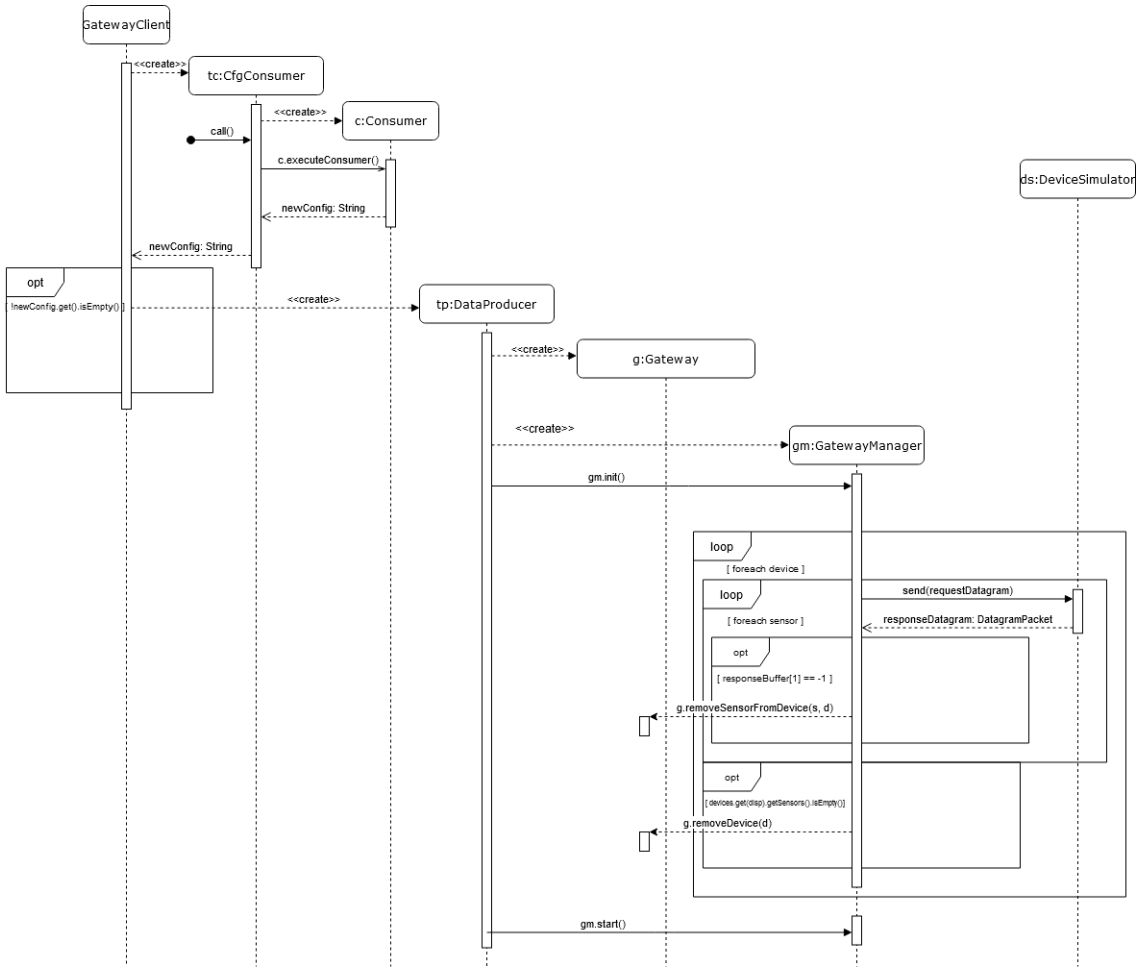

Figura 4: Diagramma di sequenza che rappresenta la richiesta di una prima configurazione ed un primo settaggio del gateway

Nel diagramma di sequenza è rappresentata la richiesta e la ricezione di una prima configurazione da parte del CfgConsumer.
La configurazione viene quindi passata al DataProducer, che tramite un metodo crea il Gateway e il GatewayManager (sulla base della configurazione ottenuta) e questo, tramite il metodo init() verifica che tutti i dispositivi nella configurazione siano effettivamente presenti.

5.2.5 Diagramma di attività

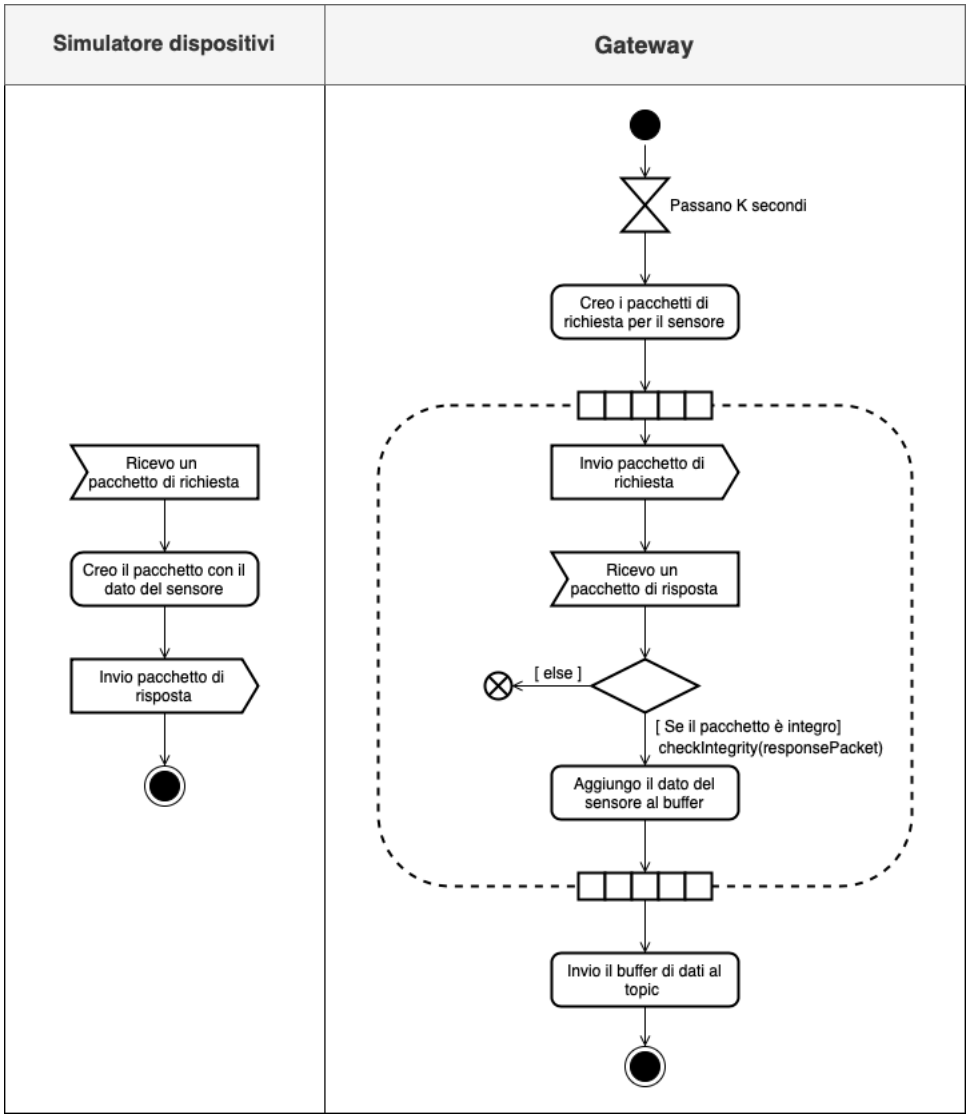

Figura 5: Diagramma di attività che rappresenta un'iterazione all'interno del metodo start() della classe gateway

Nel diagramma è rappresentato il metodo start() della classe Gateway, in cui vengono creati dei pacchetti di richiesta e vengono inviati al simulatore dei dispositivi.
Quest'ultimo, dopo aver generato i dati, invia un pacchetto di risposta che, se è integro, viene aggiunto ad un buffer che, una volta pieno, viene inviato ad un topic KAFKA$_G$.

5.3 Kafka

All'interno del progetto RIoT viene fatto uso della piattaforma APACHE KAFKA$_G$ per trasportare e trasformare i dati provenienti dai dispositivi, verso le altre componenti del sistema.
Per questo progetto è stato utilizzato un broker singolo, ma nulla vieta, in un futuro, di estendere l'architettura creando un cluster composto da più broker.

I principali topic che sono stati creati e che vengono utilizzati dalle vari componenti sono:

- un topic per ogni gateway, all'interno del quale vengono riversati i dati raccolti dai dispositivi associati;
- un topic per ogni gateway, nel quale vengono inviate le configurazioni per i gateway stessi;
- un topic in cui vengono inseriti i messaggi di ALERT$_G$ quando uno o più sensori superano delle soglie stabilite.

Per effettuare il rilascio di questa componente è stato realizzato un apposito file DOCKER-COMPOSE$_G$ (al cui interno sono presenti anche le altre componenti) che imposta automaticamente gli indirizzi e le porte in cui ascoltare e ricevere i messaggi. Quindi per modificare le configurazioni del broker KAFKA$_G$ o per aggiungerne altri è sufficiente modificare il file docker-compose.yml fornito.
Di seguito viene riportato un piccolo estratto del file di configurazione.

```yaml
zookeeper:
  container_name: kafka_zookeeper
  image: 'bitnami/zookeeper:3'
  networks:
    - kafka-net
  ports:
    - '2181:2181'
  volumes:
    - 'zookeeper_data:/bitnami'
  environment:
    - ALLOW_ANONYMOUS_LOGIN=yes

kafka:
  container_name: kafka_core
  image: 'bitnami/kafka:2'
  networks:
    - kafka-net
  ports:
    - '9092:9092'
    - '29092:29092'
  volumes:
    - 'kafka_data:/bitnami'
  environment:
    - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
    - ALLOW_PLAINTEXT_LISTENER=yes
    - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
    - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,PLAINTEXT_HOST://:29092
    - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
  depends_on:
    - zookeeper
```

Figura 6: Estratto del file docker-compose.yml in cui viene impostato Kafka

5.3.1 Estensione

5.3.1.1 Aggiungere un broker Kafka Per aggiungere un ulteriore broker è necessario modificare il file docker-compose.yml.
All'interno del file è necessario creare un nuovo service in cui è necessario copiare il broker preesistente. modificandone: il nome del servizio, il nome del CONTAINER$_G$ e la mappatura della porte; prestando, inoltre, attenzione a mantenere le stesse porte interne.
Per poterlo utilizzare è necessario solamente specificare le nuove porte assegnate all'interno delle componenti.
Per modificare le configurazioni si consiglia di seguire il link seguente:

https://github.com/bitnami/bitnami-docker-kafka

5.4 Data collector

La componente *data collector* permette di trasferire i dati prodotti dai singoli gateway ed inviati nei vari topic di Kafka, all'interno di un database di tipo TIMESERIES$_G$.
Allo stesso tempo, permette di filtrare i dati collezionati ed inviare dei messaggi di avviso, all'interno di un apposito topic, nel caso in cui vengano rilevati dei valori anomali.
Il filtraggio dei dati viene fatto a partire dagli alert impostati all'interno della WEB APP$_G$ e salvati in un database relazionale (POSTGRESQL$_G$).

- La componente è stata sviluppata in Java 11.

5.4.1 Diagramma dei package

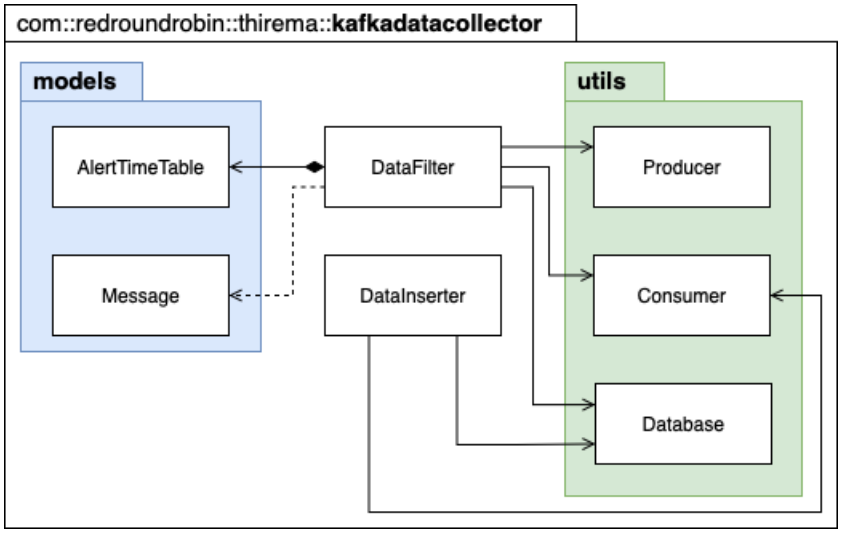

Figura 7: Diagramma dei package della componente data collector

5.4.2 Dipendenze esterne

La componente ha due dipendenze esterne:

- **KafkaProducer<K, V>**, classe concreta che implementa l'interfaccia Producer<K,V>. Svolge il compito di client per il cluster KAFKA$_G$, pubblicando dati all'interno di un topic. La classe DataFilter ne possiede un riferimento.
- **KafkaConsumer<K,V>**, classe concreta che implementa l'interfaccia Consumer<K,V>. Svolge il compito di client per il cluster Kafka, consumando i messaggi resenti all'interno di uno o più topic. La classi DataInserter e DataFilter ne possiedono un riferimento.

