DStack

dstackgroup@gmail.com

Manuale per lo sviluppatore

Informazioni sul documento

Nome documento	Manuale per lo sviluppatore
Versione	v1.0.0
Data approvazione	2019-05-08
Responsabili	Niccolò Vettorello
Redattori	Alberto Schiabel Federico Rispo
Verificatori	Elton Stafa
Stato	Redazione
Lista distribuzione	Gruppo *DStack* *Prof. Tullio Vardanega* *Prof. Riccardo Cardin* *Imola Informatica S.P.A.*
Uso	Esterno

Sommario

Il presente documento ha lo scopo di presentare le tecnologie e l'architettura del sistema agli sviluppatori interessati al software Butterfly.

Diario delle Modifiche

Versione	Descrizione	Nominativo	Ruolo	Data
v1.0.0	**Approvazione per il rilascio in RA**	Niccolò Vettorello	*Responsabile di Progetto*	2019-05-08
v0.2.0	**Verifica finale**	Elton Stafa	*Verificatore*	2019-05-08
v0.1.4	**Incremento §?? e §10**	Federico Rispo	*Verificatore*	2019-05-02
v0.1.3	**Incremento §5 §?? e §9**	Alberto Schiabel	*Progettista*	2019-04-30
v0.1.2	**Incremento sezioni §6, §7**	Alberto Schiabel	*Verificatore*	2019-04-27
v0.1.1	**Aggiunta sezione D e §??**	Federico Rispo	*Progettista*	2019-04-27
v0.1.0	**Verifica per rilascio in RQ**	Enrico Trinco	*Verificatore*	2019-04-08
v0.0.4	**Incremento §5 e §??**	Alberto Schiabel	*Progettista*	2019-04-07
v0.0.3	**Incremento §15 e appendici §A e §B**	Alberto Schiabel	*Progettista*	2019-04-05
v0.0.2	**Incremento §2, 3**	Alberto Schiabel	*Progettista*	2019-04-03
v0.0.1	**Creazione scheletro del documento**	Enrico Trinco	*Responsabile di Progetto*	2019-03-25

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Obiettivo

Lo scopo del *Manuale per lo sviluppatore* è presentare l'architettura a microservizi di cui il prodotto Butterfly è composto, l'organizzazione del codice sorgente e soprattutto dare informazioni utili al mantenimento e all'estensione del progetto. Questo documento ha inoltre il fine di fillustrare le procedure di installazione e di sviluppo locale, di citare i framework e le librerie di terze parti coinvolte, e di presentare la struttura del progetto a livelli progessivi di dettaglio, grazie ai diagrammi di package, di classe e di sequenza.

1.2 Scopo del Prodotto

Butterfly nasce dall'esigenza di uniformare e accentrare la gestione delle segnalazioni generate a partire da sistemi di terze parti, quali Redmine, GitLab e SonarQube. Questi strumenti sono parte integrante dei processi gestionali, di versionamento e di Continuous Integration dell'azienda committente. La maggior parte di essi fornisce già dei meccanismi di notifica ed inoltro delle possibili segnalazioni, sono configurabili e accessibili da dashboard molto diverse tra loro, di difficile interazione e anche con limitazione di accessibilità. Inoltre, in caso di segnalazioni di bug in ambienti di produzione è fondamentale assicurarsi che gli sviluppatori in grado di risolvere il problema siano segnalati tempestivamente, senza aspettare che loro accedano a qualche dashboard specifica. Il gruppo *DStack* si propone quindi di sviluppare una rete di soluzioni che offrano un'interfaccia condivisa, estendibile per gestire le segnalazioni relative alla pipeline di sviluppo software di *Imola Informatica S.P.A.*. Questa interfaccia deve inoltre permettere una configurazione automatica e personalizzabile di tali segnalazioni.

1.3 Prerequisiti di Conoscenza

Per poter comprendere al meglio il contenuto di questo documento, facendo in particolar modo riferimento ai diagrammi presentati nelle sezioni seguenti, è opportuno che il lettore abbia almeno un'infarinatura generale del linguaggio UML2, del concetto di API REST, e di che cosa sia un Message Broker.

1.4 Elenco definizioni utili

Nella parte conclusiva di questo documento viene riportato un breve elenco di vocaboli, corredati dal rispettivo significato, con l'intento di agevolare il lettore nella piena comprensione del testo. Considerata l'utenza finale di Butterfly, il gruppo ha deciso di omettere quei termini che sono noti a chiunque possegga conoscenze basilari di informatica. La prima occorrenza di tali termini all'interno del documento è caratterizzata da una piccola `'G'` posta a pedice.

1.5 Nota sulle versioni

Il gruppo garantisce il funzionamento dei servizi che compongono Butterfly solo nel caso in cui vengano usate le versioni esatte delle librerie e dei framework di sviluppo che saranno elencati nel resto del documento. Nonostante sia sufficientemente

probabile che le successive versioni degli strumenti utilizzati mantengano la retrocompatibilità con le versioni utilizzate attualmente, ciò non può essere garantito con certezza. Il gruppo non si assume quindi nessuna responsabilità nel caso in cui il prodotto risenta, del tutto o in parte, di problematiche riconducibili ad una dipendenza di terze parti la cui versione non è supportata.

1.6 Riferimenti informativi

- «The Twelve-Factor App»:
  https://12factor.net
  - "Introduction", come riferimento per la definizione generale di cosa è un'app a microservizi;
  - "The Twelve Factors", come riferimento alla definizione delle stesse.
- Design Pattern Comportamentali
  https://www.math.unipd.it/~tullio/IS-1/2018/Dispense/E08.pdf
  - Template Method Pattern
  - Observer Pattern
  - Command Pattern
  - Strategy Pattern

2 Requisiti minimi di sistema

Presentiamo di seguito i requisiti minimi richieste per l'installazione del sistema Butterfly, suddivisi in *obbligatori* e *opzionali*. Con requisiti *obbligatori*, in questo caso intendiamo gli strumenti indispensabili sia per lo sviluppo che per l'installazione del prodotto. Invece, con requisiti *opzionali*, in questo caso intendiamo gli strumenti utili allo sviluppo e alla compilazione del codice in locale, ma non richiesti in fase d'installazione.

- Requisiti obbligatori
  - Docker versione 18.09.2: utilizzato per eseguire in un ambiente virtualizzato e controllato i processi di compilazione, analisi statica, analisi dinamica ed esecuzione;
  - Docker Compose versione 1.23.2: utilizzato per gestire la comunicazione tra i servizi istanziati tramite *Docker*;
  - Connessione ad internet: necessaria per scaricare le immagini su cui si basano i servizi virtualizzati da Docker, e le dipendenze di terze parti di Java e Node.js.
- Requisiti opzionali
  - Java versione 11[1] (LTS), nella variante *OpenJDK*: linguaggio utilizzato per scrivere tutti i microservizi che interagiscono direttamente col Message Broker Apache Kafka;
  - Maven versione 3.5.2[2]: è il package manager più famoso per Java;
  - Node.js versione 10.15.3[3] (LTS): framework per l'esecuzione di codice JavaScript in ambienti diversi dal browser, sul quale poggia il microservizio Gestore Personale;
  - Yarn versione 1.15.2[4]: analogamente a Maven, è uno dei package manager più utilizzati per gestire le dipendenze e le fasi di sviluppo in ambiente Node.js;

[1] Link per scaricare la versione richiesta di Java: `http://download.oracle.com/otn-pub/java/jdk/11.0.2+9/f51449fcd52f4d52b93a989c5c56ed3c/jdk-11.0.2_linux-x64_bin.tar.gz`
[2] Link per scaricare la versione richiesta di Maven: `http://mirror.nohup.it/apache/maven/maven-3/3.5.2/binaries/apache-maven-3.5.2-bin.tar.gz`
[3] Link per scaricare la versione richiesta di Node.js: `https://nodejs.org/dist/v10.15.3/node-v10.15.3-linux-x64.tar.xz`
[4] Link per scaricare Yarn: `https://yarnpkg.com/latest.msi`

3 Tecnologie e librerie di terze parti interessate

Di seguito vengono brevemente descritte le tecnologie e le librerie di terze parti adottate per il progetto Butterfly.

3.1 Tecnologie

3.1.1 Tecnologie condivise tra i servizi

3.1.1.1 JSON

JSON è il formato di serializzazione testuale scelto per l'interfacciamento con le REST API esposte dal microservizio chiamato *Gestore Personale*. Esso è un formato estremamente diffuso per lo scambio dei dati in applicazioni client-server. Si basa su oggetti, ovvero coppie chiave/valore, e supporta i tipi booleano, stringa, numero, e lista. È semplice e leggibile ad occhio umano, inoltre non necessita di alcun processo di compilazione particolare per essere modificato. Node.js supporta nativamente il formato JSON (un JSON valido è anche un oggetto JavaScript valido), mentre Java richiede l'utilizzo di particolari librerie per leggere e modificare oggetti JSON.

3.1.2 Tecnologie per i Producer e Consumer

3.1.2.1 Java

Linguaggio di programmazione orientato agli oggetti a tipizzazione statica, progettato per essere il più possibile indipendente dalla piattaforma di esecuzione. Nell'ambito di Butterfly, il gruppo l'ha utilizzato per descrivere i Producer e Consumer che comunicano con Apache Kafka, il cui principale client ufficiale è stato scritto appositamente per linguaggi della JVM come Java.

3.1.2.2 Maven

Strumento che ricopre il ruolo di gestore delle dipendenze e delle fasi di sviluppo di un progetto Java. Le dipendenze sono specificate nel file di configurazione chiamato `pom.xml`. Ogni dipendenza richiesta è scaricata automaticamente da Maven e salvata localmente.

3.1.2.3 Apache Avro

Avro è il sistema per la serializzazione dei dati che abbiamo deciso di impiegare per serializzare i messaggi scambiati tra i Producer e i Consumer di Butterfly. Esso utilizza schemi JSON per la definizione dei valori e dei tipi, permette l'autogenerazione di classi Java per la creazione e la lettura di oggetti la cui struttura è definita secondo lo schema di Avro, ed è un sistema di serializzazione binario. Esso è stato scelto sia perché è uno dei formati open source più diffusi all'interno della community di sviluppatori di Apache Kafka, sia perché è l'unico formato supportato da Confluent Schema Registry.

3.1.2.4 Confluent Schema Registry

Sistema che permette di gestire in maniera retrocompatibile la comunicazione tra Producer e Consumer che usano lo stesso record Avro per comunicare tra loro. Confluent Schema Registry si assicura che la comunicazione avvenga correttamente anche se i Producer e Consumer coinvolti utilizzano versioni disallineate dello stesso schema Avro.

3.1.2.5 Apache Kafka
Sistema di messaggistica open source distribuito che consente di creare applicazioni in tempo reale tramite flussi di dati. Il progetto, inizialmente scritto in Scala, passato poi a Java, è sviluppato dalla Apache Software Foundation e mira a fornire un sistema di gestione dei feed G di dati altamente scalabile con caratteristiche quali performance elevate e bassa latenza. Kafka è uno dei più famosi Message Broker open source, rappresentato da una coda di messaggi con capacità di scalabilità estreme, di tipo publish/subscribe, la cui architettura segue i principi di un transaction log distribuito. Il sistema poggia su un meccanismo di scalabilità indipendente offerto dal servizio di Apache Zookeeper.

3.1.3 Tecnologie per lo user manager

3.1.3.1 Node.js
Piattaforma ad eventi runtime open source per l'esecuzione di codice JavaScript a lato server. Grazie all'architettura event-driven, permette di gestire I/O asincroni, ottimizzando così la scalabilità e la capacità di trasmissione effettiva utilizzata da un canale di comunicazione.

3.1.3.2 Typescript
Linguaggio di programmazione open source sviluppato da Microsoft. Estende il linguaggio di programmazione JavaScript aggiungendo il supporto alle interfacce, ai tipi e alle classi, e promuove una serie di pratiche che rendono molto più facile ragionare sul codice scritto rispetto a "vanilla" JavaScript.