5.4.3 Diagramma delle classi

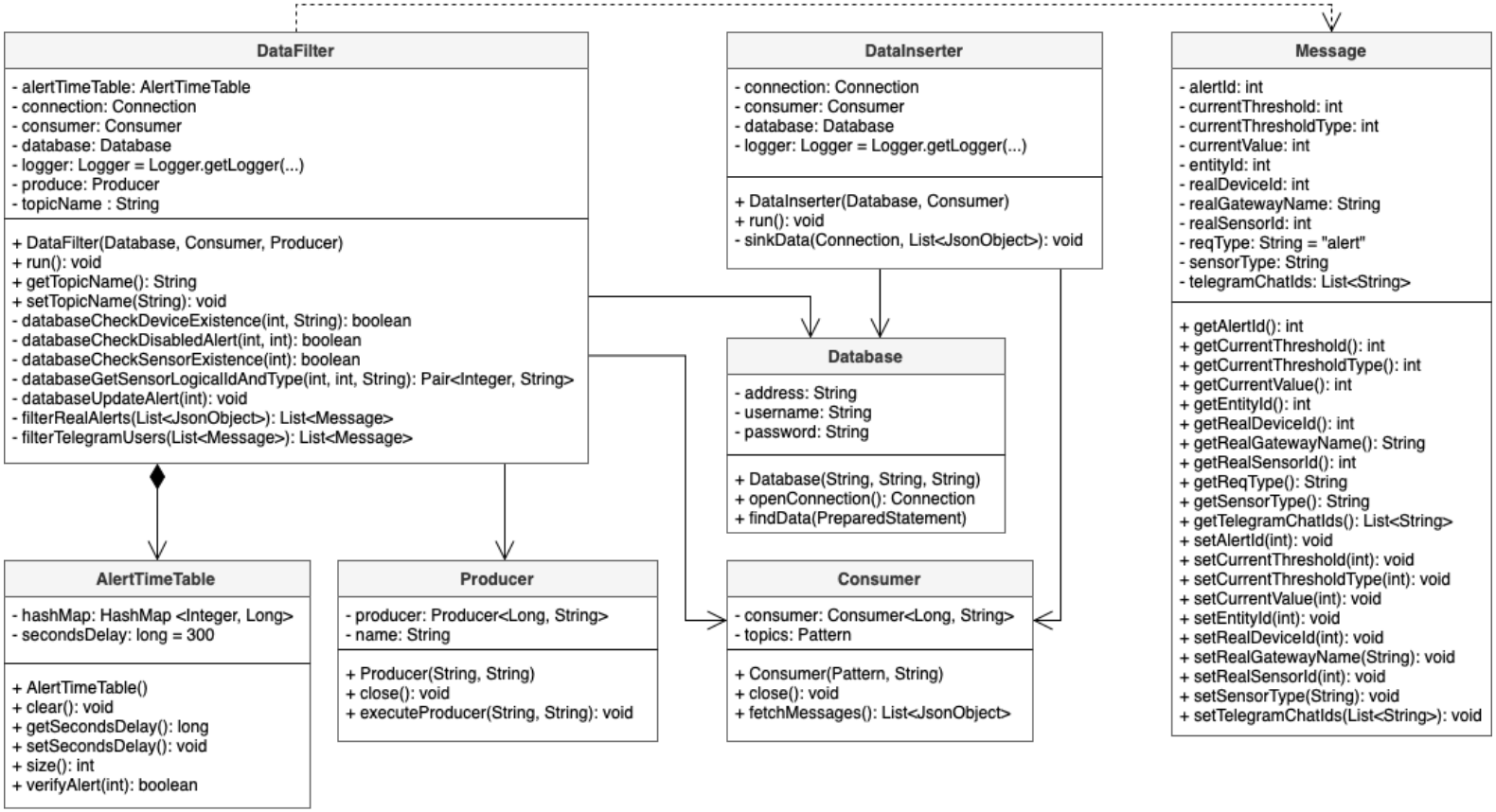

Figura 8: Diagramma delle classi della componente data collector

Come si vede nel diagramma sono presenti le seguenti classi:

- **Database**: rappresenta un database; i suoi campi propri sono infatti lo username, la password e l'indirizzo del database che rappresenta;
- **Consumer**: rappresenta un consumer di Kafka, che è infatti contenuto al suo interno. La classe viene utilizzata per "consumare" messaggi da uno o più topic di Kafka (in questo caso dai topic contenenti i dati reperiti tramite i vari gateway presenti nel sistema);

- **Producer**: rappresenta un producer di Kafka, che è infatti contenuto al suo interno. La classe viene utilizzata per "produrre" messaggi contenenti messaggi di alert all'interno di un apposito topic;
- **DataInserter**: è una classe al cui interno è presente un riferimento alla classe Database (che rappresenta TIMESCALE$_G$ in questo caso) ed alla classe Consumer. Il suo compito è quello di reperire i dati da tutti i topic collegati ad un gateway ed inserirli all'interno del database non relazionale;
- **DataFilter**: classe che contiene oltre che un riferimento a Database (che rappresenta Postgre) e a Consumer, anche un riferimento a Producer e AlertTimeTable; questo perché DataFilter dopo aver filtrato i dati, grazie ai suoi metodi (e grazie alla classe AlertTimeTable), inserisce i messaggi derivati dai dati consumati direttamente in un topic Kafka tramite un producer;
- **AlertTimeTable**: classe utilizzata per definire e far rispettare un certo delay tra l'invio di un messaggio di alert ed un messaggio successivo. Questo limite è stato fissato solo per avvisi di uno stesso sensore.
- **Message**: rappresenta un messaggio di alert inviato da DataFilter all'interno di un topic adibito questa funzione. Contiene infatti tutte le informazioni necessarie alle API per inviare un messaggio ad uno o più utenti iscritti, tramite la Web-app, alla ricezione di quel determinato alert.

5.4.4 Diagrammi di sequenza

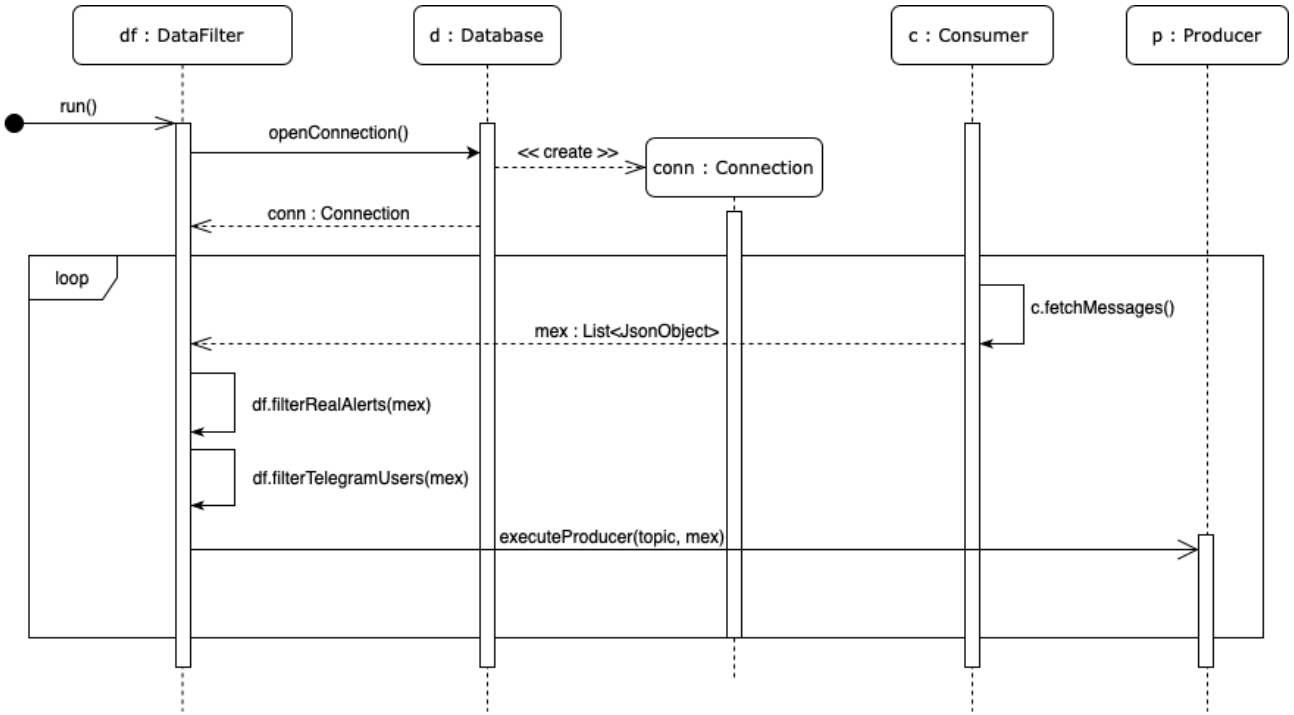

Figura 9: Diagramma di sequenza in cui viene mostrato il funzionamento del filtraggio dati nella componente data collector

Come si evince dal diagramma in alto, funzionamento di DataFilter è il seguente:

- DataFilter apre la connessione con il database relazionale Postgre;
- vengono quindi estratti i dati dai topic KAFKA$_G$ collegati ai gateway;
- i valori vengono filtrati, garantendo che dispositivo e sensore siano presenti nel database e che i valori rilevati superino effettivamente la soglia;
- i valori vengono filtrati nuovamente per verificare che gli utenti a cui dovrebbero arrivare abbiano configurato il BOT TELEGRAM$_G$ correttamente;
- infine vengono prodotti un insieme di messaggi di alert che, tramite un produttore vengono inseriti nel topic KAFKA$_G$ "alerts".

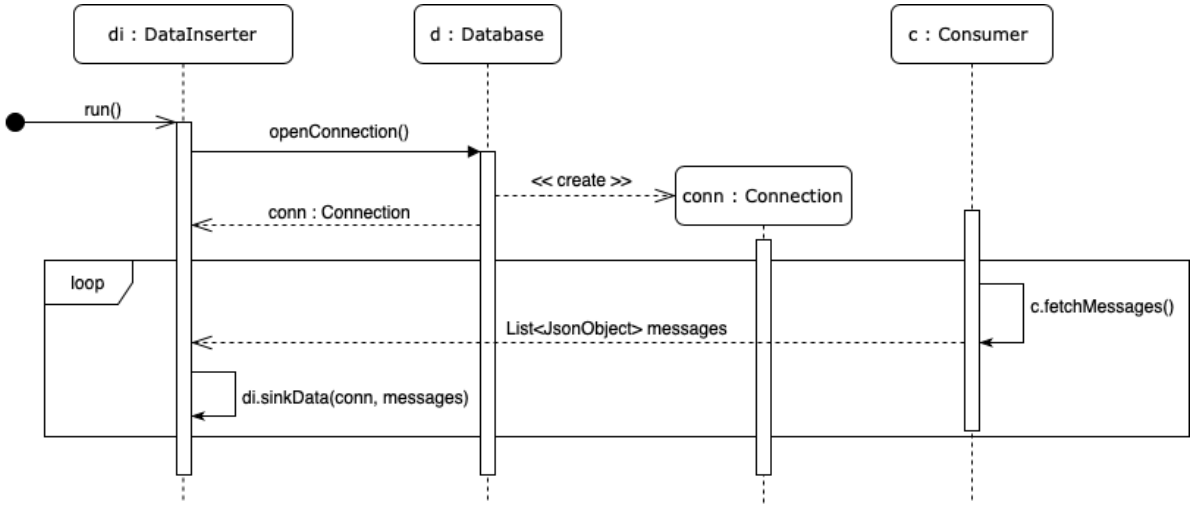

Figura 10: Diagramma di sequenza in cui viene mostrato il funzionamento dell'inserimento dati nella componente data collector

Il comportamento di DataInserter è il seguente:

- DataInserter apre una connessione con il database TIMESCALE$_G$;
- ogni qualvolta il suo Consumer trova dei nuovi dati nei topic collegati ai gateway, vengono inviati i messaggi al DataInserter;
- DataInserter infine inserisce i dati all'interno del database.

5.5 Database

All'interno dell'architettura sono utilizzati:

- un database relazionale, POSTGRESQL$_G$, che viene usato per il salvataggio delle configurazioni dei gateway, delle informazioni di utenti ed enti, oltre alle impostazioni dei grafici creati dagli utenti stessi;
- un database non relazionale, TIMESCALE$_G$, che viene usato per salvare i dati inviati dai dispositivi, le logs degli eventi e gli alert con i valori anomali rilevati.

Entrambi i database sono rilasciati tramite dockerfile, tramite il quale ne viene effettuata una prima configurazione.

5.5.1 Timescale

La struttura del database non relazionale è la seguente:

- Sensors
  - time: timestamptz
  - real_sensor_id: integer
  - real_device_id: integer
  - gateway_name: text
  - value: double
- Logs
  - time: timestamptz
  - user_id: integer
  - ip_address: varchar
  - operation: text
  - data: text

5.5.2 PostgreSql

La struttura del database SQL è rappresentata nello schema logico sottostante:

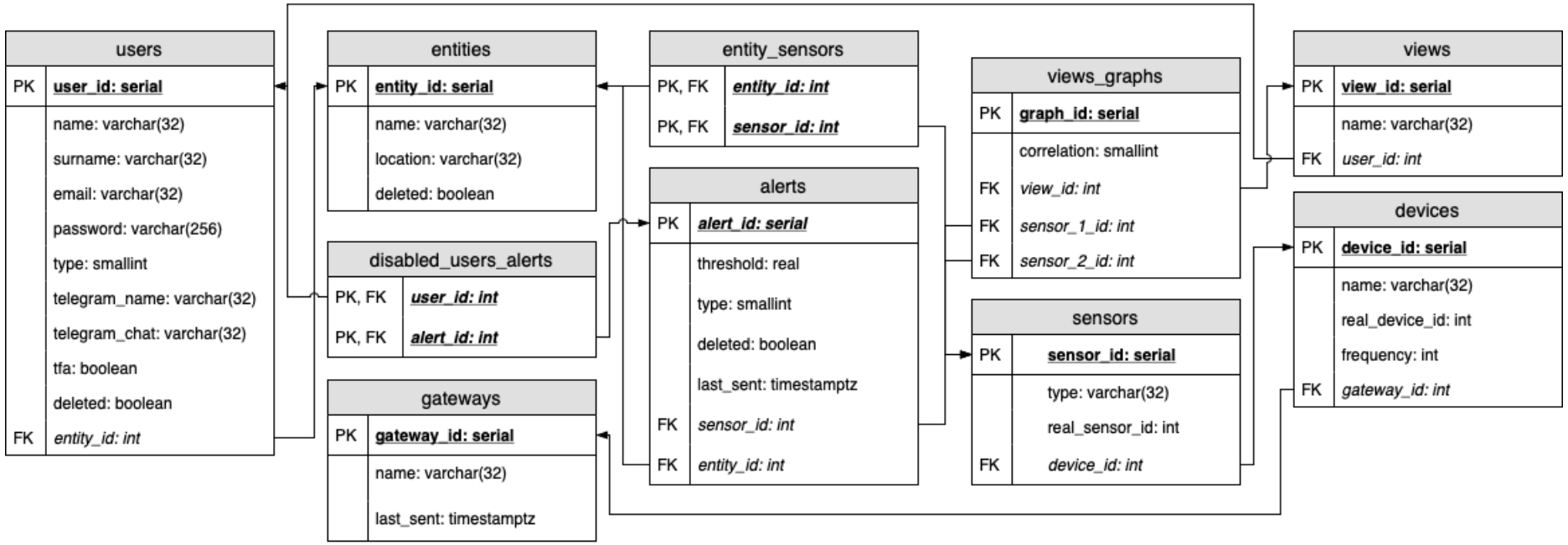

Figura 11: Diagramma logico del database relazionale

5.5.3 Estensioni

5.5.3.1 Duplicazione dei dati nel database Timescale Per aggiungere un fattore di duplicazione ai dati è necessario modificare il file docker-compose.yml inserendo un nuovo servizio: il servizio che si andrà a duplicare sarà quello di TIMESCALE$_G$, che sarà copiato e modificato.
Risulterà infatti necessario modificare il nome del servizio e del container, la mappatura delle porte (prestando attenzione a mantenere la porta interna specificata) ed infine i volumi.
Per effettuare la persistenza dei dati sarà infine necessario inserire il nuovo volume mappato all'interno di "volumes". Conclusa questa procedura sarà possibile inserire il nuovo database all'interno della componente DataCollector, specificando il corretto indirizzo.

5.5.3.2 Modifica impostazioni d'accesso ai database Per modificare le impostazioni di accesso ai database è necessario modificare le variabili d'ambiente all'interno dei rispettivi Dockerfile.

5.6 API

La componente API è il core dell'intera architettura; permette all'applicazione web di interfacciarsi con i due database menzionati precedentemente, oltre che con un BOT TELEGRAM$_G$.
La componente è stata sviluppata in Java 11, utilizzando dei FRAMEWORK SPRING$_G$: Spring boot, Spring security, Spring Kafka e Spring jpa; per l'autenticazione web è stato usato Json Web Token.

5.6.1 Diagramma dei package

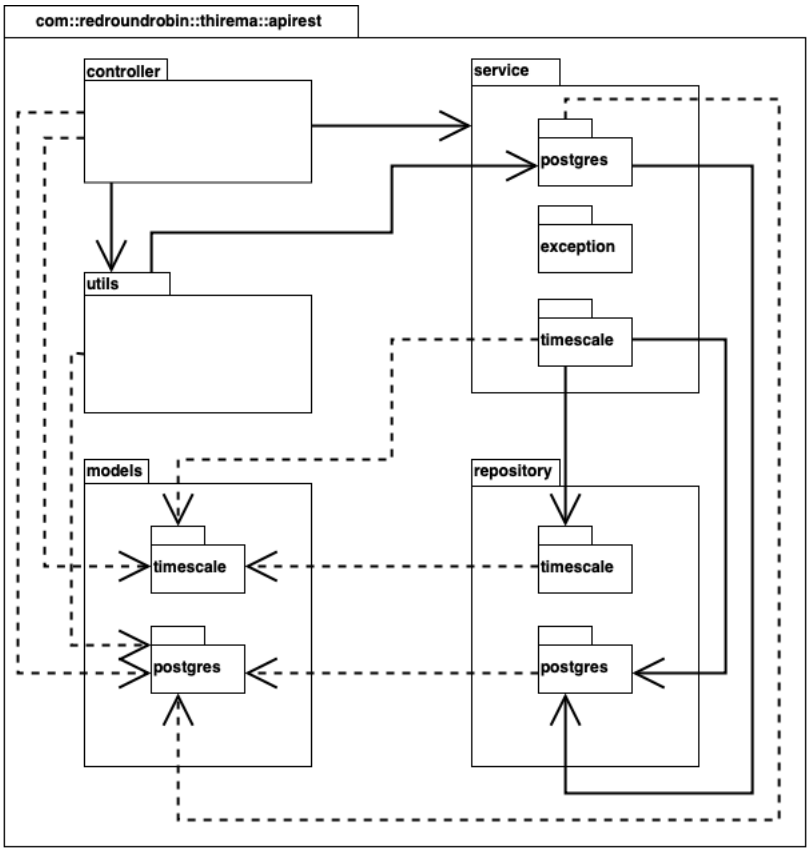

Figura 12: Diagramma dei packages per la componente API

5.6.2 Dipendenze esterne

La componente API he le seguenti dipendenze esterne:

- **Spring Boot:** che viene utilizzato per rendere le API$_G$ un microservizio;
- **Spring Boot Jpa:** contenuta nel framework precedente, viene utilizzata per permettere al package Repository e Config di interfacciarsi direttamente con il database;
- **Spring Boot Starter Security:** contenuta nel framework Spring Boot, viene utilizzata per gestire la sicurezza nell'accesso alle API$_G$;
- **SpringBoot Starter Test:** contenuta nel framework Spring Boot, viene utilizzata per eseguire i test;

- **org.Moquito:** libreria usata per fare simulare il comportamento di alcune funzioni, in fase di test;
- **io.jsonwebtoken:** libreria usata per la creazione di token necessari all'accesso ad aree riservate delle API$_G$;
- **Spring Kafka:** libreria utilizzata per la comunicazione con KAFKA$_G$;
- **Spring Doc:** libreria utilizzata per la visualizzazione delle richieste API$_G$ tramite interfaccia web;
- **org.postgreSql.Driver:** che viene utilizzato per la connessione con DBMS di tipo Postgre.

5.6.3 Diagrammi delle classi

Al fine di semplificare la comprensione delle dipendenze della componente API$_G$, si è deciso di suddividere i diagrammi per package, mostrando in dettaglio quelli più significativi.

Package Utils
All'interno del package Utils sono presenti le seguenti classi:

- **JwtRequestFilter**: la classe dipende dalle classi UserService e JwtUtils ed estende la classe OncePerRequestFilter, permettendo di filtrare le richieste effettuate dal bot Telegram o dalla Web-app verso le API$_G$;
- **JwtUtil**: la classe implementa l'interfaccia Component del FRAMEWORK SPRING$_G$. Più nel dettaglio la classe JwtUtil permette di gestire i token: è posssibile infatti creare varie tipologie di token che permettono l'autenticazione e la richiesta di dati alle API. Sempre all'interno di questa classe è presente un metodo per verificare l'autenticità di un certo token;
- **CustomAuthenticationManager**: questa classe dipende da UserService ed implementa l'interfaccia AuthenticationManager del framework Spring, permettendo quindi la gestione e la personalizzazione dell'autenticazione.

Package Config
Il diagramma del package Config mostra le dipendenze delle classi di configurazione della componente API$_G$.
Il package Config contiene le classi in cui vengono effettuate le configurazioni di deiverse componenti delle API. Sono infatti presenti:

- **PostgresConfig** e **TimescaleConfig**: queste classi vengono utilizzate per configurare l'accesso ai nostri database ed implementano le interfacce PropertySource, JpaRepository e Configuration del framework Spring;
- **KafkaConsumerConfig**: questa classe viene utilizzata per configurare l'accesso al broker Kafka contenuto all'interno del sistema. Per poter essere implementata ha bisogno delle dipendenze verso le interfacce Configuration ed EnableKafka del framework Spring;
- **SecurityConfig**: questa classe viene utilizzata per configurare i servizi di sicurezza impiegati dalle API, e permettere o meno ad alcune richieste HTTP di accedere a questa componente. Ha delle dipendenze verso UserService e WebSecurityAdapter mentre implementa l'interfaccia EnableWebSecurity.