3.1.3.3 PostreSQL
Potente database relazionale open source. È noto per essere stato tra i primi database relazionali ad aver supportato nativamente il tipo JSON nei costrutti SQL.

3.1.3.4 HTTP REST
Protocollo di comunicazione stateless basato sul concetto di locazione delle risorse, alle quali è possibile accedere tramite un identificatore globale unico attraverso richieste GET, POST, PUT e DELETE.

3.2 Librerie di terze parti

3.2.1 Librerie producer e consumer

3.2.1.1 Plugin Avro
Libreria per il supporto di Avro.

- **Link repository Maven:** https://mvnrepository.com/artifact/org.apache.avro/avro/1.8.2
- **Versione utilizzata:** 1.8.2

3.2.1.2 Spark Java
Framework open source per una gestione veloce ed efficace delle chiamate HTTP. Permette l'esposizione degli endpoint HTTP necessari ai servizi di terze parti e per inoltrare gli eventi tramite webhook.

- **Link repository Maven**: https://mvnrepository.com/artifact/com.sparkjava/spark-core/2.7.2
- **Versione utilizzata**: 2.7.2

3.2.1.3 Client Kafka

Libreria ufficiale per l'interfacciamento ad Apache Kafka tramite linguaggi della JVM, in particolare Java e Scala.

- **Link repository Maven**: https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients/2.1.0
- **Versione utilizzata**: 2.1.0

3.2.1.4 GSON

Libreria fornita da Google utilizzata dal middleware dispatcher per trasformare gli oggetti ricevuti dal consumer da una stringa JSON ed assegnarlo ad un oggetto Java. Permette anche la conversione opposta: da oggetto Java a stringa JSON.

- **Link repository Maven**: https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.5
- **Versione utilizzata**: 2.8.5

3.2.1.5 gitlab4j

Libreria che mette a disposizione API Java per interagire con repositories GitLab sfruttando le *GitLab REST API*. Questa libreria fornisce inoltre pieno supporto per i *GitLab Webhooks* ed i *GitLab System hooks*.

- **Link repository Maven**: https://mvnrepository.com/artifact/org.gitlab4j/gitlab4j-api/4.9.14
- **Versione utilizzata**: 4.9.14

3.2.1.6 Telegram Bots

Libreria che permette di creare e gestire Bot di Telegram utilizzando il linguaggio Java.

- **Link repository Maven**: https://mvnrepository.com/artifact/org.telegram/telegrambots/4.1.2
- **Versione utilizzata**: 4.1.2

3.2.1.7 Simple Slack API

Libreria che permette la creazione e la gestione di Bot per Telegram utilizzando il linguaggio Java.

- **Link repository Maven**: https://mvnrepository.com/artifact/com.github.Ullink/simple-slack-api/1.2.0
- **Versione utilizzata**: 1.2.0

3.2.1.8 slf4j

La libreria **S**imple **L**ogging **F**acade for Java (SLF4J) fornisce un'astrazione per interagire facilmente con vari frameworks di log per Java.

- **Link repository Maven**: https://mvnrepository.com/artifact/org.slf4j/slf4j-api/1.7.25
- **Versione utilizzata**: 1.7.25

3.2.2 Librerie User Manager

3.2.2.1 Pino

Pino è attualmente la libreria che offre funzionalità di logging per Node.js ad avere le prestazioni più elevate. Il log viene emesso a *stdout* in formato JSON e ha un impatto minimale sulle prestazioni del sistema in esecuzione.

- **Link repository npm**: https://www.npmjs.com/package/pino
- **Versione utilizzata**: 5.12.2

3.2.2.2 Joi

Libreria per validare i campi degli oggetti JSON, permette di definirne la struttura attesa, il tipo e gli eventuali vincoli di validazione.

- **Link repository npm**: https://www.npmjs.com/package/joi
- **Versione utilizzata**: 14.3.1

3.2.2.3 Koa

Microframework HTTP per la creazione di API REST in Node.js. Esso è il successore del più famoso framework *Express*, del quale ne condivide i creatori originali, ma è molto più leggero, performante, e offre nativamente un'interfaccia asincrona tramite Promise JavaScript.

- **Link repository npm**: https://www.npmjs.com/package/koa
- **Versione utilizzata**: 2.7.0

3.2.2.4 pg-promise

Client Node.js per la connessione e l'interazione col database PostgreSQL. Esso offre un'interfaccia asincrona tramite Promise JavaScript, e consente di dichiarare le query da inviare al database in file ad estensione `".sql"` provvisti di parametri nominali sostituiti a tempo d'esecuzione.

- **Link repository npm**: https://www.npmjs.com/package/pg-promise
- **Versione utilizzata**: 8.6.4

3.2.2.5 @slack/bolt

Client ufficiale TypeScript per la connessione ad applicazioni create sulla piattaforma Slack. Esso è stato impiegato per il servizio che gestisce la configurazione degli account di contatto di Slack, Slack Account Configurator.

- **Link repository npm**: https://www.npmjs.com/package/@slack/bolt
- **Versione utilizzata**: 1.0.1

3.2.3 Librerie per i test

3.2.3.1 JUnit

Framework di test dedicato per il linguaggio Java che permette la scrittura di test ripetibili in modo organizzato e semplice.

- **Link repository Maven**: https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine/5.3.1
- **Versione utilizzata**: 5.3.1

3.2.3.2 Jest

Framework di test open source sviluppato da Facebook e viene utilizzato per testare codice scritto in JavaScript. Inizialmente usato per il testing su React, ora sta diventando sempre di più il punto di riferimento per i test.

- **Link repository nom**: https://www.npmjs.com/package/jest
- **Versione utilizzata**: 24.7.1

3.2.3.3 supertest

Descrizione

- **Link repository npm**: https://www.npmjs.com/package/supertest
- **Versione utilizzata**: 4.0.2

4 Procedura d'installazione

4.1 Configurazione ambiente di lavoro iniziale

4.1.1 Configurazione ambiente per lo sviluppo

In modo da avere un ambiente di lavoro già configurato e pronto per lo sviluppo è necessario possedere queste quattro tecnologie:

- Java;
- Maven;
- Node.js;
- Yarn;

4.1.2 Installazione Java

Nel caso cui Java non sia presente nel sistema operativo oppure la versione è inferiore a quella richiesta, puoi scaricare Java da questo link. Successivamente è necessario impostare le variabili d'ambiente JAVA_HOME e PATH.

4.1.2.1 Windows: su sistemi operativi Windows è necessario eseguire i seguenti step:

- Estrarre la cartella nella posizione desiderata;
- Accedere alle impostazioni sulle variabili d'ambiente;
- Creare la nuova variabile di sistema JAVA_HOME, dove il "Valore variabile" corrisponde al percorso della directory di Java;
- Aggiungere alla variabile di sistema Path il seguente valore: `%JAVA_HOME%\bin`.
- Confermare le operazioni.

4.1.2.2 Linux: sui sistemi operativi Linux è necessario seguire i seguenti step:

- Spostare il file scaricato nella cartella sulla quale si vuole installare Java;
- Eseguire il comando `tar -zxf "jdk-ver_linux-x_bin.tar.gz"`;
- Eseguire il comando `nano /etc/environment`;
- Inserire `JAVA_HOME="percorso_della_cartella"` e successivamente salvare;
- Eseguire il comando `source /etc/enviroment`;

Per verificare l'esito dell'installazione, basta digitare sul proprio terminale il comando `java -version`. L'esito dell'installazione è positivo solamente se viene riportata la versione scaricata di java.

4.1.3 Installazione Maven

Nel caso in cui Maven non sia installato o la versione presente sia di una versione precedente a quella richiesta, puoi scaricare Maven da questo link.
Prerequisito per l'installazione corretta di Maven è la corretta installazione di *Java* e la variabile d'ambiente *JAVA_HOME*.

4.1.3.1 Windows:

- Estrarre la cartella nella cartella desiderata;
- Accedere alle impostazioni sulle variabili d'ambiente;
- Creare la variabile di sistema *M2_HOME*, dove il "Valore variabile" corrisponde al percorso della directory di Maven;
- Aggiungere alla variabile di sistema Path il seguente valore: `%M2_HOME%\bin`.
- Confermare le operazioni.

4.1.3.2 Linux: per i sistemi operativi Linux bisogna seguire i seguenti passaggi:

- Spostare il file scaricato nella cartella sulla quale si vuole installare Maven;
- Eseguire il comando `tar xzvf "apache-maven-ver_linux-x_bin.tar.gz"`;
- Aggiungere al PATH la directory bin attraverso il comando `export PATH="percorso_cartella_maven/bin:$PATH"`.

Per verificare l'esito dell'installazione, basta digitare sul proprio terminale il comando `mvn -version`. L'esito dell'installazione è positivo solamente se:

- La versione di Maven corrisponde a quella scaricata;
- La versione di Java riportata corrisponde a quella precedentemente scaricata.

4.1.4 Installazione Node.js

Nel caso Node.js non soddisfi la versione richiesta oppure non è presente nel sistema puoi fare riferimento a questi step sul come installarlo correttamente.

4.1.4.1 Windows: Scaricare l'ultima versione $LTS_{\mathbf{G}}$ attraverso questo link.

4.1.4.2 Linux

- Da terminale, eseguire i seguenti comandi
  1. `curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -`;
  2. `sudo apt-get install -y nodejs`;
  3. `sudo apt-get install -y build-essential`;

4.1.5 Installazione di Yarn

Nel caso cui Yarn non soddisfi la versione richiesta oppure non è presente nel sistema, puoi fare riferimento alla guida ufficiale presente a questo link. Una volta cliccatoci, il sito ti fornirà la guida compatibile col tuo sistema operativo.

4.1.6 Installazione Git Bash (solo per Windows)

Nel caso in cui lo sviluppo e la manutenzione di Butterfly da sistema operativo Microsoft Windows 10, è necessario installare **Git Bash**. Le probabilità che quest'applicazione sia già installata nel computer di uno sviluppatore sono elevate, in quanto essa viene distribuita assieme al client ufficiale *Git*$_{\mathbf{G}}$. Git Bash permette di eseguire comandi della bash shell di Linux anche in ambiente Windows, permettendo quindi di poter eseguire gli script di utilità `*.sh` in modalità multipiattaforma, senza richiedere quindi una riscrittura in *Batch*$_{\mathbf{G}}$ o in *Powershell*$_{\mathbf{G}}$. È possibile scaricare la Git Bash tramite questo link. Non sono stati testati sistemi Windows precedenti alla versione 10 del sistema operativo, quindi possiamo garantire che siano adatti allo sviluppo di Butterfly.

4.2 Configurazione ambiente per l'esecuzione

4.2.1 Installazione Docker

Nel caso in cui Docker non soddisfi la versione richiesta oppure non è presente nel sistema, puoi fare riferimento a questi link per scaricare Docker nel tuo sistema operativo.

4.2.1.1 Windows: https://runnable.com/docker/install-docker-on-windows-10

4.2.1.2 Linux: https://runnable.com/docker/install-docker-on-linux

4.2.2 Installazione Docker Compose

Nel caso in cui Docker non soddisfi la versione richiesta oppure non è presente nel sistema, puoi fare riferimento a questi link per scaricare Docker nel tuo sistema operativo.
Prerequisito fondamentale è la presenza di **Docker** già installato sul proprio dispositivo. Se non è ancora installato fare riferimento alla sotto sezione precendente.

4.2.2.1 Windows: https://docs.docker.com/compose/install/

4.2.2.2 Linux: https://linuxize.com/post/how-to-install-and-use-docker-compose-on-ubuntu-18-04/

4.3 Installazione Butterfly ed Avvio

Di seguito sono descritti gli step da eseguire in ordine per poter installare l'intero sistema Butterfly.