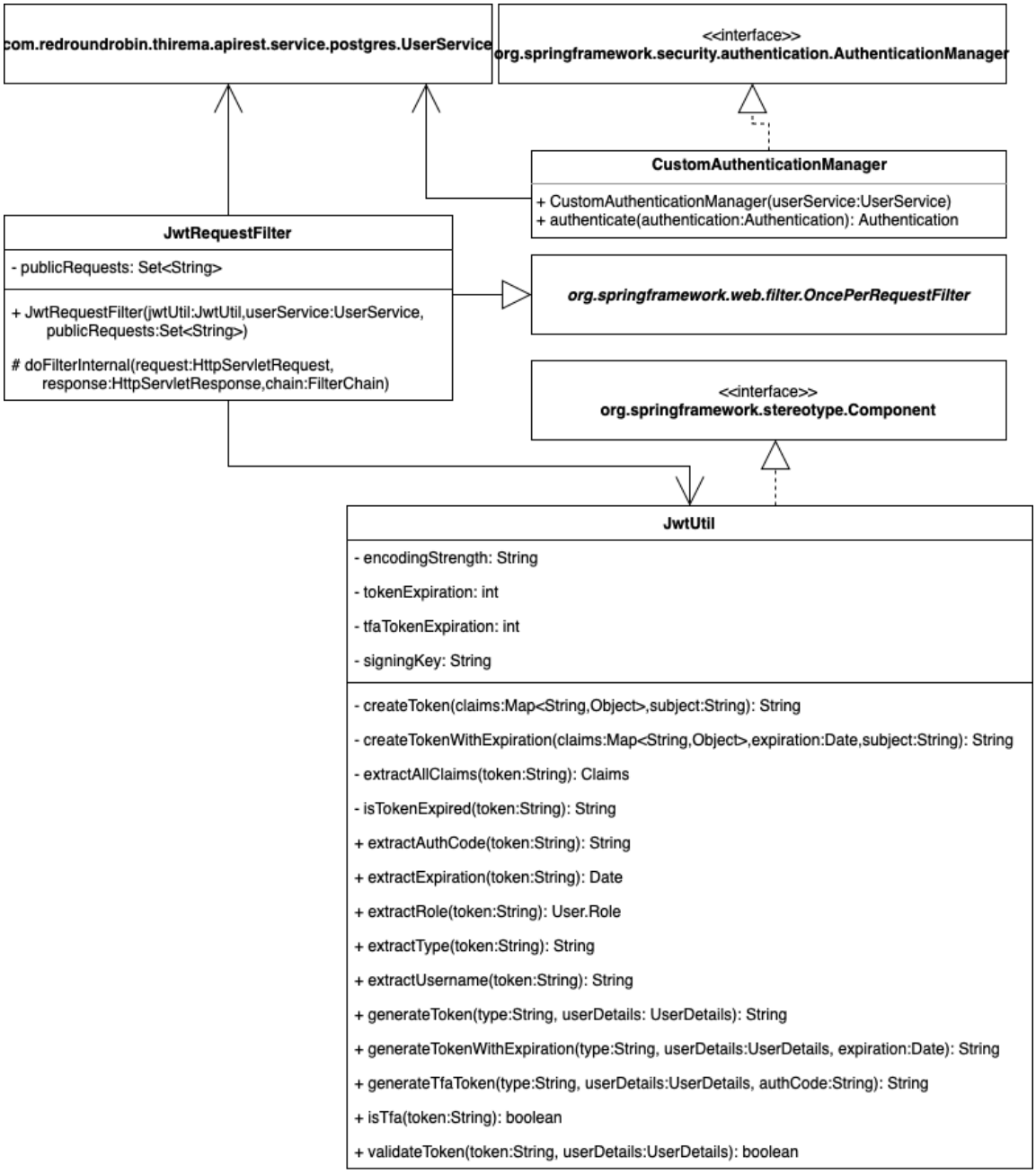

Figura 13: Diagramma del package utils della componente API

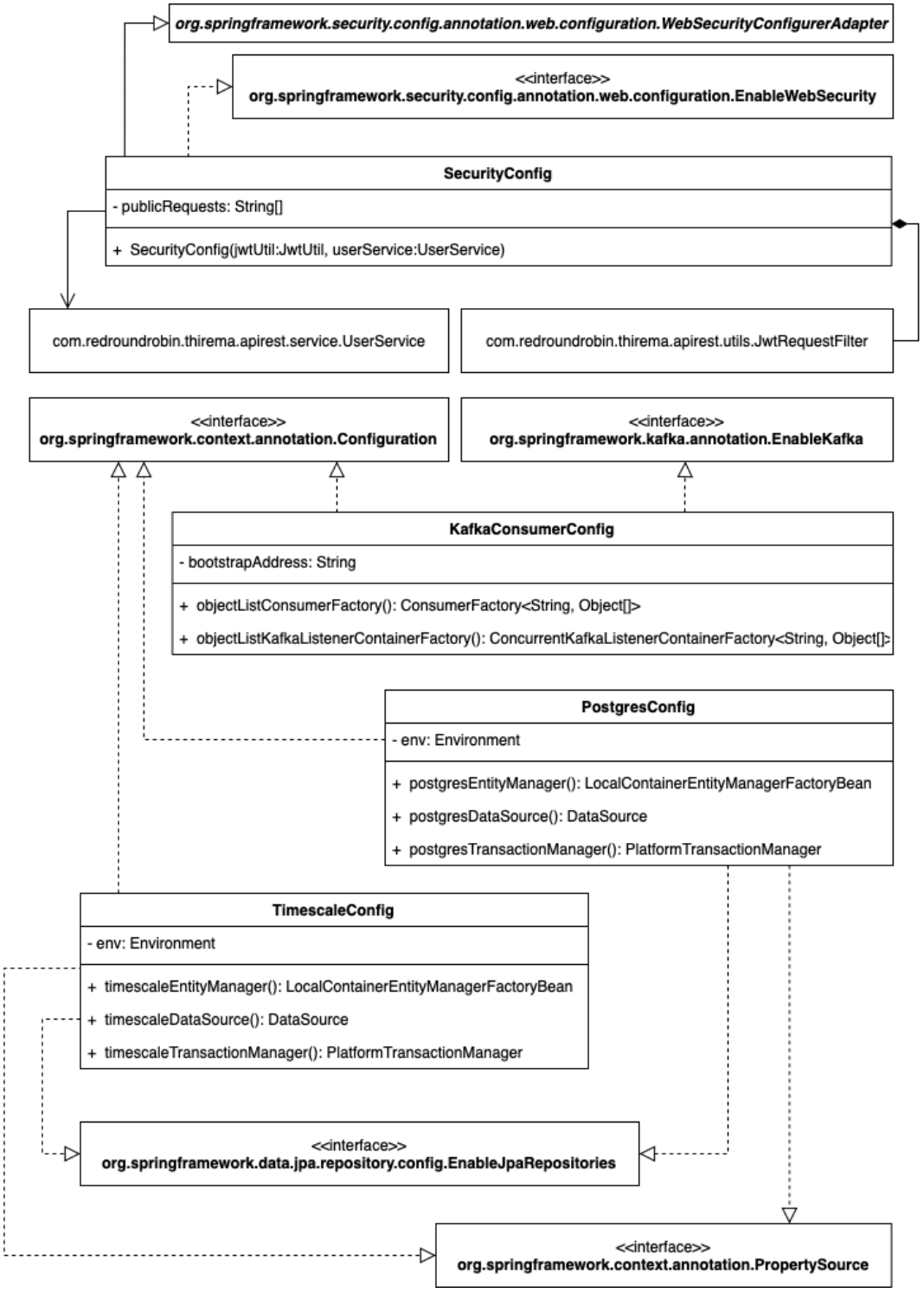

Figura 14: Diagramma del package config della componente API

Package Controllers

Il package Controllers visualizzabile all'interno del file *Immagini/API-PackageController.png* contiene i controller per mappare le richieste GET e POST che è possibile ricevere da parte della WEB APP$_G$ o di TELEGRAM$_G$.
Ogni classe per poter funzionare implementa l'interfaccia RequestMapping e RestController del framework Spring. Inoltre estendono la classe astratta CoreController che permette di gestire le varie autorizzazioni che hanno gli utenti per effettuare o meno una certa richiesta. Le classi presenti nel package sono le seguenti:

- **EntityController**: permette di gestire le richieste che riguardano i dati da reperire o inserire dal database per quanto riguarda gli enti presenti nel sistema. Contiene un riferimento ad EntityService;
- **ViewController**: permette di gestire le richieste che riguardano la gestione della configurazioni di una o più view all'interno della omonima sezione della Web-app. Contiene un riferimento a ViewService;
- **ViewGraphController**: permette di gestire le richieste che riguardano le configurazioni dei vari grafici presenti all'interno di una specifica view all'interno della Web-app. Contiene un riferimento a ViewGraphService;
- **DataController**: permette di gestire le richieste che riguardano il reperimento di valori di un determinato sensore presenti all'interno del database Timescale. Contiene un riferimento a SensorService;
- **SensorController**: permette di gestire le richieste che riguardano i sensori salvati all'interno del database relazionale. A differenza di DataController questa classe non è incentrata sul raccogliere i dati dei sensori. Contiene un riferimento a SensorService;
- **DeviceController**: permette di gestire le richieste che riguardano il reperimento dei dispositivi e sensori sulla base di un certo ente o id di un certo dispositivo. Contiene un riferimento SensorService e DeviceService;
- **GatewayController**: permette di gestire le richieste che riguardano i vari Gateway presenti nel sistema; Per ogni gateway è possibile reperire i dispositivi ed i sensori. Contiene un riferimento a DeviceService, GatewayService e SensorService;
- **UserController**: permette la gestione delle richieste che riguardano la creazione, eliminazione e gestione dei dati degli utenti presenti nel sistema;
- **AuthController**: questa classe permette di gestire le richieste che riguardano l'autenticazione (sia normale che 2FA) degli utenti all'interno della Web-app o del bot Telegram. Contiene un riferimento verso CustomAuthenticationManager e TelegramService.

Package Models

Nel diagramma in alto vengono descritte le dipendenze del package Models, che rappresentano le relazioni che ci sono fra le varie entità del database relazionale. Le classi presenti nel diagramma sono le seguenti:

- **postgres.User**: rappresenta l'entità User e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede un riferimento a postgres.User.Role;
- **postgres.Entity**: rappresenta l'entità Entity e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede un riferimento a postgres.Sensor;
- **postgres.Gateway**: rappresenta l'entità Gateway all'interno e le sue relazioni del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity;
- **postgres.View**: rappresenta l'entità view e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede un riferimento a postgres.User;
- **postgres.Sensor**: rappresenta l'entità Sensor e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede un riferimento a postgres.Device;
- **postgres.Device**: rappresenta l'entità Device e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede un riferimento a postgres.Gateway;
- **postgres.Alert**: rappresenta l'entità Alert e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede dei riferimenti a postgres.Entity e postgres.Sensor;
- **postgres.Viewgraph**: rappresenta l'entità ViewGraph e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity e possiede dei riferimenti a postgres.View e postgres.ViewGraph.Correlation;
- **postgres.ViewGraph.Correlation**: rappresenta l'entità Correlation di un certo grafico e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity
- **postgres.User.Role**: rappresenta l'entità Role di un determinato utente e le sue relazioni all'interno del database relazionale. Implementa le interfacce Serializable e javax.persistence.Entity;
- **timescale.Sensor**: rappresenta l'entità Sensor all'interno del database non relazionale. Implementa le interfacce Serializable e javax.persistence.Entity;
- **timescale.Log**: rappresenta l'entità Log all'interno del database non relazionale. Implementa le interfacce Serializable e javax.persistence.Entity.

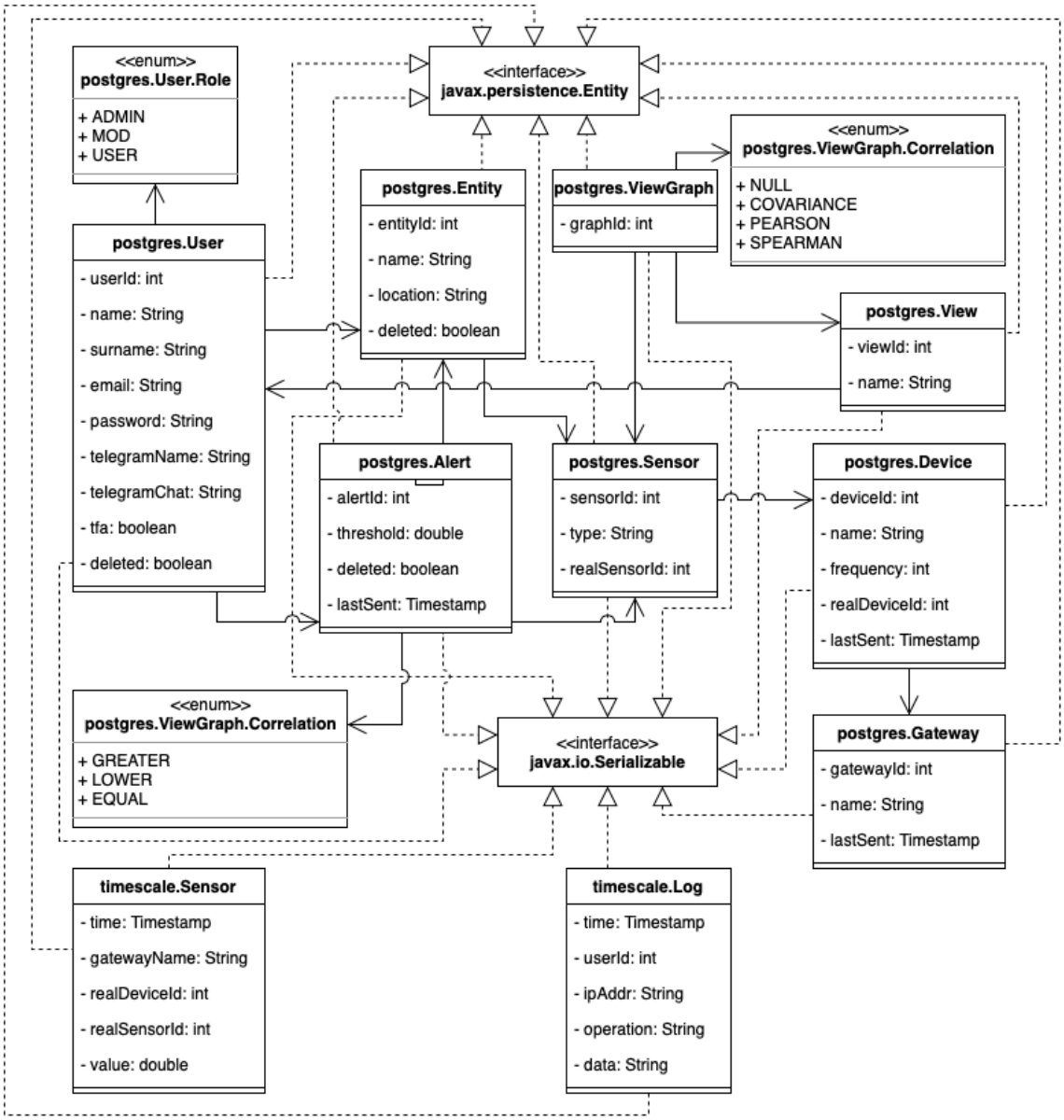

Figura 15: Diagramma del package models della componente API

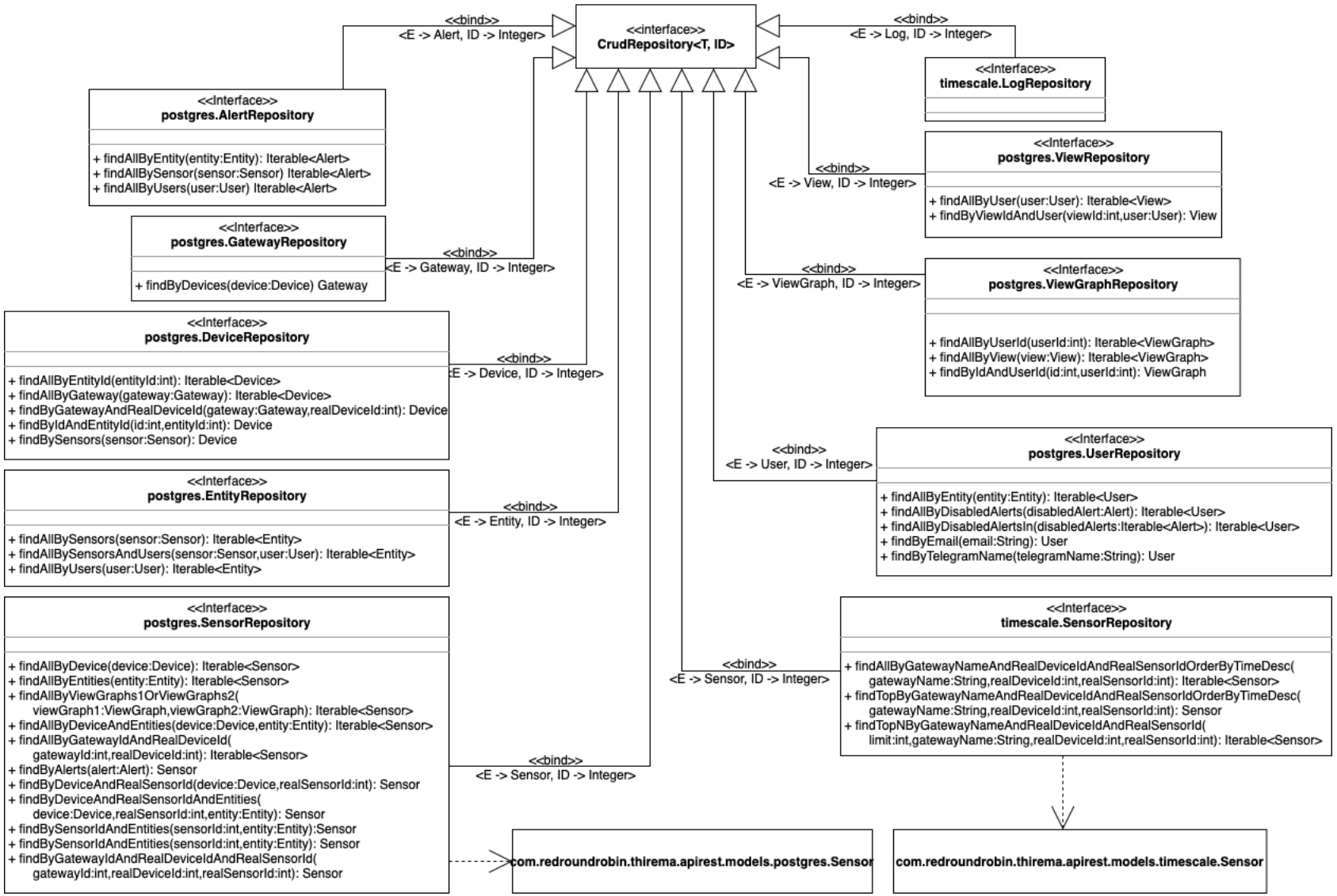

Figura 16: Diagramma del package repository della componente API

Package Repository Nel diagramma che rappresenta le dipendenze del package Repository della componente API_G, sono presenti le interfacce che rappresentano le query che vengono effettuate a database nelle varie tabelle. Tutte le interfacce estendono l'interfaccia CrudRepository. Le interfacce presenti del package rappresentato dal diagramma sono:

- **postgres.AlertRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire gli alert per uno specifico ente, sensore o utente;
- **postgres.GatewayRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire il gateway a partire da uno specifico dispositivo;
- **postgres.DeviceRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire i dispositivi a partire da un ente, un sensore o da un identificativo;
- **postgres.EntityRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire un ente a partire dai suoi sensori associati o dai suoi utenti;
- **postgres.SensorRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire uno o più sensori a partire da un dispositivo, un ente, una view, un gateway o un alert. Oltre ad estendere l'interfaccia CrudRepository utilizza anche la classe models.postgres.Sensor;
- **postgres.UserRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire uno o più utenti a partire da un ente, email o username Telegram oppure dall'avere o meno degli alert disabilitati;
- **postgres.ViewGraphRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire i grafici a partire da un utente o da una vista;
- **timescale.LogRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire i log all'interno del database non relazionale;
- **timescale.SensorRepository**: questa interfaccia rappresenta le query che è possibile fare per reperire i dati di un sensore a partire dal nome del gateway e dall'id del sensore.

Package Service Nel diagramma delle classi all'interno del file *Immagini/API-PackageService.png* che rappresenta il package Service sono rappresentate classi che effettuano le query a database ad un livello di astrazione più alto rispetto al package Repository. Quest'ultimo viene infatti utilizzato dai service per effettuare le query a database. Ogni classi del package Service implementa l'interfaccia Service del framework Spring. Le classi presenti nel package rappresentato nel diagramma sono le seguenti:

- **postgres.ViewService**: questa classe permette di reperire dati relativi alle View dal database relazionale. Ha dei riferimenti verso ViewRepository e UserRepository;
- **postgres.ViewgraphService**: questa classe permette di reperire dati relativi ai grafici relativi ad una view dal database relazionale. Ha dei riferimenti verso ViewRepository, ViewGraphRepository e SensorRepository;
- **postgres.SensorService**: questa classe permette di reperire dati relativi ai sensori dal database relazionale. Ha dei riferimenti verso ViewgraphRepository, SensorRepository, AlertRepository ed EntityRepository;
- **TelegramService**: questa classe permette di reperire dati relativi all'autenticazione a due fattori o all'invio di comandi dal database relazionale. Contiene un riferimento a RestTemplate del framework Spring;
- **postgres.GatewayService**: questa classe permette di reperire dati relativi ai gateway dal database relazionale. Possiede dei riferimenti verso GatewayRepository e DeviceRepository;
- **postgres.EntityService**: questa classe permette di reperire dati relativi agli enti dal database relazionale. Possiede dei riferimenti a EntityRepository, UserRepository e SensorRepository;
- **postgres.DeviceService**: questa classe permette di reperire dati relativi ai dispositivi dal database relazionale. Possiede dei riferimenti verso DeviceRepostory, GatewayRepository, EntityRepository e SensorRepository;
- **postgres.AlertService**: questa classe permette di reperire dati relativi agli alert dal database relazionale. Possiede dei riferimenti versoAlertRepository, EntityRepository, UserRepository e SensorRepository;
- **postgres.UserService**: questa classe permette di reperire dati relativi agli utenti dal database relazionale. Possiede dei riferimenti verso UserRepository, AlertRepository ed EntityRepository. Inoltre estende l'interfaccia UserDetailService del framework Spring;
- **timescale.LogService**: questa classe permette di reperire dati relativi ai logs di sistema dal database non relazionale. Ha un riferimento all'interfaccia LogRepository;
- **timescale.SensorService**: questa classe permette di reperire dati relativi ai dati di Timescale dei sensori immagazzinati nel database relazionale. Ha infatti due riferimenti distinti: uno verso postgres.SensorRepository ed uno verso timescale.SensorRepository;

5.6.4 Diagrammi di sequenza

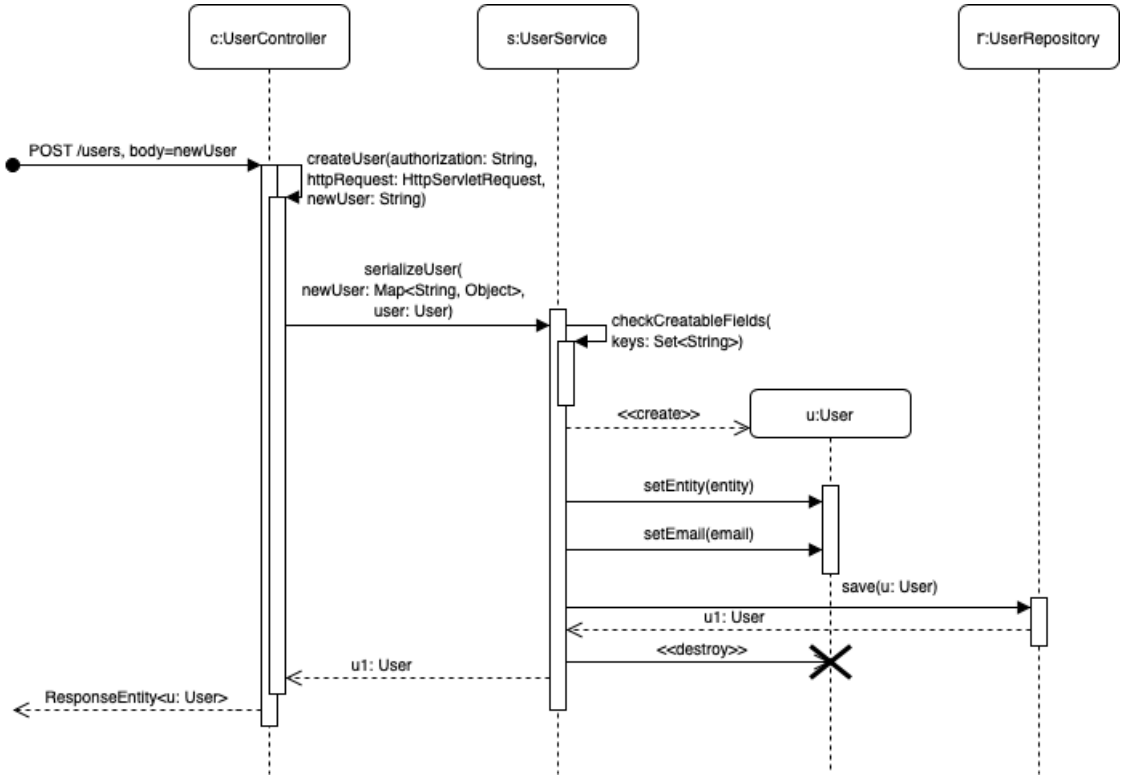

Figura 17: Diagramma di sequenza che mostra l'inserimento di un utente all'interno della componente API

Nel diagramma di sequenza in alto, alla ricezione di una richiesta POST dalla WEB APP$_G$, lo UserController chiede allo UserService di inserire un nuovo utente, dopo aver verificato che la richiesta provenisse da un utente con l'autorizzazione necessaria a crearne uno nuovo.
Viene creata poi un'istanza di User che, dopo averne impostato i campi, viene inviata a UserRepository, chiedendone il salvataggio nel database.
Viene infine restituita l'entità inserita ed inviata in risposta alla WEB APP$_G$ in formato JSON$_G$.

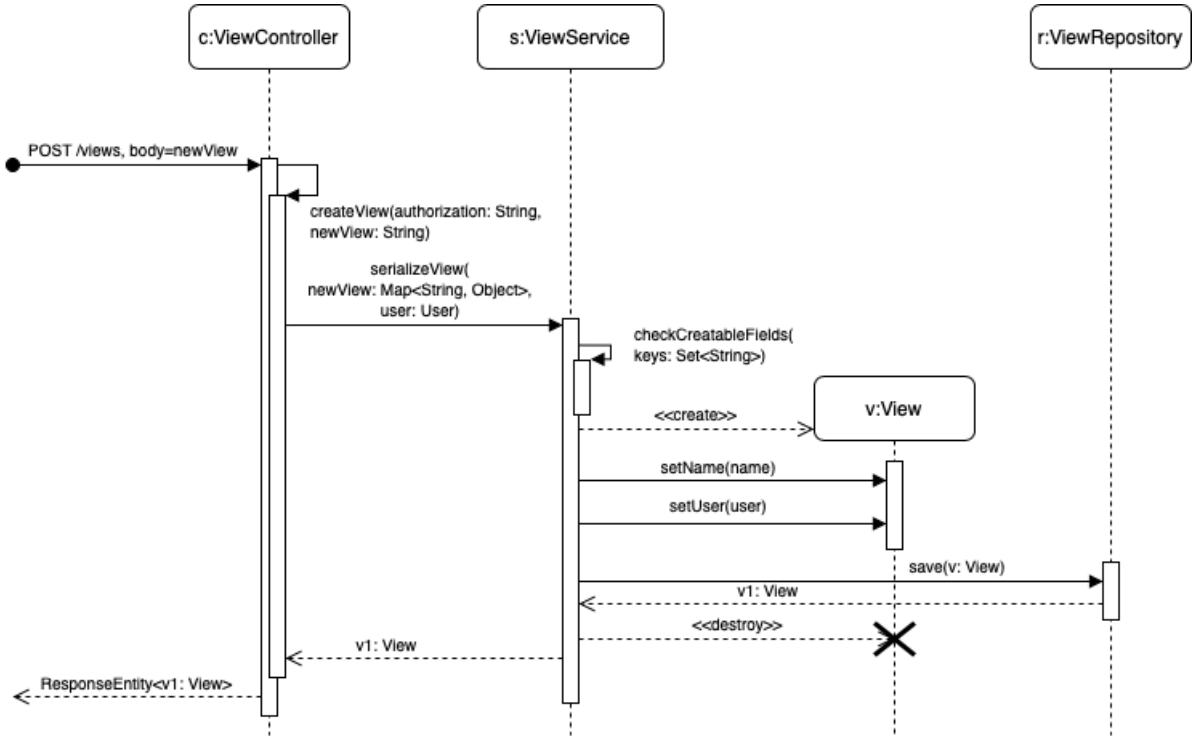

Figura 18: Diagramma di sequenza che mostra l'inserimento di una view all'interno della componente API

Il diagramma descrive l'inserimento di una view all'interno del database. Il procedimento inserito è molto simile a quello descritto per l'inserimento di un utente.

5.6.5 Documentazione OpenAPI

Per le API è stato generato un documento in formato OpenAPI con un file YAML. Questo documento è stato convertito in PDF per comodità di lettura, e contiene tutte le specifiche delle API realizzate con le relative chiamate e i relativi codici di errore in risposta.
I file in questione si possono trovare ai seguenti percorsi:

- `OpenAPI/riot-api.v1.pdf` - documento PDF descrittivo di tutte le API realizzate;
- `OpenAPI/riot-api.v1.yaml` - documento in formato YAML auto-generato tramite Spring.

5.6.6 Estensione

5.6.6.1 Aggiungere una richiesta API Per inserire una nuova richiesta, se non si usano le classi già esistenti, è necessario che la nuova classe creata implementi l'interfaccia RestController, tramite la notazione *@RestController*, da inserire precedentemente alla definizione della classe.
All'interno della classe è necessario definire una funzione usando una notazione di tipo mapping, ad esempio "@GetMapping", inserendo all'interno delle parentesi tonde la stringa associata all'URI della richiesta che si vuole implementare. Ad esempio:

```
@GetMapping(value = \{"/{deviceId:.+}/sensors"\})
```

Per esporre una risorsa senza necessità di autenticazione è necessario aggiungere all'array publicRequests l'URL della richiesta. L'array publicRequests è un attributo della classe SecurityConfig.

5.7 Bot Telegram

La componente BOT TELEGRAM$_G$ permette di ricevere codici di autenticazione a due fattori, notifiche di ALERT$_G$ ed inviare direttamente dei comandi ai singoli dispositivi, per alterarne lo stato.
La componente è stata sviluppata usando JavaScript ed i moduli Axios, HTTP e Telegraf.