4.3.1 Download Codice Sorgente

Il codice sorgente di Butterfly è disponibile al seguente indirizzo pubblico: https://github.com/dstack-group/Butterfly. Prima di tutto è necessario scaricare tale repository da *GitHub* e posizionarla in una cartella del proprio sistema. Utilizzare *git*$_{\mathbf{G}}$ per effettuare l'operazione di *clone* delle directory non è uno step necessario, tuttavia disporre di un sistema di versionamento distribuito come *git* è fortemente

consigliato per lo sviluppo e il mantenimento di Butterfly. In alternativa all'utilizzo di git, è possibile scaricare un archivio zip dei sorgenti al seguente indirizzo: https://github.com/dstack-group/Butterfly/archive/master.zip. In tutte le seguenti sottosezioni assumeremo che la directory corrente sia quella della repository appena scaricata.

4.3.2 Installazione Servizi di Terze Parti

All'interno della cartella "third-party-services" sono contenute le immagini Docker necessarie all'avvio delle applicazioni di produzione di terze parti:

- **Redmine**, gestita dallo script "./redmine.sh";
- **GitLab**, gestita dallo script "./gitlab.sh";
- **SonarQube**, gestita dallo script "./sonarqube.sh";

Ognuno di questi servizi è eseguito tramite lo strumento di virtualizzazione *Docker Compose*$_{\mathbf{G}}$ ed è accessibile tramite interfaccia web. Inoltre, ogni script sopracitato dispone dei seguenti comandi:

- **start**: avvia il servizio;
- **stop**: interrompe l'esecuzione del servizio;
- **reset** interrompe il servizio, ne elimina i dati salvati e lo riavvia:
- **logs** mostra a terminale i log di *Docker Compose*$_{\mathbf{G}}$ relativi al servizio in esecuzione:

Ad esempio, per avviare Redmine, è necessario posizionarsi nella cartella "third-party-services" e lanciare il comando `./redmine.sh start`, mentre per resettare GitLab è sufficiente `./gitlab.sh reset`.

Chiaramente, questi servizi sono utili solo a scopo di test. Per un utilizzo professionale, consigliamo l'utilizzo di server dedicati per l'esecuzione dei processi di gestione progetto, analisi statica e versionamento del codice.

4.3.3 Compilazione servizi Java

Per mandare in esecuzione il sistema esistono essenzialmente due modi[5], qui illustrati:

- `./run.sh --build dev`: lancia tutti i servizi offerti da *Butterfly*, compresi **pgAdmin**, che permette di visionare lo stato corrente del DB, e **OpenAPI**, per la documentazione delle REST API;
- `./run.sh --build prod`: manda in esecuzione l'effettivo ambiente di produzione.

Per informazioni più dettagliate si fa riferimento al capitolo n. 11.

[5]Si consiglia inoltre di utilizzare il comando `./run.sh -build test` per verificare che l'ambiente di esecuzione sia conforme

5 Architettura

5.1 Descrizione

Di seguito è presentata l'architettura ad alto livello del sistema Butterfly. Il prodotto è costituito da una rete di microservizi la cui interazione con l'utente avviene tramite API REST, e la cui comunicazione interna al sistema si basa sul Messaging Pattern 'Publish/Subscribe' nella sua accezione topic-based, in cui il message broker intermediario è Apache Kafka. Agli utenti che desiderano interagire con il sistema, lo *User Manager* espone le operazioni CRUD tramite API REST. Per operazioni *CRUD* intendiamo metodi di creazione (in inglese ***C**reate*), lettura (***R**ead*), modifica (***u**pdate*) e cancellazione (***D**elete*) che coinvolgono le entità modellate dal servizio **Gestore Personale**, che nel codice (scritto in lingua inglese) è stato chiamato **User Manager**. Nel resto del documento utilizzeremo indistintamente entrambe le nomenclature per identificare il medesimo servizio.
Come richiesto dal capitolato, Butterfly aderisce ai principi delle 12 Factor Apps.

5.2 Visione generale

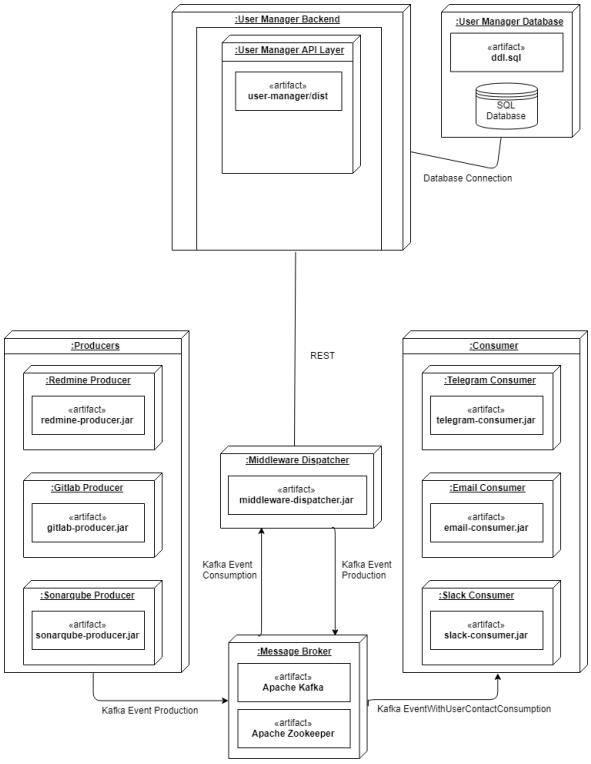

Figura 1: Butterfly:Deployment Diagram

Come si può notare nella Figura 2, ogni microservizio rappresenta un artefatto a sé, che può potenzialmente essere messo in esecuzione in nodi separati connessi tra loro. Per quanto riguarda i microservizi realizzati in Java, ogni file `jar` contiene al suo interno tutte le dipendenze di cui necessita a runtime: non è quindi richiesta alcuna altra libreria di dipenza in formato jar. Poiché i ruoli svolti dai singoli servizi Producer sono fondamentalmente gli stessi, e analogamente i ruoli svolti dai singoli servizi Consumer sono molto simili, abbiamo deciso di creare dei piccoli mo-

duli che definiscono delle interfacce e delle classi astratte comuni. A causa di ciò, l'interfaccia pubblica del servizio **"gitlab-producer"** è identica a quella dei servizi **"redmine-producer"** e **"sonarqube-producer"**, così come l'interfaccia pubblica di **"telegram-consumer"** è molto simile a quella di **"email-consumer"**.

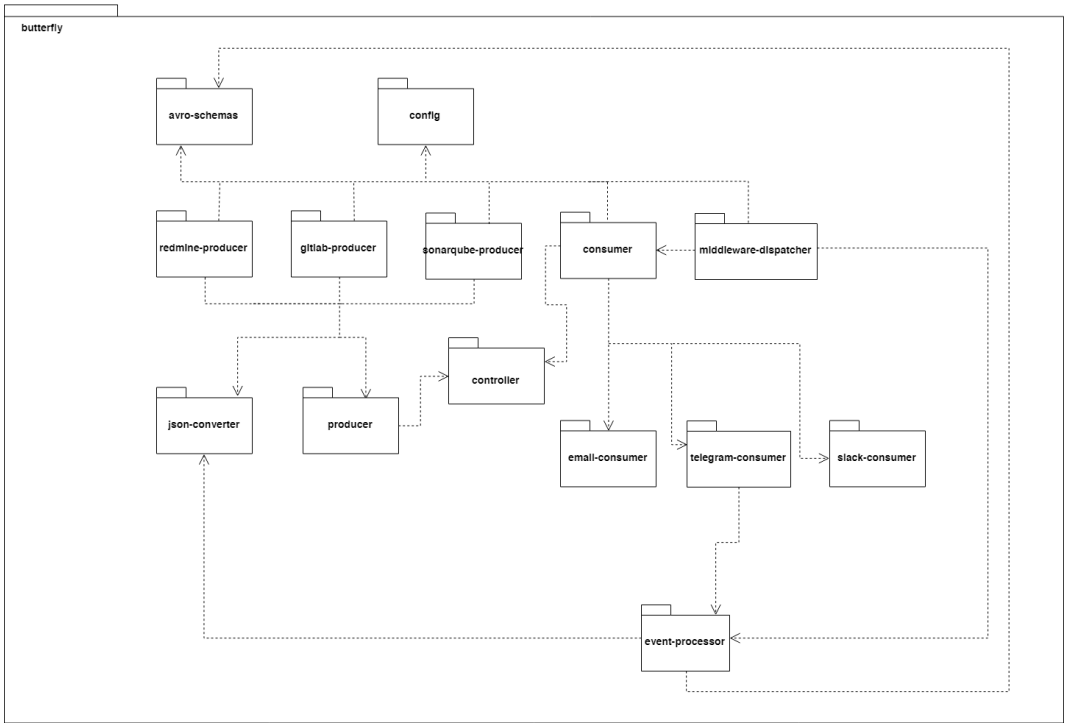

Figura 2: Butterfly:Package Diagram

5.2.1 Gestione della Configurazione

Uno degli aspetti di spicco delle applicazioni che aderiscono ai principi delle 12 Factor Apps è la flessibilità richiesta per gestire le configurazioni delle applicazioni. In Butterfly, i parametri configurabili dagli sviluppatori sono lette da variabili d'ambiente definite in file ad estensione `".env"` e iniettate nel contesto di esecuzione dei servizi Docker relativi. *AbstractConfigManager* è una classe astratta definita nel package *it.unipd.dstack.butterfly.config* del modulo `config` che permette di leggere valori di configurazioni, applicare conversioni di tipo (da stringa ad intero, da string a booleano) e ritornare valori di default nel caso in cui la configurazione cercata non sia correntemente settata. *AbstractConfigManager* applica il *Template Method Pattern*, in quanto definisce interamente la logica per il ritorno dei valori di default in caso di assenza di quanto cercato, di casting dei tipi e delle possibili eccezioni, ma delega alle classi concrete che la estenderanno il compito di definire come leggere le configurazioni (tale metodo è *AbstractConfigManager::readConfigValue*). Nel caso in cui la variabile di configurazione cercata non sia definita, e non sia passato nemmeno un valore di default al metodo del gestore delle configurazioni, viene lanciata un'eccezione. Fare fallire l'applicazione in caso di omissioni di configurazione obbligatorie è conforme alle *12 Factor App*, ed è inoltre una tecnica nota col nome di *fail fast*. L'implementazione utilizzata dai servizi è EnvironmentConfigManager, che legge le configurazioni necessarie dalle variabili d'ambiente. Esiste un modulo analogo scritto in Node.js per poter facilitare la lettura e la conversione di variabili di configurazione

da parte del Gestore Personale. Questi due moduli sono interamente testati tramite test d'unità la cui coverage è al 100%.