5.7.1 Diagramma delle classi

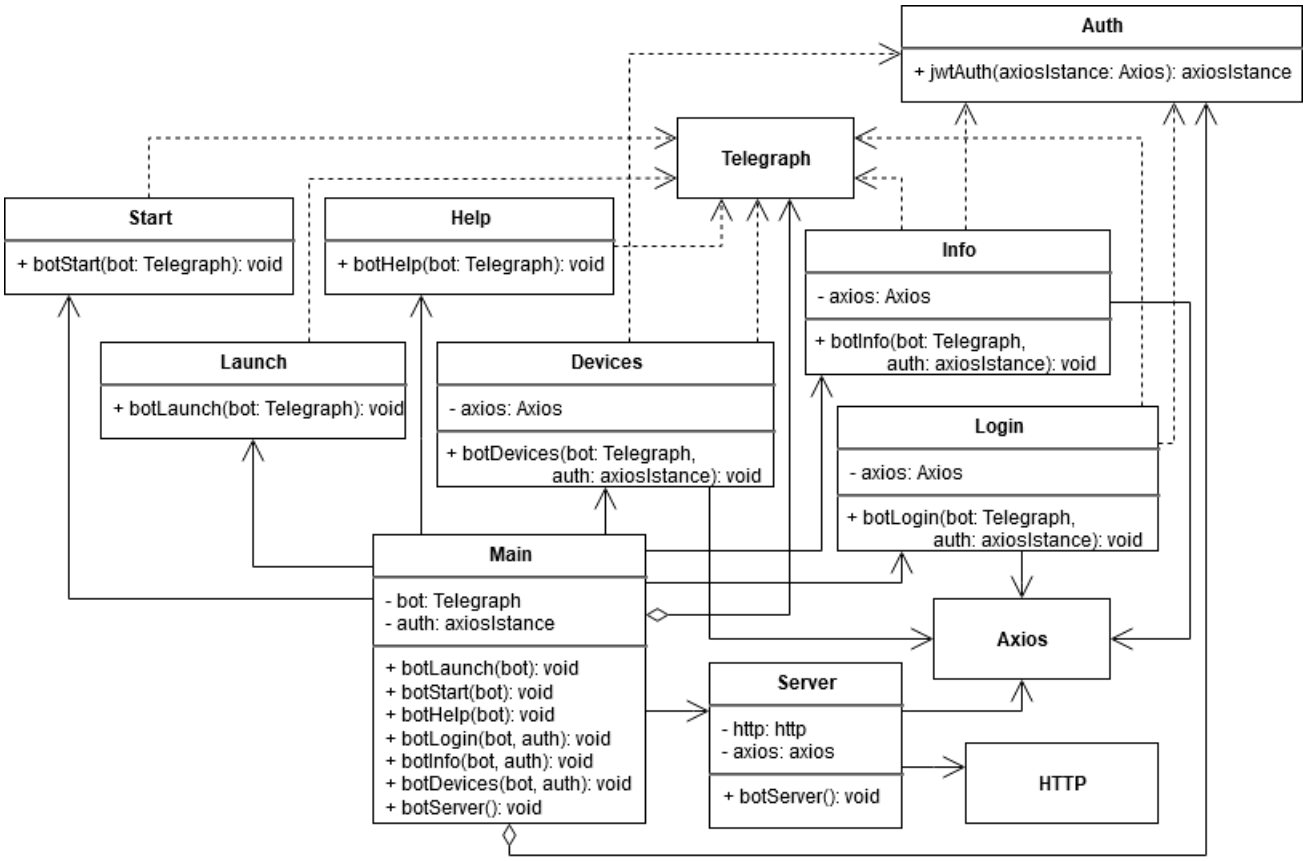

Figura 19: Diagramma delle classi della componente bot Telegram

Nel diagramma in figura sono presenti le seguenti parti:

- **Main**: questa parte racchiude tutte le richieste ed i comandi che è possibile effettuare dal bot e per questo ha delle dipendenze verso tutti i comandi disponibili, oltre che verso il modulo Telegraf;
- **Server**: questa componente viene utilizzata per implementare le funzionalità di ricezione di richieste da parte delle API per quanto riguarda l'invio di alert o l'invio del 2FA token. Ha un riferimento verso HTTP e Axios;

- **Start**: questa componente è necessaria per permettere ad un utente di far partire un'istanza del bot Telegram sul proprio dispositivo;
- **Help**: il compito il questa componente è implementare il comando "help" e mostrare di conseguenza la lista di tutti i comandi disponibili agli utenti oltre che una breve descrizione della procedura di autenticazione tramite Telegram;
- **Devices**:il compito di questa componente è implementare il comando "devices": questo comando mostra agli amministratori una lista di tutti i dispositivi che possiedono almeno un sensore abilitato alla ricezione dei comandi. In seguito è possibile selezionare un dispositivo, un sensore ed infine scegliere quale comando inviare. Ha dei riferimenti verso Axios e Telegraf/Markup;
- **Info**: il compito di questa componente è implementare il comando "info", permettendo di visualizzare le informazioni del proprio account. Ha un riferimento verso Axios;
- **Login**: il compito di questa componente è implementare il comando "login", permettendo all'utente di autenticarsi ed autorizzare il proprio account Telegram all'esecuzione dei diversi comandi. Ha un riferimento verso Axios;
- **Config**: questa componente ha li compito di caricare i comandi disponibili del proprio bot in modo da abilitare la funzione di suggerimento dei comandi all'interno del bot Telegram. Ha un riferimento verso Axios;
- **jwtAuth**: questa componente è utilizzata per implementare una parte del comando "login". Più in dettaglio imposta il token di autenticazione ottenuto tramite le API.

5.7.2 Dipendenze esterne

La componente ha tre dipendenze esterne:

- **Telegraf**, modulo che permette di collegarsi con le API ufficiali di TELEGRAM$_G$. Ogni comando ne ha un riferimento;
- **Axios**, modulo che permette di effettuare richieste POST e GET e di ritornare una risposta. Viene utilizzato da Server, Login e Status per comunicare con le API e per inviare uno o più messaggi agli utilizzatori del bot;
- **HTTP**, modulo che permette di creare un server HTTP per restare in ascolto di eventuali richieste. Viene utilizzato da Server per ascoltare eventuali richieste delle API.

5.7.3 Diagramma di sequenza

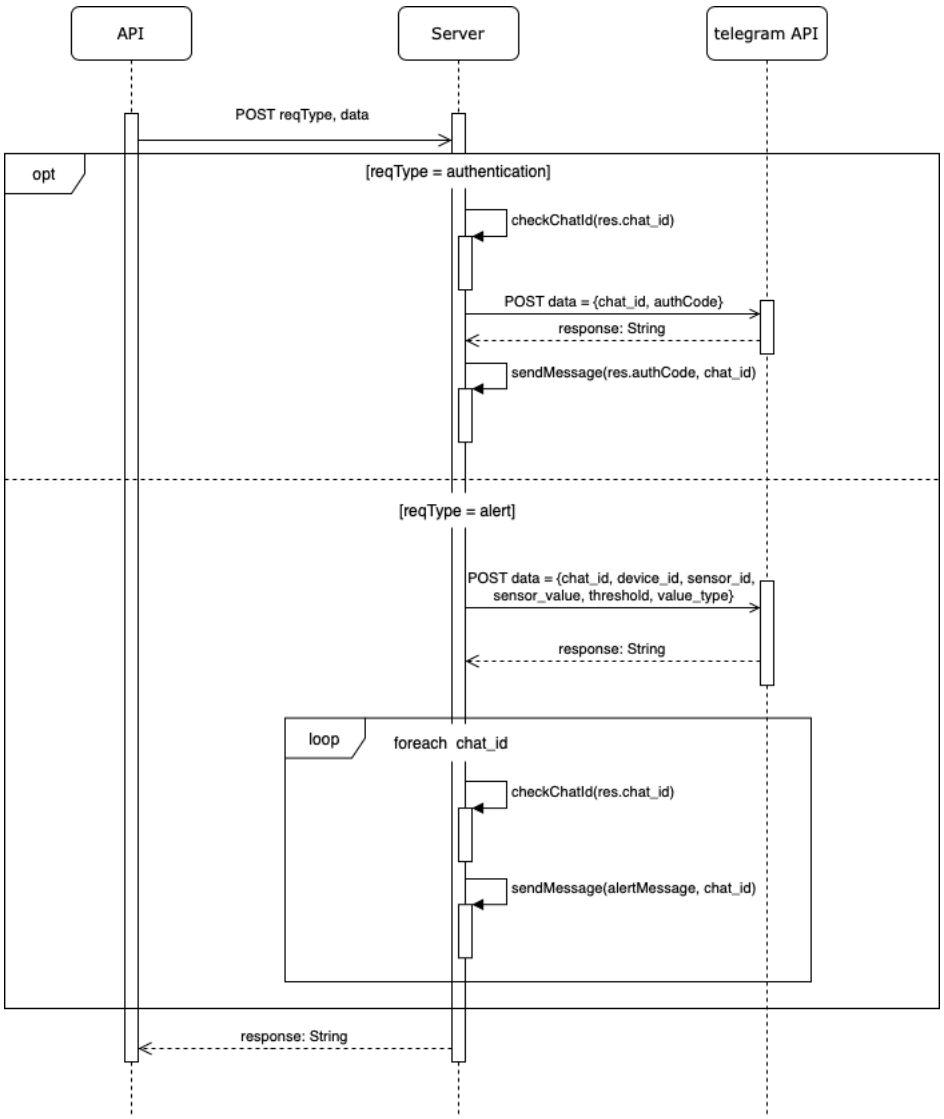

Figura 20: Diagramma di sequenza che riporta la ricezione di una richiesta POST delle API all'interno della componente bot Telegram

Nel diagramma di sequenza in alto viene mostrata la ricezione di una richiesta POST dalle API. La richiesta può essere di due tipologie:

- **authentication:** in cui vengono inviati dalle API un chat Id ed un codice di autenticazione; quest'ultimo dovrà poi essere inviato al chat Id specificato per permettere all'utente l'autenticazione sulla WEB APP$_G$;
- **alert:** in cui vengono mandati dalle API una lista di chat id ed un insieme di dati che poi andranno composti in un messaggio ed inviati a tutti i chat Id specificati.

5.7.4 Estensione

5.7.4.1 Inserimento di un nuovo comando Per inserire un nuovo comando all'interno del bot è necessario creare un file .js all'interno della cartella commands.
Poiché sarà poi necessario esportare questo comando, per poi inserirlo all'interno di main.js, l'intestazione del nuovo comando dovrà essere del tipo:

```
const botNomeComando = (bot) => {
bot.command( "nomeComando", (param) => {
  ...
  });
};
```

5.8 Web application

La componente WEB APP$_G$ ha il compito di interfacciare gli utenti con i dispositivi censiti dal sistema e visibili al loro ENTE$_G$ di appartenenza.
Le principali funzionalità messe a disposizione riguardano la visualizzazione di grafici contenenti i dati di determinati sensori, la modifica delle configurazioni dei gateway e l'aggiunta o la rimozione di dispositivi e/o sensori.
La componente è stata sviluppata utilizzando i FRAMEWORK LARAVEL$_G$ e VUE.JS$_G$.

5.8.1 Diagramma dei package

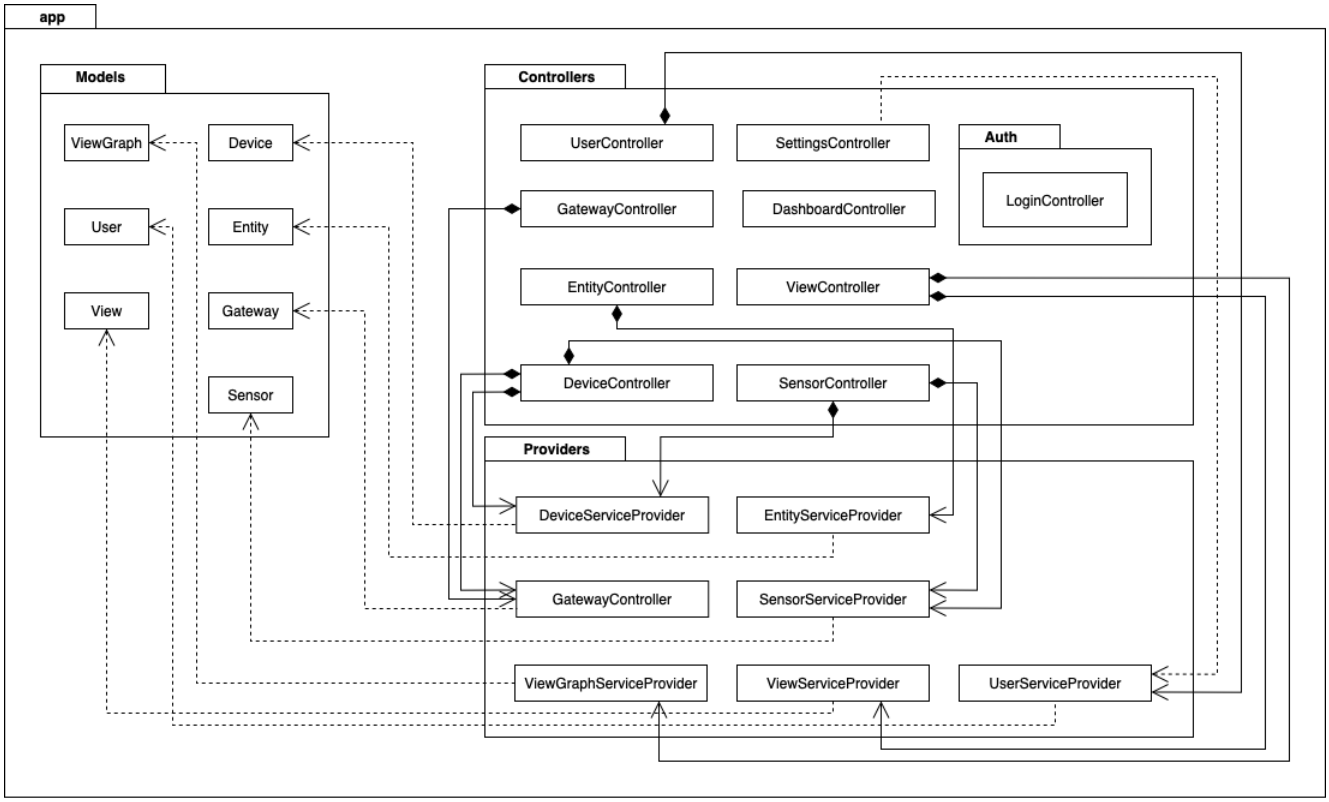

Figura 21: Diagramma dei package della componente web app

5.8.2 Dipendenze esterne

La componente ha le seguenti dipendenze esterne:

- **Laravel:** framework alla base della WEB APP$_G$, da cui vengono prese le basi dei controllers e dei models. Inoltre è il gestore delle view;
- **Laravel/UI:** viene utilizzata per l'autenticazione all'interno della web app;
- **guzzlehttp/guzzle:** viene utilizzata per effettuare richieste HTTP in linguaggio PHP;
- **Axios:** viene utilizzata per effettuare delle richieste HTTP in JAVASCRIPT$_G$;
- **Datatables.js:** viene utilizzata per la paginazione delle tabelle;

- **Apexcharts:** viene utilizzata per fare i grafici visibili all'interno dell'applicazione web;
- **davejamesmiller/laravel-breadcrumbs:** questa dipendenza viene utilizzata per generare i breadcrumb all'interno delle pagine visualizzate;
- **Vue.js:** framework che permette, tra le altre cose, di utilizzare array e variabili all'interno delle pagine per permetterne una visualizzazione dinamica dei contenuti;
- **Bootstrap:** framework utilizzato per la creazione dell'interfaccia grafica dell'applicazione web;
- **JQuery:** libreria utilizzata da BOOTSTRAP$_G$ per molte delle sue componenti;
- **Popper.js:** altra libreria utilizzata da BOOTSTRAP$_G$.