5.2.2 Diagramma dei package

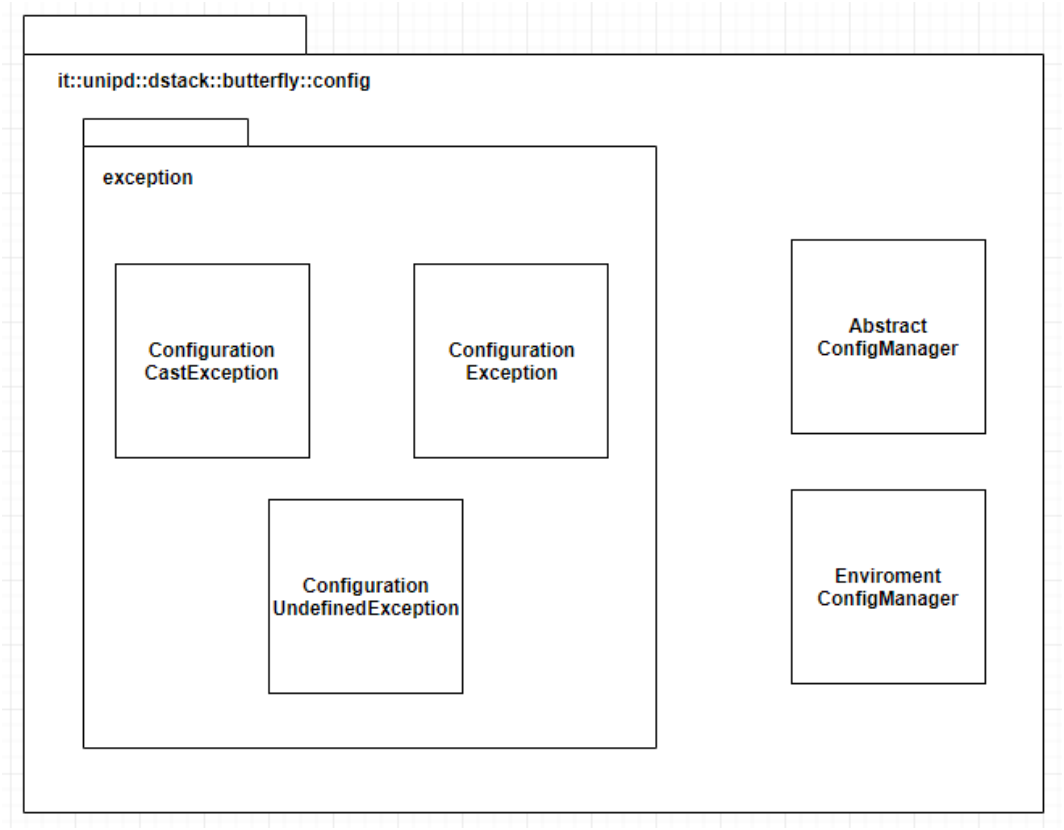

Figura 3: Diagramma dei package: *AbstractConfigManager*

5.2.3 Diagramma dei classi

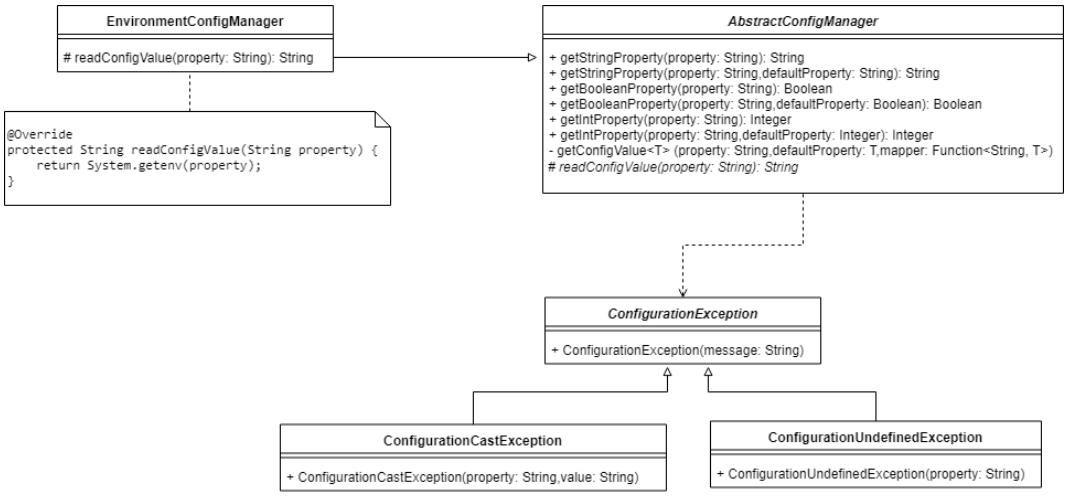

Figura 4: Diagramma delle classi: *AbstractConfigManager*

5.2.4 Controller

L'interfaccia più semplice e più generale è *Controller*, definita nel package *it.unipd.dstack.butterfly.controller* del modulo `controller`. Il suo contratto deve essere rispettato da ogni servizio di produzione e consumo. Il metodo *Controller::start* deve essere invocato per avviare l'applicazione, *Controller::close* la deve far terminare, rilasciando anche le risorse allocate durante l'esecuzione, e il metodo *Controller::gracefulShutdown* è un'implementazione di default dello spegnimento dell'applicazione all'arrivo di un interrupt di sistema (come SIGINT). L'interfaccia *Controller* è implementata da due classi astratte, *ConsumerController* e *ProducerController*, rispettivamente collocate nel package *it.unipd.dstack.butterfly.consumer.consumer.controller* del modulo `producer` e nel package *package it.unipd.dstack.butterfly.producer.producer.controller* del modulo `consumer`.

5.2.5 Diagramma dei package

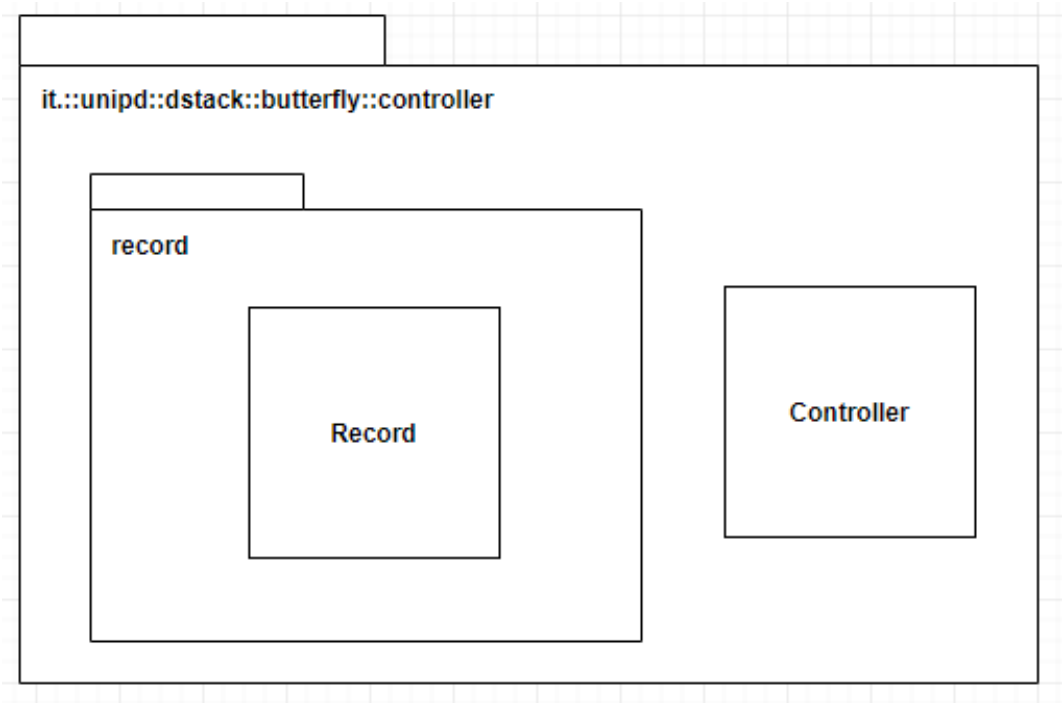

Figura 5: Diagramma dei package: *Controller*

5.2.6 Diagramma dei classi

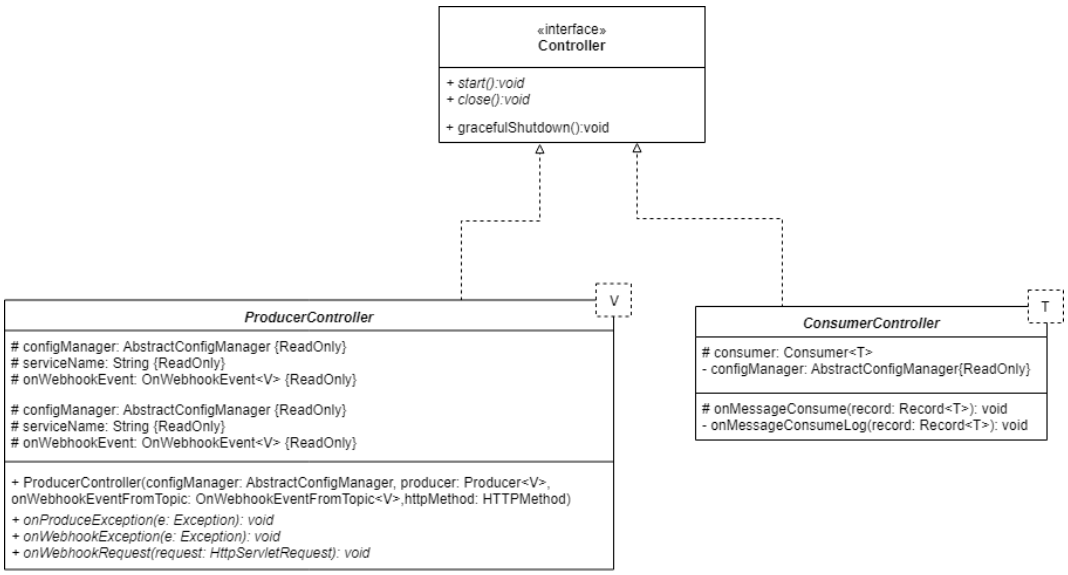

Figura 6: Diagramma delle classi: *Controller*

5.2.7 Producer

In Butterfly, i servizi raggruppati sotto il nome di **Producer** svolgono i seguenti ruoli:

- Inizializzano un server HTTP e attendono in ascolto l'arrivo di nuovi eventi da parte dell'applicazione di terze parti relativa al servizio in esecuzione;
- Decodificano gli eventi in ingresso estrapolando le informazioni più rilevanti;
- Utilizzano questo sottoinsieme di informazioni per creare degli oggetti *Event* la cui struttura sia normalizzata e comune a tutti gli altri servizi Producer all'interno di Butterfly;
- Inoltrano gli eventi normalizzati al Message Broker.

La definizione di come questo avvenga è descritto nella classe astratta *ProducerController* presente nel package *it.unipd.dstack.butterfly.producer.producer.controller* del modulo `producer`. Il server HTTP viene avviato in ascolto a seguito dell'invocazione del metodo *ProducerController::start*. La classe che si occupa di definire che metodo invocare quando arriva un nuovo evento da un'applicazione di terze parti è WebhookHandler, che è costruita tramite *Builder Pattern* ed è definita nel package *it.unipd.dstack.butterfly.producer.webhookhandler.WebhookHandler*. È grazie a questa classe che ad ogni richiesta HTTP ricevuta viene invocata l'implementazione concreta del metodo astratto *ProducerController::onWebhookRequest*.

Tutti i Producer supportati necessitano la definizione delle seguenti variabili d'ambiente:

- **SERVICE_NAME**: stringa che indica il nome del servizio correntemente in esecuzione. È usato principalmente a scopo di debug;
- **KAFKA_TOPIC**: stringa indica l'unico topic di Kafka in cui l'applicativo di tipo Producer andrà a scrivere;

- **SERVER_PORT**: la porta usata dal framework HTTP SparkJava per attenere le richieste HTTP.
- **WEBHOOK_ENDPOINT**: definizione della rotta HTTP che deve essere configurata nei servizi di produzione di terze parti (*GitLab*, *Redmine*, *SonarQube*) per inoltrare le notifiche degli eventi tramite WebHook.

I servizi Producer attualmente supportati da Butterfly riguardano le seguenti applicazioni di terze parti:

- **Redmine**;
- **GitLab**;
- **SonarQube** (applicativo Producer non ancora sviluppato).

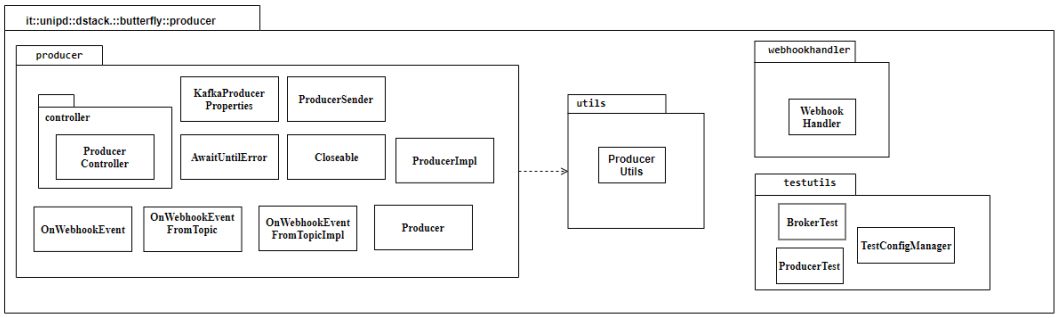

Figura 7: Diagramma dei package: *Producer*

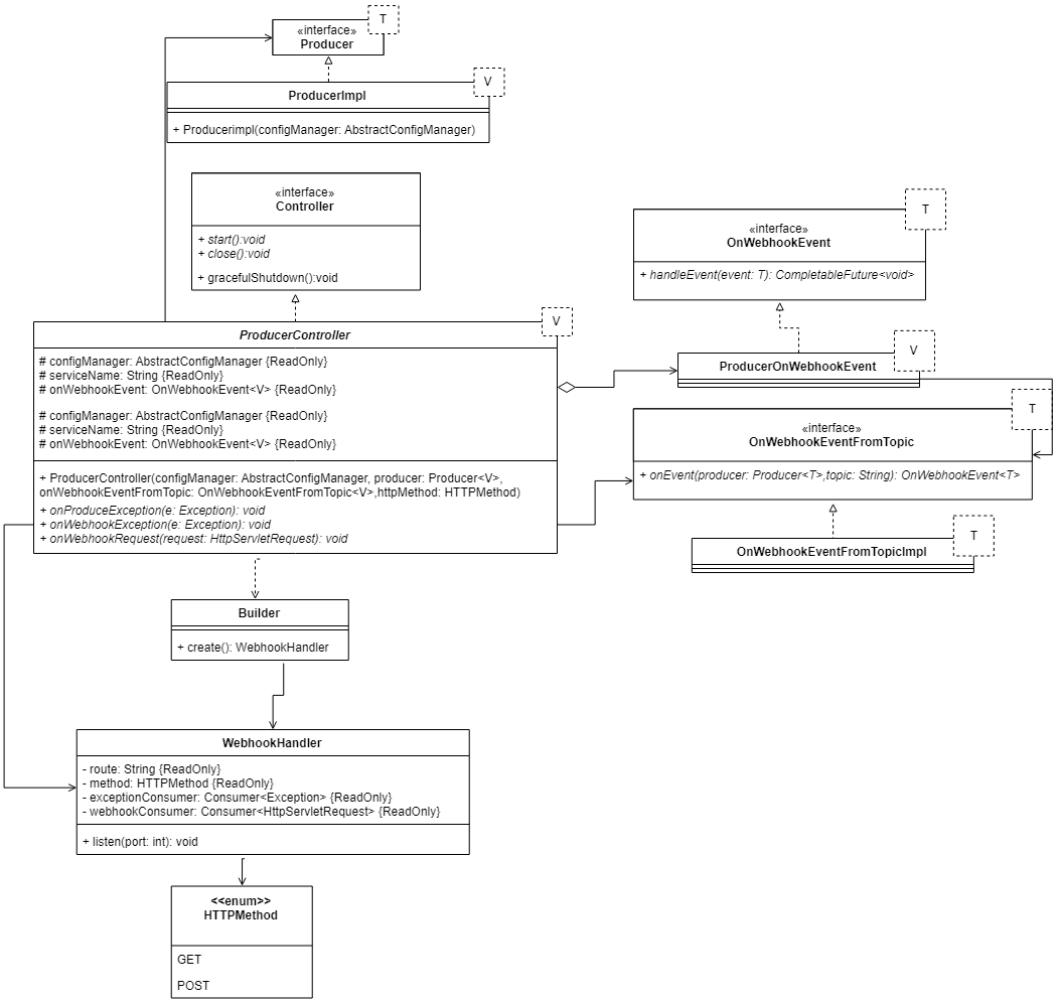

Figura 8: Diagramma delle classi: *Producer*

5.2.8 Consumer

In Butterfly, i servizi raggruppati sotto il nome di **Consumer** svolgono i seguenti ruoli:

- Iniziano un processo ciclico di *"discovery"* di messaggi nella coda del Broker;
- Da ogni messaggio appartenente all'eventuale nuovo blocco di messaggi letti (in cui la dimensione massima del blocco e l'intervallo di tempo con cui ne viene controllata l'esistenza sono regolabili da variabili d'ambiente) vengono estratte informazioni relative all'utente da contattare e il contenuto dell'evento da notificare (nella sua struttura normalizzata dalle applicazioni Producer);
- A partire da queste informazioni, l'applicazione di tipo Consumer crea un messaggio da inviare al sistema di terze parti relativo, formattando il testo nella maniera più appropriata;
- Il Consumer, che implementa un pattern Observer, notifica la ricezione del messaggio all' applicativo Controller specifico.
- Tale messaggio viene inviato al sistema di terze parti, che invierà finalmente una notifica dell'evento all'utente finale.

I servizi Consumer attualmente supportati da Butterfly fanno riferimento alle seguenti piattaforme di contatto di terze parti:

- **Telegram**;
- **Email**;
- **Slack**.

Tutti i Producer supportati necessitano la definizione delle seguenti variabili d'ambiente:

- **SERVICE_NAME**: stringa che indica il nome del servizio correntemente in esecuzione. È usato principalmente a scopo di debug;
- **KAFKA_TOPIC**: stringa indica l'unico topic di Kafka in cui l'applicativo di tipo Consumer rimmarà in ascolto;

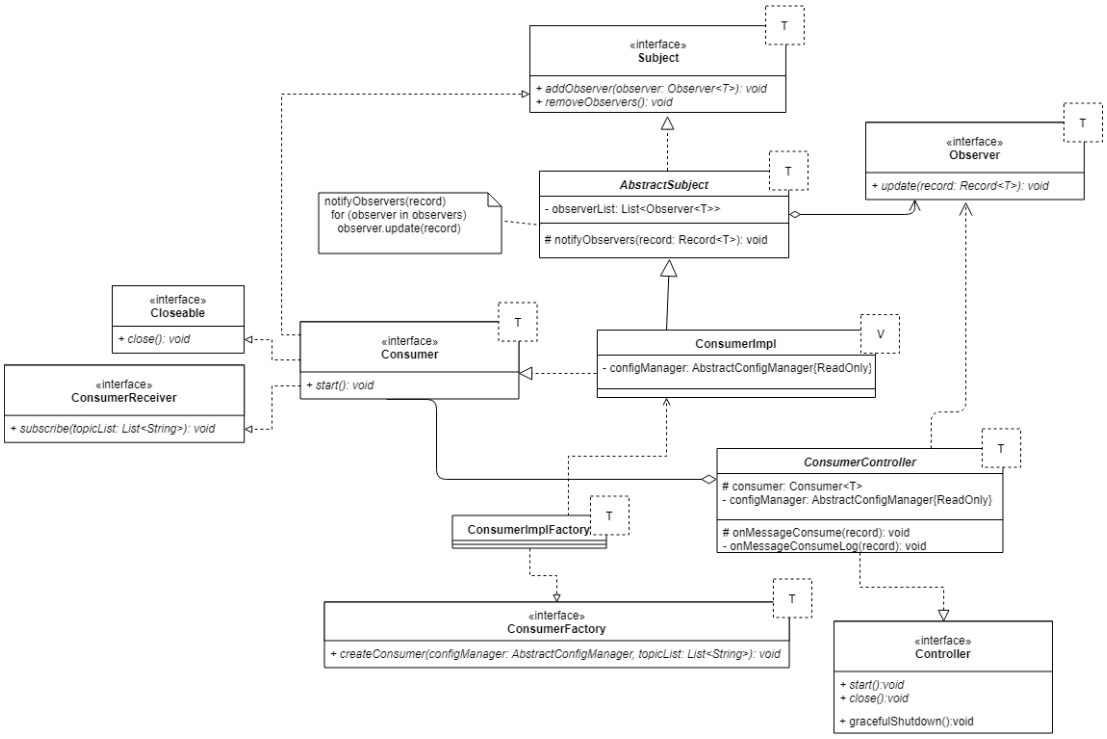

Figura 9: Diagramma delle classi: *Consumer*

5.2.9 Middleware Dispatcher

Il Middleware Dispatcher (o più semplicemente *Dispatcher*) è il nodo di transizione che collega i Producer, i Consumer e lo User Manager. Esso è incaricato di leggere tutti gli eventi normalizzati dai servizi Producer (che ricordiamo condividono la stessa struttura normalizzata, a prescindere dal servizio di produzione originale) e inoltrarli agli utenti che possono essere interessati alle informazioni di tali eventi. Il *Middleware Dispatcher*, come i *Producer* e i *Consumer*, è un servizio stateless, ovvero non salva alcuno stato in maniera persistente, e il carico di lavoro può quindi essere facilmente parallelizzato eseguendo più istanze del servizio Middleware Dispatcher (anche su macchine diverse). In effetti, il *Middleware Dispatcher* è sia un Producer

che un Consumer. In particolare, esso consuma gli eventi normalizzati di GitLab, Redmine e SonarQube inseriti nella coda di messaggi di Kafka, li inoltra al *Gestore Personale* (non prima di averne convertito la struttura in formato compatibile con le REST API, ovvero JSON), e produce tanti nuovi messaggi quanti sono i sistemi di contatto associati a tutti gli utenti ritornati dalla richiesta HTTP al Gestore Personale.

Per poter funzionare correttamente, il *Gestore Personale* necessita delle seguenti variabili d'ambiente:

- **SERVICE_NAME**: il nome del servizio in esecuzione, utilizzato principalmente nei log;
- **USER_MANAGER_URL**: stringa che indica l'URL da invocare per effettuare la ricerca degli utenti che devono ricevere la notifica dell'evento dello dal Message Broker;
- **USER_MANAGER_REQUEST_TIMEOUT_MS**: numero che indica il numero di millisecondi dopo il quale il *Gestore Personale* è considerato irraggiungibile;
- **MESSAGE_TOPIC_PREFIX**: stringa che indica il prefisso con il quale devono essere chiamati i topic di consumo. Il valore di default è "contact-", quindi i topic a cui si devono sottoscrivere i consumer sono "contact-telegram", "contact-slack", "contact-email".