5.8.3 Diagrammi delle classi

Nel diagramma delle classi della componente WEB APP$_G$, rappresentato nel file *Immagini/WebApp-Classi.png*, sono presenti principalmente i provider ed i controller: i provider sono le componenti che effettuano le richieste API$_G$ da parte della web app e lavorano quindi a basso livello.
I controller invece dialogano con i provider ad un livello di astrazione più alto. Le pagine view non sono presenti in quanto il framework LARAVEL$_G$ le gestisce tramite pagine HTML. Le classi presenti nel diagramma sono quindi:

- **Model**: tutte le classi che rappresentano un oggetto estendono la classe Model:
  - **User**: questa classe rappresenta gli utenti del sistema;
  - **Entity**: questa classe rappresenta gli enti presenti nel sistema;
  - **Gateway**: questa classe rappresenta i gateway presenti nel sistema;
  - **Device**: questa classe rappresenta i dispositivi presenti nel sistema;
  - **Sensor**: questa classe rappresenta i sensori presenti nel sistema;
  - **ViewGraph**: questa classe rappresenta i grafici presenti nel sistema;
  - **View**: questa classe rappresenta le view presenti nel sistema;
  - **Alert**: questa classe rappresenta gli alert presenti nel sistema.
- **Provider**: tutte le classi provider estendono la classe BasicProvider e possiedono un riferimento a GuzzleHttp/Client:
  - **UserServiceProvider**: questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati degli utenti. Utilizza la classe User;
  - **EntityServiceProvider**:questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati degli enti. Utilizza la classe Entity;
  - **GatewaySeviceProvider**:questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati dei gateway. Utilizza la classe Gateway;
  - **DeviceSeviceprovider**: questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati dei dispositivi. Utilizza la classe Device;
  - **SensorServiceProvider**: questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati dei sensori Utilizza la classe Sensor;
  - **ViewGraphServiceProvider**: questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati dei grafici all'interno delle view. Utilizza la classe ViewGraph;

  - **ViewServiceProvider**: questa classe viene utilizzata per effettuare le richieste HTTP alle API per ottenere i dati delle view. Utilizza la classe View.

- **Controller**: tutte le classi controller estendono la classe Controller che estende a sua volta Illuminate/Routing/Controller:
  - **UserController**: questa classe viene utilizzata per creare le viste appartenenti alla sezione di gestione degli utenti. Per fare ciò possiede un riferimento a UserServiceProvider;
  - **EntityController**: questa classe viene utilizzata per creare le viste appartenenti alla sezione di gestione degli enti. Per fare ciò possiede un riferimento alla classe EntityServiceProvider;
  - **GatewayController**: questa classe viene utilizzata per creare le viste appartenenti alla sezione di gestione dei gateway. Per fare ciò possiede un riferimento a GatewayServiceProvider;
  - **DeviceController**: questa classe viene utilizzata per creare le viste appartenenti alla sezione di gestione dei dispositivi. Ha per questo un riferimento a DeviceServiceProvider, GatewayServiceProvider e SensorServiceProvider;
  - **LoginController**: questa classe viene utilizzata per creare la vista che permette agli utenti l'autenticazione. Per fare ciò usa la classe Illuminate/Http/Request;
  - **DashborardController**: questa classe viene utilizzata per creare la dashboard dei vari utenti;
  - **SettingsController**: questa classe viene utilizzata per creare la vista di gestione delle informazioni di un utente. Usa la classe UserServiceProvider;
  - **SensorController**: questa classe viene utilizzata per creare le viste in cui vengono visualizzati i dati/grafici dei sensori. Per questo possiede un riferimento a ViewGraphServiceProvider e a SensorServiceProvider;
  - **ViewController**: questa classe viene utilizzata per creare le viste appartenenti alla sezione di gestione delle view. Per fare ciò possiede un riferimento a ViewServiceprovider e ViewGraphServiceProvider.

5.8.4 Diagrammi di sequenza

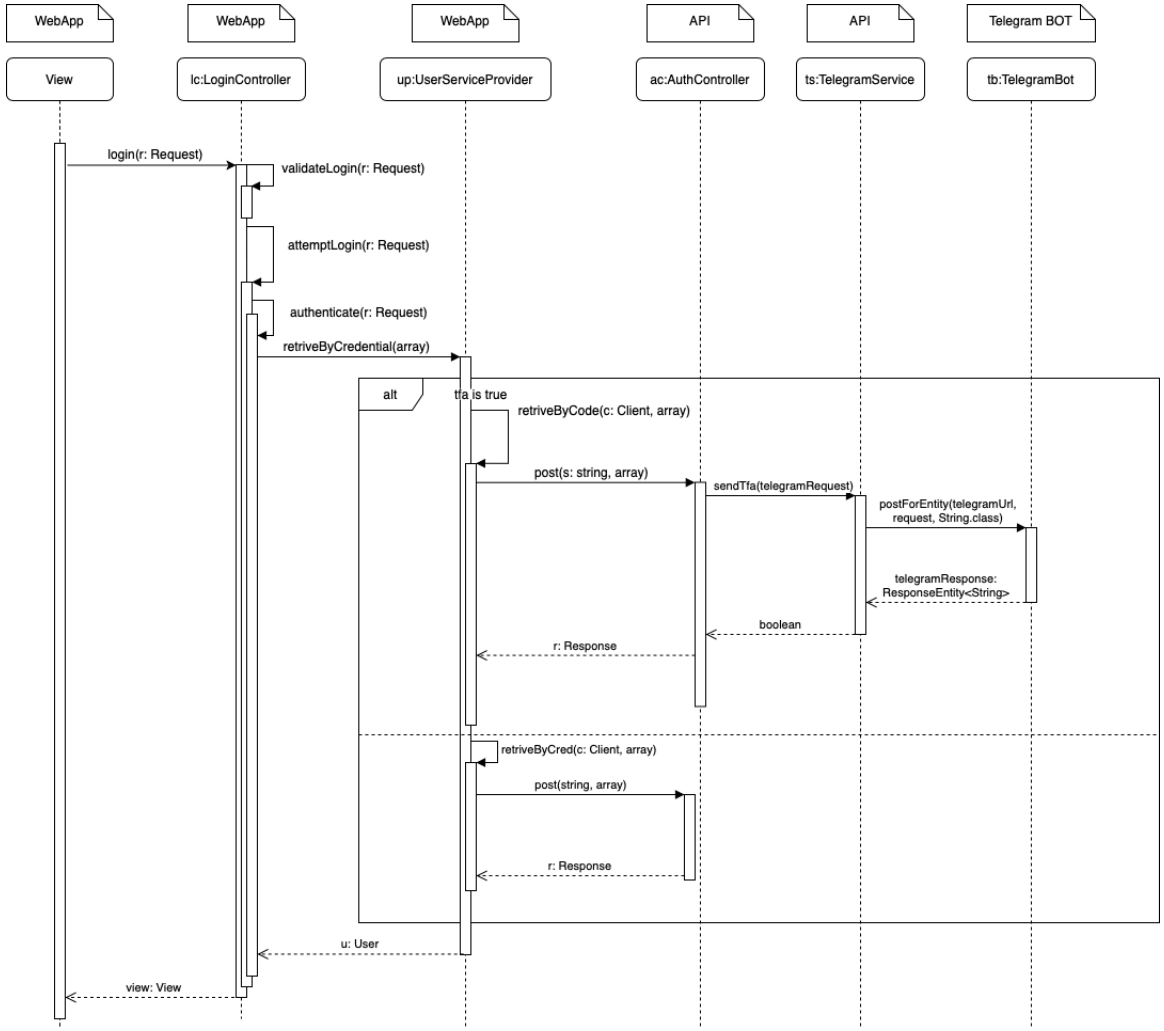

Figura 22: Diagramma di sequenza che illustra l'autenticazione all'interno della componente web app

Nel diagramma di sequenza che rappresenta l'autenticazione, alla richiesta della vista di effettuare l'autenticazione, il LoginController verifica che le credenziali siano corrette ed in seguito, se l'utente ha attivato l'autenticazione a due fattori, invia il codice tramite richiesta HTTP POST al bot di Telegram, il quale visualizza il codice nella chat dell'utente.
Mentre, se l'utente non l'ha attivata, invia semplicemente la risposta all'utente. In entrambi i casi viene mostrata la risposta affermativa o negativa all'utente e se positiva viene effettuata l'autenticazione.

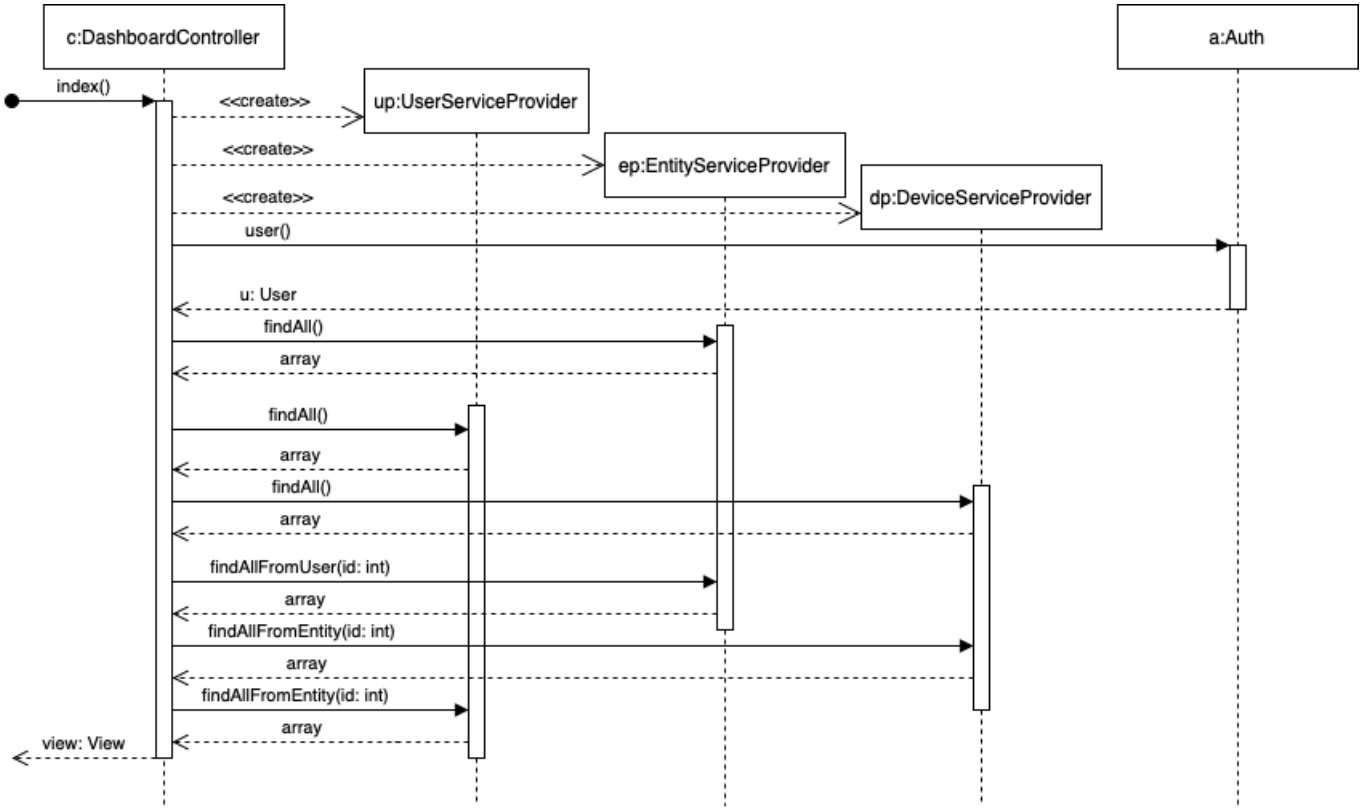

Figura 23: Diagramma di sequenza che illustra la visualizzazione della schermata dashboard all'interno della componente web app

Come si evince dal diagramma, il DashboardController crea uno UserService, un EntityServiceProvider ed un DeviceServiceProvider, i quali effettuano le richieste API$_G$ per richiedere le informazioni da visualizzare nella dashboard, quali ad esempio il numero di utenti di un certo ente o le informazioni dell'account dell'utente che sta visualizzando la dashboard. Infine tutte le informazioni ricavate vengono restituite alla pagine view.

A Lista dei bug noti

In questa sezione è riportata una *fotocopia* dei bug riscontrati a partire dall'incremento V, a seguito della realizzazione del *proof of concept*. Si riportano anche i link ai principali *Issue Tracking System* (ITS) di tutte le componenti.

- ITS, swe-gateway
- ITS, swe-kafka-db
- ITS, swe-api
- ITS, swe-webapp
- ITS, swe-telegram

Periodo	Componente	Descrizione	Stato
Incremento V	swe-gateway	La nuova configurazione ricevuta non viene applicata	risolto
Incremento V	swe-gateway	La configurazione di default non veniva applicata correttamente	risolto
Incremento V	swe-gateway	Il gateway crashava se non trovava Kafka attivo e funzionante	risolto
Incremento VI	swe-kafka-DB	Non veniva importato il file .sql per effettuare la prima configurazione all'avvio dei db	risolto
Incremento VII	swe-telegram	Problema di compatibilità tra i JSON che le API inviavano e quello che il bot si aspettava	risolto
Incremento VII	swe-telegram	Il bot non avviava correttamente il server per la ricezione delle richieste HTTP	risolto
Incremento VII	swe-telegram	Il bot non invia correttamente gli alert all'utente designato	risolto
Incremento VII	swe-webapp	Login non funzionante con TFA	risolto
Incremento VII	swe-api	Le API non inviano il codice corretto tramite telegram	risolto
Incremento VII	swe-api	Il token inviato non è valido	risolto
Incremento VII	swe-api	Le api non ritornano un token valido per Telegram	risolto
Incremento VII	swe-kafka-DB	Le tabelle del database relazionale hanno dei campi mancanti	risolto
Incremento VII	swe-kafka-DB	Il data filter non recupera gli alert positivi da mandare poi a Telegram	risolto
Incremento VIII	swe-kafka-DB	Il data collector non reperiva correttamente i dati dai topic per la creazione dei grafici	risolto

Continua nella pagina successiva...