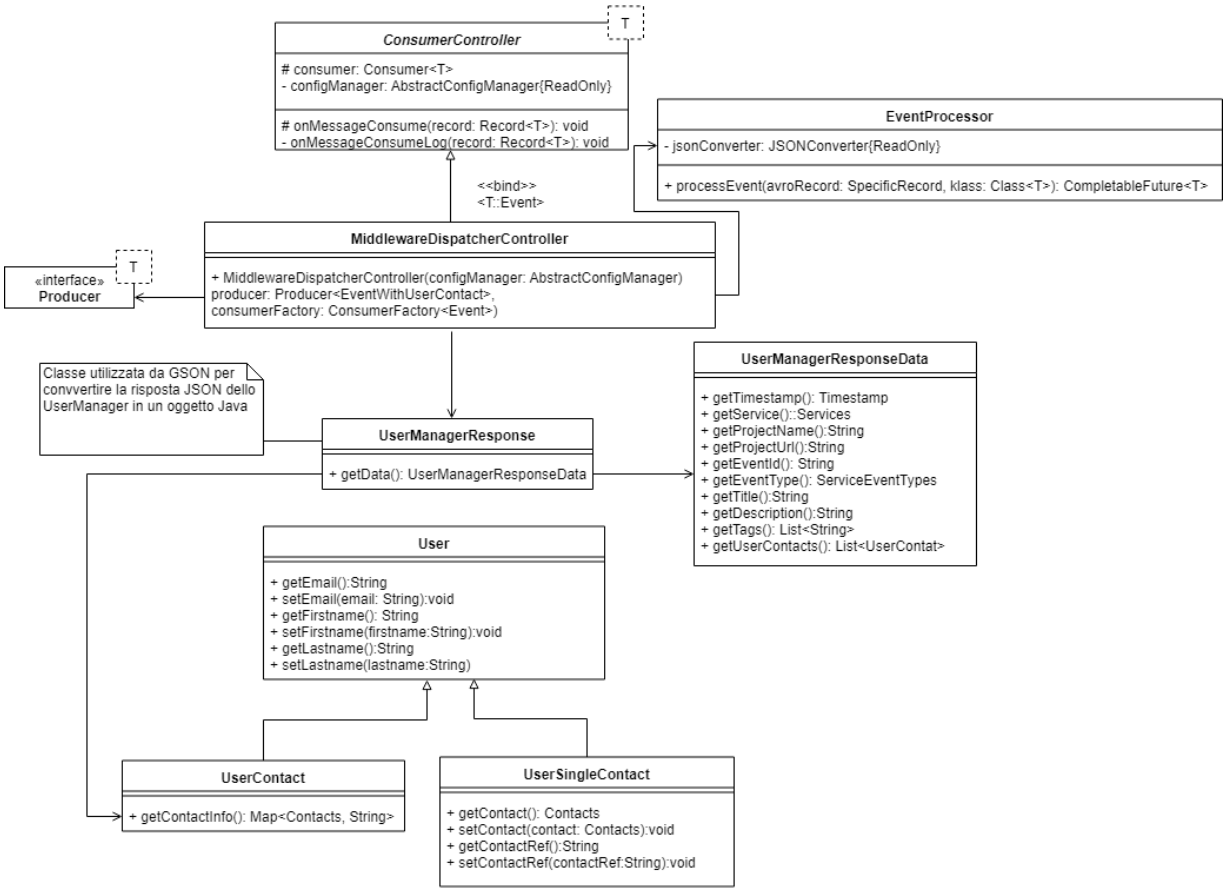

Figura 10: Diagramma delle classi: *Middleware Dispatcher*

5.2.10 User Manager

Lo **User Manager**, o *Gestore Personale*, rappresenta il cuore del progetto Butterfly, ed è l'unico microservizio di Butterfly che permette un'interazione diretta con gli utenti. Esso espone infatti API REST per gestire la registrazione degli utenti nel sistema, la configurazione degli account di contatto per gli utenti, la creazione e la gestione dei progetti di interesse, e la gestione delle iscrizioni degli utenti a progetti di interesse per gli utenti. Lo User Manager è incaricato di validare l'input degli utenti, inoltrare gli intenti della richiesta HTTP al database relazionale associato (basato su PostgreSQL), e restituire il risultato della richiesta all'utente. Ma il *Gestore Personale* non è usato solo dagli utenti esterni: esso è il nodo del sistema a cui si rivolge il servizio *Middleware Dispatcher* a seguito della lettura di un evento, e lo *User Manager* risponde alla richiesta del *Middleware Dispatcher* ritornando la lista degli utenti che più probabilmente sono interessati a ricevere una notifica dell'evento corrente. La logica del processo di decisione degli utenti che devono riceve le notifiche degli eventi in ingresso nel sistema è contenuta interamente nella query di ricerca definita nella *stored function* `public.search_receivers`. Tale funzione svolge le seguenti considerazioni:

- Se l'email dell'utente creatore dell'evento non è nulla, esclude l'utente che combacia con tale email dalla lista di utenti che dovrebbero essere notificati di quell'evento: essendone già l'autore, quella persona non deve ricevere una notifica dell'evento prodotto;
- Tutti gli utenti che sono marcati come "disabilitati" (campo `enabled` della tabella `public.user`) sono esclusi a priori dalla ricerca;
- Tutti gli utenti che non hanno alcuna iscrizione ad eventi attiva sono esclusi dai risultati della ricerca;
- Vengono invece presi in considerazione gli utenti che sono iscritti agli eventi del progetto coinvolto e che sono associati dal servizio di produzione originale (ad esempio, *GITLAB*);
- Di questi, vengono selezionati solamente quegli utenti le cui parole chiavi specificate al momento dell'iscrizione agli eventi compaiano nel titolo o nella descrizione dell'evento corrente;
- Per tutti gli utenti individuati, sono ritornati anche i sistemi di contatto ad essi associati con i quali gli utenti stessi hanno deciso di essere notificati dell'evento corrente.

Per poter funzionare correttamente, il *Gestore Personale* necessita delle seguenti variabili d'ambiente:

- **DATABASE_HOST**: stringa che indica la locazione del database Postgres nella rete. Tale valore è risolto tramite il processo di aliasing delle reti di *Docker Compose*;
- **DATABASE_NAME**: stringa che indica il nome del database a cui connettersi;
- **DATABASE_PASSWORD**: stringa che indica la password del database a cui connettersi;

- **DATABASE_PORT**: numero che indica la porta TCP di connessione al database Postgres. Di default è la numero 5432;
- **DATABASE_USER**: stringa che indica il nome dell'utente che sta instaurando una connessione con il database selezionato;
- **APP_NAME**: stringa utilizzata principalmente per scopi di debug, che indica il nome del servizio in esecuzione;
- **APP_HOST**: stringa che indica il nome dell'host di esecuzione del server dello *User Manager*;
- **APP_PORT**: numero che indica la porta TCP sulla quale si metterà in ascolto il server HTTP del gestore personale.

Il codice applicativo dello User Manager è presente nella cartella "./user-manager/user-manager-rest-api". La definizione dello schema del database è presente nella cartella "./user-manager/user-manager-database".

6 Producer

I servizi Producer attualmente supportati da Butterfly riguardano le seguenti applicazioni di terze parti:

- **Redmine**;
- **GitLab**;
- **SonarQube**.

6.1 Redmine producer

Redmine producer è un applicativo che ascolta richieste HTTP emesse dal plugin *redmine_webhook*[6] di Redmine. Esso è in grado di ricevere e normalizzare i seguenti eventi:

- creazione nuova segnalazione all'interno di un progetto Redmine;
- modifica segnalazione esistene all'interno di un progetto Redmine.

Il processo di normalizzazione di tali eventi avviene all'interno del package `it.unipd.dstack.butterfly.producer.redmine.webhookmanager`, che contiene un client in grado di tradurre le richieste HTTP in arrivo a oggetti evento inoltrabili alla coda di messaggi di Kafka. Non essendo stato trovato alcun client per Redmine scritto in Java, il gruppo ha provveduto a creare un client da 0. La struttura del client è simile a quella del client di Gitlab, sia in termini di nomenclature utilizzate, sia per come sono suddivise le classi. Il codice applicativo del Redmine Producer è contenuto all'interno della cartella "./butterfly/redmine-producer". Il topic utilizzato per la produzione di eventi provenienti da Redmine è "redmine-service".

6.1.1 Diagramma dei package

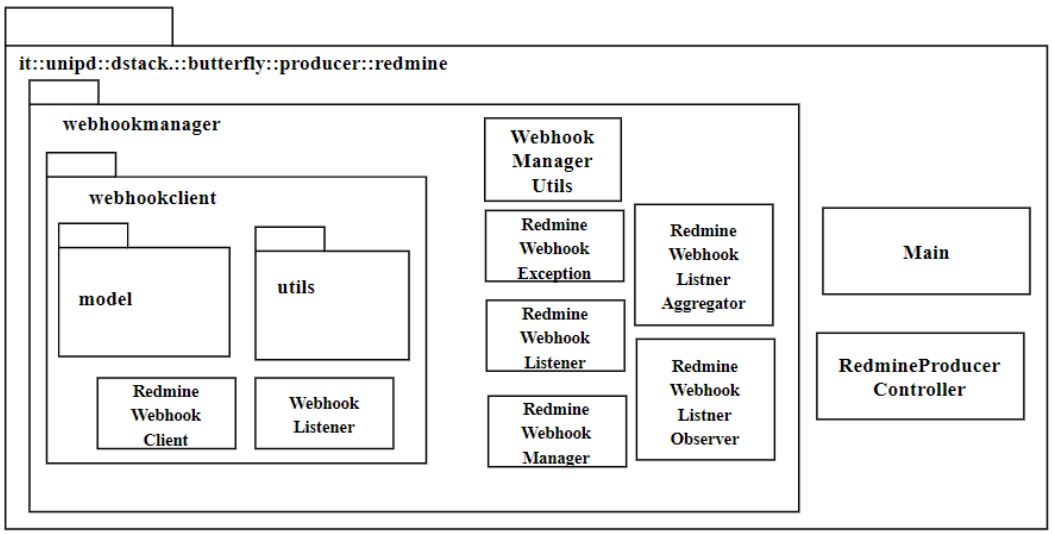

Figura 11: Diagramma dei package: *Redmine producer*

[6] https://github.com/suer/redmine_webhook

6.1.2 Diagramma delle classi

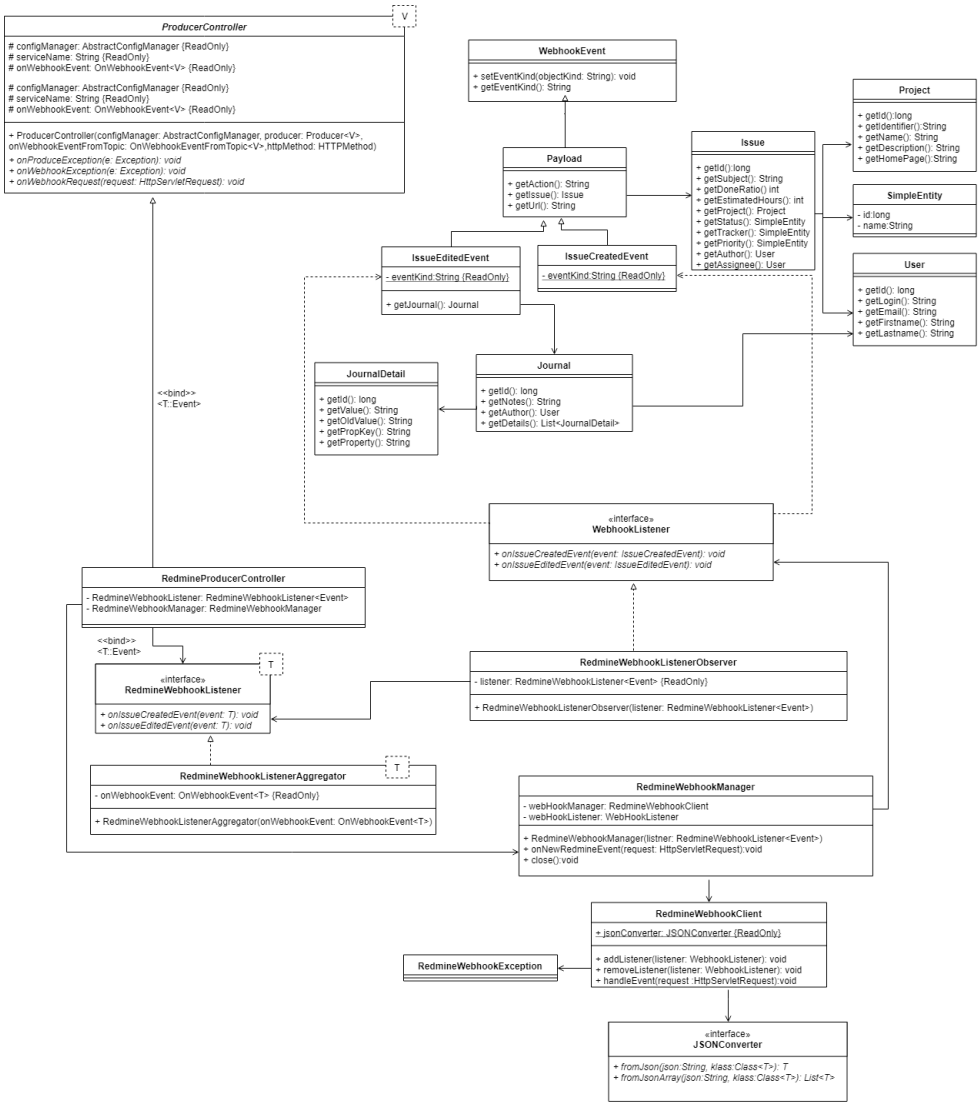

Figura 12: Diagramma delle classi: *Redmine producer*

6.2 Gitlab producer

Gitlab Producer è un applicativo che ascolta richieste HTTP emesse dalla piattaforma di versionamento web Gitlab. Esso è in grado di ricevere e normalizzare i seguenti eventi:

- nuovo Push di commit, dal quale il gruppo estrae informazioni relative ai singoli commit pubblicati in tale Push;
- evento generico relativo alle Issue, dal quale il gruppo estrae informazioni relative alla creazione e alla modifica delle Issue;
- evento generico relativo alle Merge Request, dal quale il gruppo estrae informazioni relative alla creazione, modifica, merge e chiusura senza accettazione della Merge Request considerata.

Per effettuare il parsing degli eventi in arrivo dal webhook di Gitlab abbiamo utilizzato la libreria *gitlab4j*, che abbiamo contenuto all'interno del package `it.unipd.dstack.butterfly.producer.gitlab.webhookmanager`.

Il codice applicativo del Gitlab Producer è presente nella cartella "./butterfly/gitlab-producer".

6.2.1 Diagramma dei package

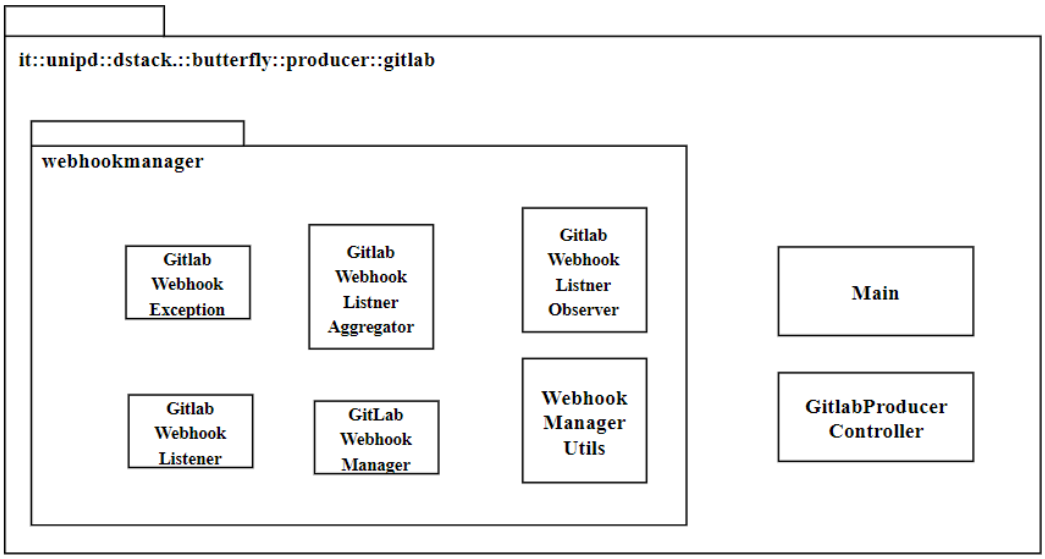

Figura 13: Diagramma dei package: *Gitlab producer*

6.2.2 Diagramma delle classi

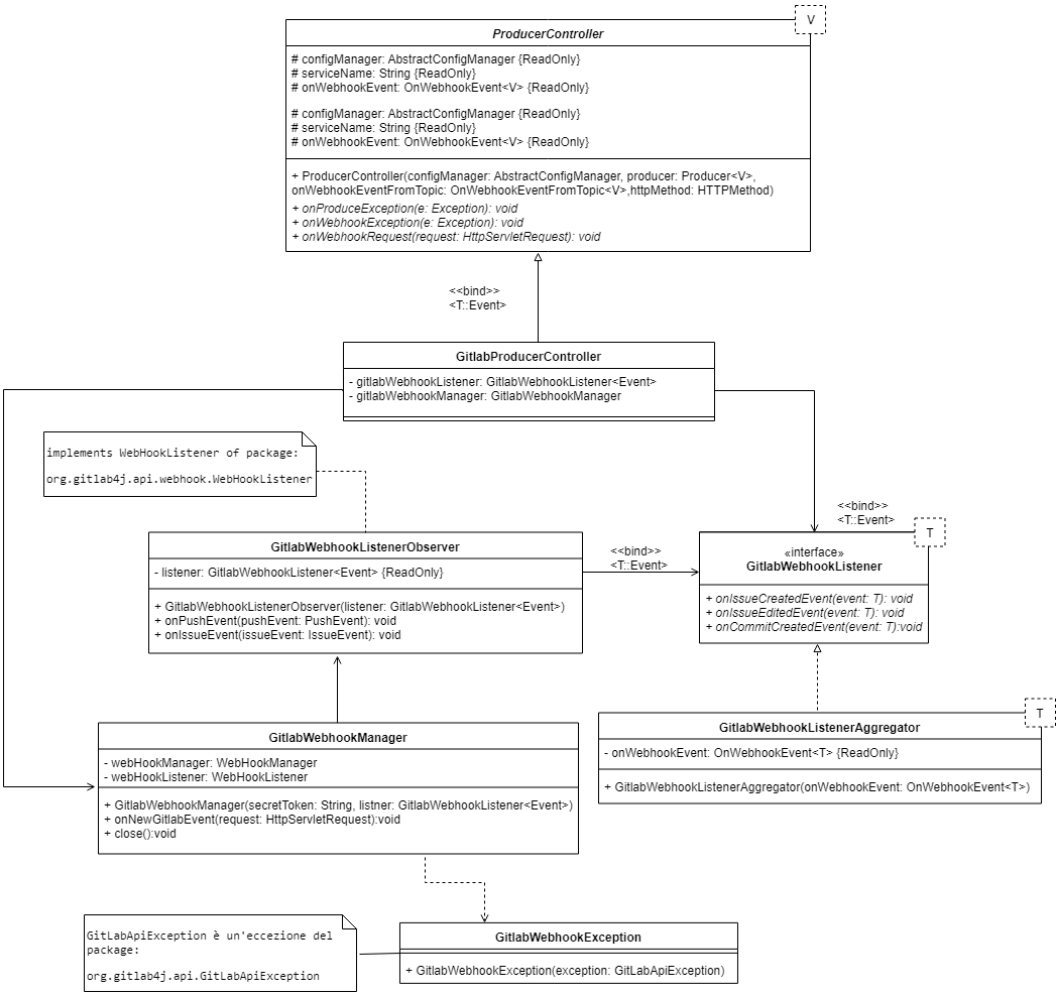

Figura 14: Diagramma delle classi: *Gitlab producer*

6.3 Sonarqube producer

Sonarqube producer è un applicativo che ascolta richieste HTTP emesse dal webhook di Somarqube. Esso è in grado di ricevere e normalizzare i seguenti eventi:

- Completamento dell'analisi di un progetto con Sonarqube;

Il processo di normalizzazione di tali eventi avviene all'interno del package `it.unipd.dstack.butterfly.producer.sonarqube.webhookmanager`, che contiene un client in grado di tradurre le richieste HTTP in arrivo a oggetti evento inoltrabili alla coda di messaggi di Kafka. Non essendo stato trovato alcun client per Sonarqube scritto in Java, il gruppo ha provveduto a creare un client da 0. La struttura del client è simile a quella del client di Gitlab, sia in termini di nomenclature utilizzate, sia per come sono suddivise le classi. Il codice applicativo del Sonarqube Producer è contenuto all'interno della cartella "./butterfly/sonarqube-producer". Il topic utilizzato per la produzione di eventi provenienti da Sonarqube è "sonarqube-service".

6.3.1 Diagramma dei package

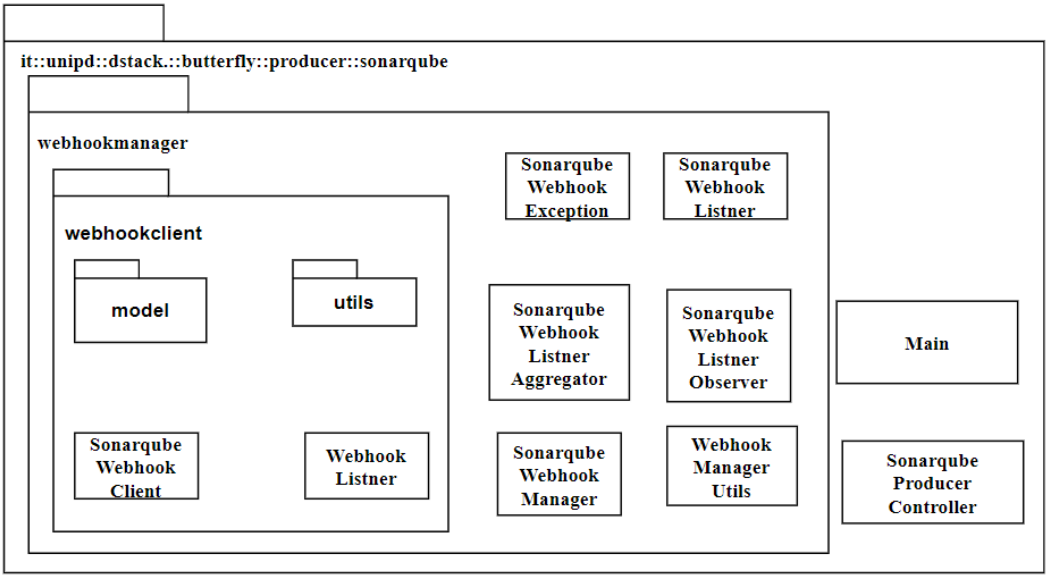

Figura 15: Diagramma dei package: *Sonarqube producer*

6.3.2 Diagramma delle classi

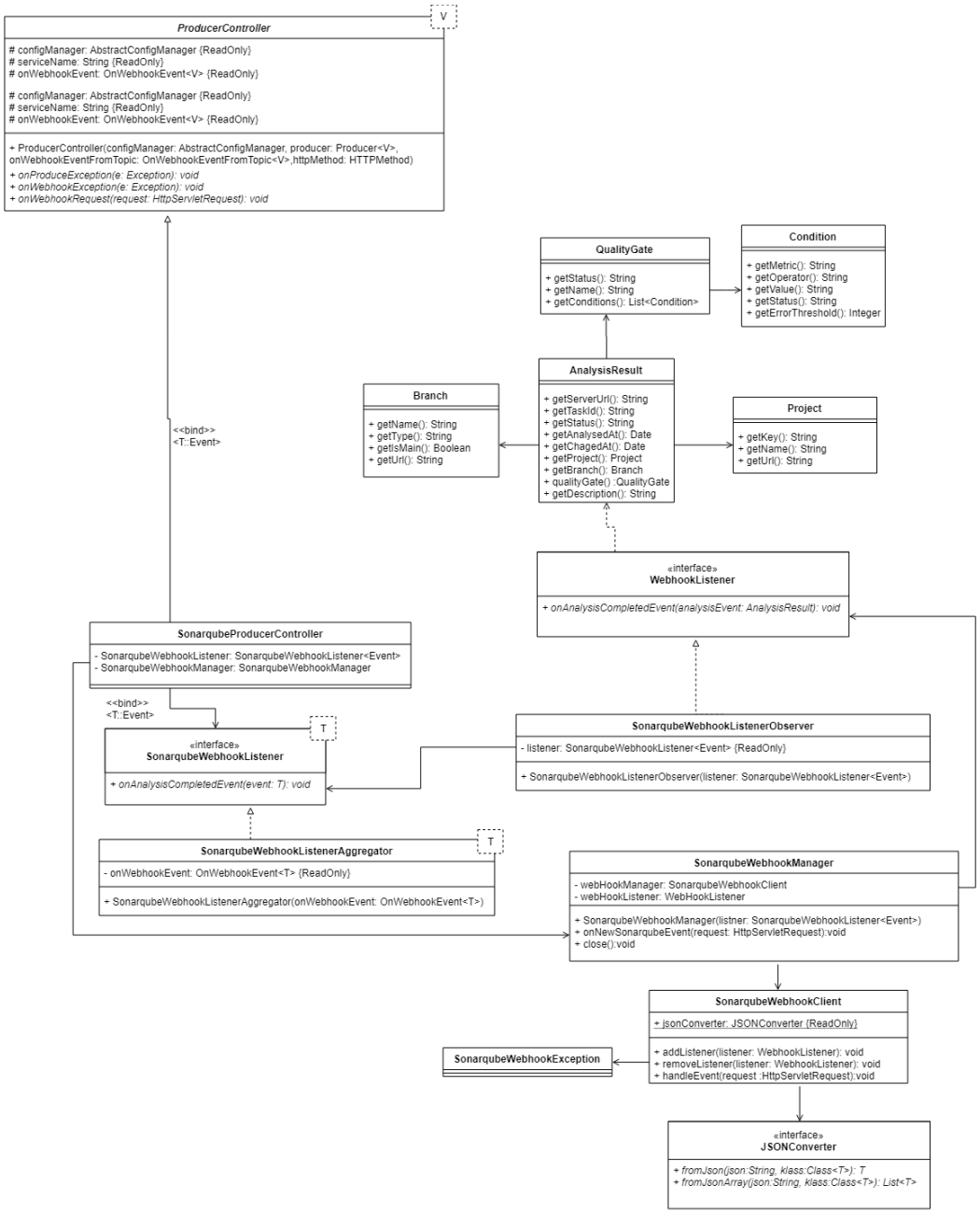

Figura 16: Diagramma delle classi: *Sonarqube producer*

7 Consumer

I servizi Consumer attualmente supportati da Butterfly riguardano le seguenti applicazioni di terze parti:

- **Email**;
- **Telegram**;
- **Slack**.