Periodo	Componente	Descrizione	Stato
Incremento VIII	swe-api	Le api inviavano i dati dei dispositivi in un formato non conforme	risolto
Incremento VIII	swe-webapp	I grafici partono da un grafico vuoto, ma devono partire con almeno 20 dati prima	risolto
Incremento VIII	swe-webapp	Bottone eliminazione di una pagina view mancante	risolto
Incremento IX	swe-webapp	Titoli delle sezioni non omogenei	risolto
Incremento IX	swe-webapp	La pagina di dettaglio di un dispositivo va unita con la lista sensori come da AdR	risolto
Incremento IX	swe-webapp	Sistemare gli ID nelle tabelle per accedere ai dettagli	risolto
Incremento IX	swe-webapp	Le sezioni breadcrumb non rispecchiano quanto riportato nei menù	risolto
Incremento IX	swe-webapp	La distinzione tra ID reale e ID logico non è chiara	risolto
Incremento IX	swe-webapp	Visualizzazione errata della sezione dispositivi e sensori	risolto
Incremento IX	swe-webapp	Il tasto torna indietro non funziona correttamente nella navigazione delle sezioni	risolto
Incremento X	swe-webapp	Le soglie degli alert sono sbagliate (invertito maggiore e minore)	risolto
Incremento X	swe-api	La disattivazione dell'aut. TFA poteva provocare l'eliminazione della chat Telegram dal DB	risolto
Incremento X	swe-webapp	Tasti per la creazione di un alert mancanti	risolto
Incremento X	swe-webapp	Il bottone per il reset della password era presente per i mod	risolto
Incremento X	swe-webapp	Un singolo utente non può disattivare tutti gli alert del suo ente per sè stesso	risolto
Incremento X	swe-api	Un moderatore non può modificare un utente (Errore API)	risolto
Incremento X	swe-webapp	Aggiungere la possibilità di creare un grafico con un solo sensore nelle view	risolto
Incremento X	swe-api	Vengono visualizzati i logs di tutti gli utenti e non solo dell'ente a cui appartiene il mod	risolto
Incremento X	swe-webapp	Un moderatore non visualizza il bottone per aggiungere un alert	risolto
Incremento X	swe-webapp	L'ultimo invio dell'alert mostra una data sbagliata	risolto

Continua nella pagina successiva...

Periodo	Componente	Descrizione	Stato
Incremento X	swe-webapp	Manca un messaggio di successo / errore alla creazione dell'utente	risolto
Incremento X	swe-api	Il container docker delle API non esegue un build corretto	risolto
Incremento X	swe-webapp	Il container docker della webapp non trova la connessione alle API	risolto
Incremento X	swe-telegram	Il bot di telegram nel container, se stoppato, non ritorna online	risolto
Incremento XI	swe-webapp	Errore formato per l'invio della nuova configurazione di un gateway	risolto
Incremento XI	swe-gateway	Problema uso CPU da parte del gateway durante l'attesa di una nuova config	risolto
Incremento XI	swe-kafka-DB	Conflitto nel nome dei differenti topic per la ricezione delle configurazioni	risolto
Incremento XI	swe-api	Invio di JSON non conforme al gateway	risolto
Incremento XI	swe-api	Problema di connessione con Kafka durante l'invio delle configurazioni	risolto
Incremento XI	swe-api	Errori di connessione delle API se i database sono offline	risolto
Incremento XI	swe-gateway	Problema nella ricezione dei messaggi a causa della latenza	risolto
Incremento XI	swe-kafka-DB	Il data collector non filtra correttamente gli alert a causa dell'AlertTimeTable	risolto
Incremento XI	swe-kafka-DB	I messaggi non vengono inviati sul topic degli alert	risolto
Incremento XI	swe-kafka-DB	Il collegamento ai database del KafkaDataCollector non avviene correttamente dentro Docker	risolto
Incremento XI	swe-kafka-DB	Il Data collector non recupera da tutti i topic i dati di Kafka	risolto
Incremento XII	swe-webapp	Le pagine devices / create e edit hanno dei conflitti sui name degli input dei sensori inseriti	risolto
Incremento XII	swe-webapp	Manca la possibilità di visualizzare gli enti autorizzati ai un sensore e di aggiungere / rimuoverli	risolto
Incremento XII	swe-webapp	Il codice Javascript per la creazione / modifica sensori non funziona correttamente	risolto

Continua nella pagina successiva...

Periodo	Componente	Descrizione	Stato
Incremento XII	swe-webapp	Aggiungere un select per determinare se un sensore è abilitato o meno alla ricezione comandi	risolto
Incremento XII	swe-webapp	Il checkbox per abilitare l'invio dei comandi non è allineato molto bene	risolto
Incremento XII	swe-webapp	L'invio della configurazione ai gateway non funziona	risolto
Incremento XII	swe-webapp	Errore del campo telegram name dalle impostazioni utente	risolto
Incremento XII	swe-webapp	Aggiungere un background quando si aggiunge un sensore, per identificarne uno nuovo	risolto
Incremento XII	swe-webapp	Errore nella sidebar laterale che rimane aperta in modalità smartphone	risolto
Incremento XII	swe-webapp	Favicon mancante per il sito	risolto
Incremento XII	swe-webapp	La selezione di un sensore dalla pagina view e dalla pagina di creazione alert non è conforme	risolto
Incremento XII	swe-webapp	Le tabelle non sono allineate correttamente e con il giusto bordo	risolto
Incremento XII	swe-telegram	Il bot ha dei conflitti durante l'invio di un comando	risolto
Incremento XII	swe-telegram	I tasti per l'invio dei comandi non vengono formattati correttamente	risolto
Incremento XII	swe-telegram	Le emoji creano dei conflitti nei messaggi inviati all'utente	risolto
Validazione	swe-kafka-DB	La tabella alerts non viene utilizzata per il database timescale	risolto
Validazione	swe-kafka-DB	Le politiche di cancellazione a cascata di alerts, sensori e dispositivi non sono correttamente settate	risolto
Validazione	swe-webapp	L'indirizzo IP riportato nella webapp non è corretto	risolto
Validazione	swe-webapp	La email di supporto riportata è solo un placeholder, va modificata con quella del gruppo	risolto
Validazione	swe-webapp	Le view non funzionano se inserisco uno spazio	risolto
Validazione	swe-webapp	Un admin deve poter ripristinare un utente	risolto
Validazione	swe-webapp	Un admin non deve poter disattivare e ripristinare altri admin	risolto

Continua nella pagina successiva...

Periodo	Componente	Descrizione	Stato
Validazione	swe-webapp	I form non contengono l'attributo required in html5	risolto
Validazione	swe-webapp	La cancellazione alert come admin manca, insieme al relativo messaggio di successo o errore	risolto
Validazione	swe-webapp	I bottoni per la rimozione dei sensori hanno btn-small invece di btn-sm di Bootstrap	risolto
Validazione	swe-webapp	I grafici non vengono mostrati su safari 13	risolto
Validazione	swe-webapp	Il footer di modifica / aggiunta ente manca probabilmente di un div nel contenitore	risolto
Validazione	swe-webapp	Il TFA non viene disabilitato correttamente nelle impostazioni	risolto
Validazione	swe-webapp	Manca l'hash in sha512 alle password	risolto
Validazione	swe-webapp	La password non viene cambiata correttamente nelle impostazioni	risolto
Validazione	swe-webapp	Se un admin modifica email o password di un utente che è attualmente loggato nel sito, quest'ultimo riceverà un errore 403 e non potrà fare logout o login.	risolto
Validazione	swe-webapp	Il menù in mobile ha un glitch per cui scrollando in basso si chiude in automatico il menù	risolto
Validazione	swe-webapp	L'aggiunta di un sensore a un ente non funziona correttamente	risolto
Validazione	swe-webapp	Un utente non può essere riassegnato a un altro ente	risolto
Validazione	swe-webapp	Il fuso orario riportato nella webapp è sbagliato	risolto
Validazione	swe-webapp	La lista utenti dei membri di un ente è errata	risolto
Validazione	swe-webapp	L'aggiunta di un device senza sensori non funziona correttamente	risolto
Validazione	swe-webapp	L'aggiunta di un device con un gateway e id reale già presenti non funziona correttamente	risolto
Validazione	swe-api	L'invio di più comandi contemporaneamente tramite Telegram non ha sempre successo	risolto
Validazione	swe-telegram	Il comando di richiesta di invio dei comandi su telegram ritorna una risposta errata	risolto
Validazione	swe-api	La disattivazione dell'aut. TFA senza l'eliminazione dell'username Telegram provoca un errore	risolto

Continua nella pagina successiva...

Periodo	Componente	Descrizione	Stato
Validazione	swe-api	La modifica del nome di un ente provoca un errore	risolto
Validazione	swe-telegram	Quando viene effettuata la richiesta contemporanea della lista di dispositivi con autorizzazioni differenti questo provoca un errore richiedendo all'utente di rieffettuare manualmente login	risolto
Validazione	swe-telegram	Il riavvio del sistema richiede una nuova autenticazione agli utenti	risolto
Validazione	swe-webapp	Se si esegue la rimozione di un sensore da un ente, gli alert di quel sensore inseriti dall'ente rimangono	risolto

Tabella 2: Tabella contenente un riepilogo dei bug riscontrati a partire dall'incremento V

B Glossario

A

Alert

Segnale generato per comunicare un messaggio, in cui la soglia di una rilevazione è stata superata.

API

Le API (acronimo di Application Programming Interface, ovvero Interfaccia di programmazione delle applicazioni) sono set di definizioni e protocolli con i quali vengono realizzati e integrati software applicativi.

B

Bootstrap

Bootstrap è una raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web. Essa contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript.

Bot, telegram

I bot sono applicazioni di terze parti sviluppati da programmatori esterni per interagire con gli utenti tramite messaggi, comandi e richieste in linea tramite il servizio di messaggistica Telegram.

C

Chrome, Google

Browser web sviluppato da Google.

Container, Docker

Un container è una singola unità di software che contiene sia il codice che tutte le sue dipendenze in modo tale da poter essere eseguito velocemente in qualsiasi ambiente.

D

Docker

Il software Docker è una tecnologia di containerizzazione che consente la creazione e l'utilizzo dei container Linux.

Dockerfile

Un *Dockerfile* è un file di configurazione che illustra i passaggi che devono essere realizzati per compilare un'immagine di un sistema operativo come base per eseguire un certo applicativo.

Docker-compose

Il docker-compose è un comando che può essere utilizzato per comporre, letteralmente, tutta l'infrastruttura in base ai servizi che si è scelto di integrare. Fa uso dei *Dockerfile* per avviare l'eventuale build delle *immagini Docker* e permette con un solo comando di attivare (`docker-compose up -d`) o disattivare (`docker-compose down`) tutti i servizi. L'intera configurazione è salvata su un file denominato per convenzione `docker-compose.yml`.

E

Edge, Microsoft

Browser web sviluppato da Microsoft.

Ente

Con ente si intende una azienda o associazione a cui viene assegnato il monitoraggio di alcuni dispositivi.

F

Firefox, Mozilla

Browser web libero e multipiattaforma mantenuto da Mozilla Foundation.

Framework

Un framework, è un'architettura logica di supporto (spesso un'implementazione logica di un particolare design pattern) su cui un software può essere progettato e realizzato.

I

Immagine, Docker

Un'immagine Docker è un eseguibile che può essere istanziato in un container. L'immagine Docker viene prodotta a seguito della compilazione tramite il comando `docker build` di un *Dockerfile* o di un *docker-compose*.

J

Java

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

JavaScript

JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione web lato client per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

JSON

Formato utilizzato per lo scambio di dati all'interno della componenti dell'applicazione.

JUnit

Framework di unit testing per il linguaggio di programmazione Java.

K

Kafka, Apache

Apache Kafka è una piattaforma open source di stream processing scritta in Java e Scala e sviluppata dall'Apache Software Foundation. Questo progetto viene usato principalmente per tutte le applicazioni di elaborazioni di stream di dati in tempo reale.

L

Laravel, framework

Laravel è un framework open-source molto potente e usato per sviluppare applicazioni in PHP, facendo uso nativamente di BOOTSTRAP$_G$ e Vue.js. Permette di realizzare la parte back-end di un sito web seguendo il modello model-view-controller.

N

Node Package Manager (NPM)

NPM (abbreviazione di Node Package Manager) è un gestore di pacchetti per il linguaggio di programmazione JavaScript. È il gestore di pacchetti predefinito per l'ambiente di runtime JavaScript Node.js.

Node.js

Node.js è una runtime di JavaScript Open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript.

P

PostgreSQL

PostgreSQL è un completo DBMS ad oggetti rilasciato con licenza libera.

S

Safari

Browser web sviluppato da Apple Inc.

Spring, framework

Spring è un framework Java open-source che viene utilizzato per lo sviluppo di applicativi web e di servizi API.

T

Telegram

Telegram è un servizio di messaggistica istantanea e broadcasting basato su cloud.

TimeScale, DB

TimeScaleDB è un database open-source costruito per analizzare dati di serie storiche.

Timeseries, DB

Con il termine timeseries DB si intende un database il cui scopo è la memorizzazione di serie storiche di dati.

V

Vue.js

Vue.js è un framework open-source di Javascript che permette di realizzare la parte front-end dinamica in unione con Laravel. Il framework è molto flessibile e altamente scalabile, nonché compatibile anche con BOOTSTRAP$_G$.

W

Web, app, application

Con web app o applicazione web, si intende una applicazione fruibile via web per mezzo di un browser.