7.1 Email consumer

L'email consumer è un servizio che ascolta tutti i messaggi pubblicati nel topic "contact-email" del broker Kafka e, al loro arrivo, crea un nuovo messaggio email a partire dall'evento originale, imposta l'indirizzo email scelto di ognuno degli utenti individuati dal Gestore Personale e invia loro tale messaggio. L'implementazione attuale richiede un server SMTP, disponibile all'avvio dei processi Docker di Butterfly, ma è anche possibile impostare GMail come host SMTP, a patto di fornire le proprie credenziali GMail.

Email Consumer necessita dele seguenti variabili d'ambiente:

- **SERVICE_NAME**: il nome del servizio in esecuzione, utilizzato principalmente nei log;
- **KAFKA_TOPIC**: il nome del topic Kafka da cui prelevare gli eventi. Attualmente è "contact-email";
- **EMAIL_SERVER**: stringa che indica l'host SMTP da utilizzare per l'inoltro delle email degli eventi agli utenti;
- **EMAIL_ADDRESS**: indirizzo email dal quale risulterà inviata l'email;
- **EMAIL_PASSWORD**: password associata all'indirizzo email specificato precedentemente.

Il codice applicativo dell'Email Consumer è presente nella cartella "./butterfly/email-consumer".

7.1.1 Diagramma delle classi

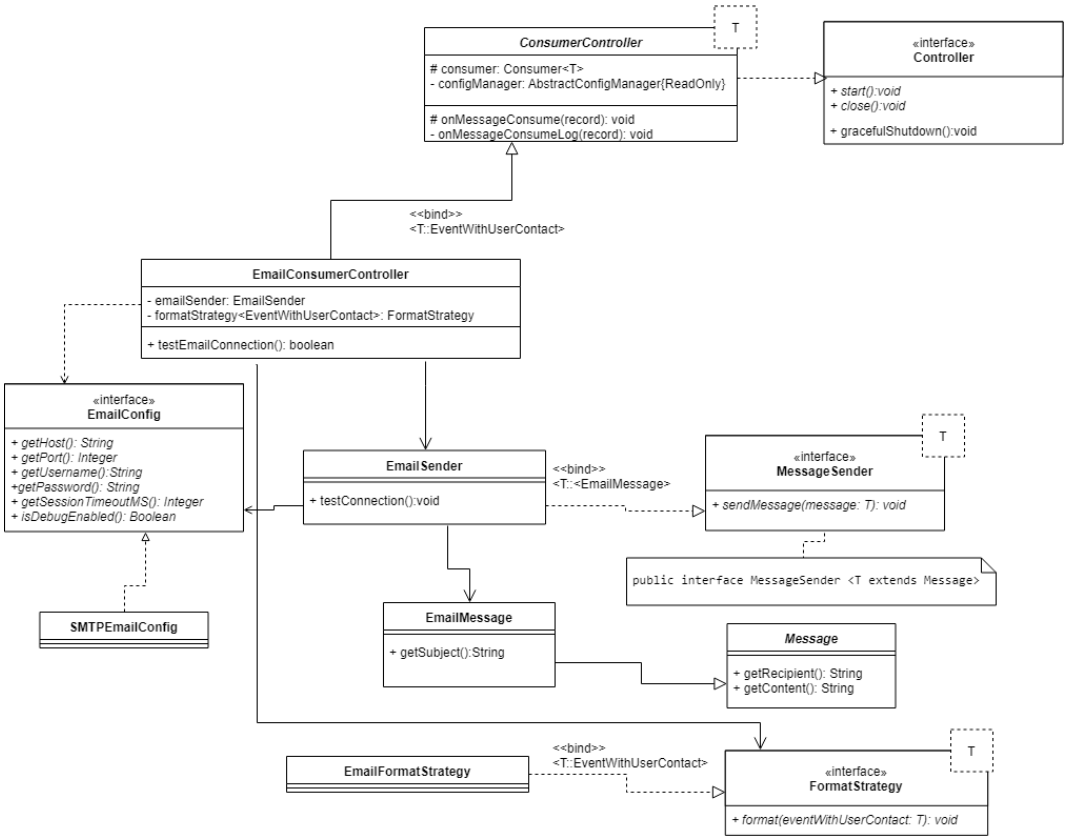

Figura 17: Diagramma delle classi: *Email consumer*

7.2 Telegram consumer

Telegram consumer è un servizio che ascolta tutti i messaggi pubblicati nel topic "contact-telegram" del broker Kafka e, al loro arrivo, crea un nuovo messaggio Telegram a partire dall'evento originale.
L'inoltro dei messaggi avviene tramite Bot Telegram. È richiesto che l'utente interagisca per prima con il bot di Telegram *ButterflyTelegramBot*, specificando la propria email. Tale email sarà inviata allo *User Manager* tramite richiesta HTTP, che, nel caso venga individuato un utente registrato con lo stesso indirizzo email, provvederà a creare un account di contatto relativo a Telegram con l'id (univoco) della conversazione in corso tra l'utente e il bot. Tale id sarà poi letto dal campo "contactRef" del messaggio letto dalla coda di Kafka, e sarà usato per l'inoltro delle notifiche relative agli eventi di interesse per l'utente compatibile con tale chat id.

Telegram Consumer necessita dele seguenti variabili d'ambiente:

- **SERVICE_NAME**: il nome del servizio in esecuzione, utilizzato principalmente nei log;
- **USER_MANAGER_URL**: stringa che indica l'URL da invocare per effettuare la creazione di un account di contatto Telegram a partire dall'email dell'utente e dall'id della conversazione con il bot Telegram;

- **USER_MANAGER_REQUEST_TIMEOUT_MS**: numero che indica il numero di millisecondi dopo il quale il *Gestore Personale* è considerato irraggiungibile;
- **KAFKA_TOPIC**: il nome del topic Kafka da cui prelevare gli eventi. Attualmente è "contact-email";
- **TELEGRAM_TOKEN**: è il token generato da Telegram per autenticare le interazioni tra il Bot di Telegram gestito da Telegram Consumer e i server di Telegram.

Il codice applicativo del Telegram Consumer è presente nella cartella "./butterfly/telegram-consumer".

7.2.1 Diagramma delle classi

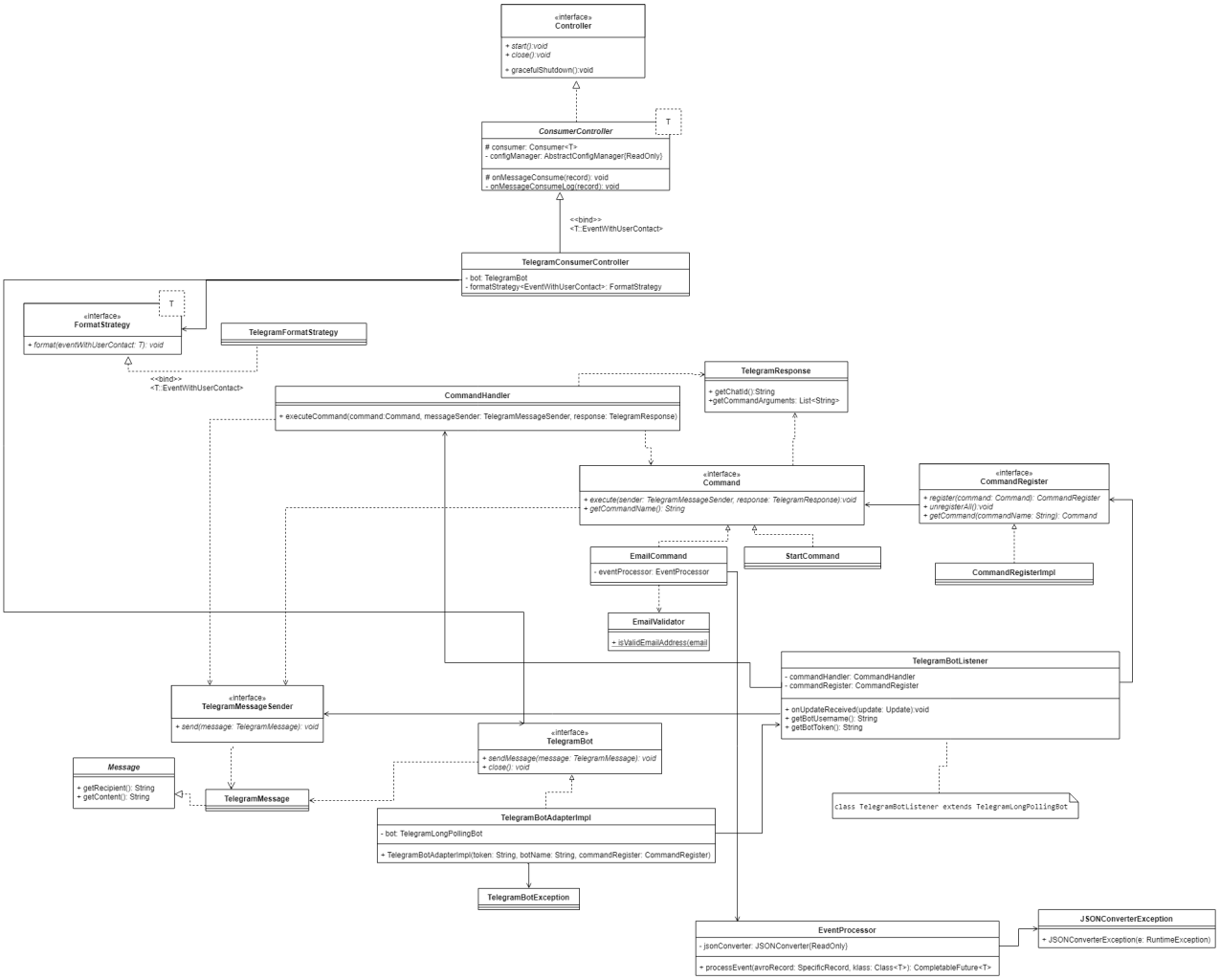

Figura 18: Diagramma delle classi: *Telegram consumer*

7.3 Slack consumer

Lo **Slack Consumer** ha il compito di ricevere tutti gli eventi prodotti destinati ad utenti in possesso di un account di contatto Slack. Tale consumer riceve solo messaggi

dal topic specificato dalla variabile di configurazione `KAFKA_ TOPIC`, che di default ha valore *contact-slack*. Il tipo di messaggi consumati è *EventWithUserContact*, un record Avro contenente l'evento da notificare e le informazion di contatto dell'utente a cui va inoltrato tale evento.

7.3.0.1 Configurazione account Slack

Affinché il servizio Middleware Dispatcher invii un messaggio allo Slack Consumer, è necessario che almeno uno tra gli utenti destinati a ricevere una notifica relativa all'evento considerato abbia un account Slack, e abbia espresso come preferenza questa piattaforma di contatto per l'invio delle notifiche corrispondenti alla subscription corrente.
È quindi necessario che l'utente abbia configurato il proprio account di contatto di Slack nel Gestore Personale. Dal momento che il riferimento di contatto per la piattaforma di contatto Slack è l'id di un utente di Slack, informazione non facilmente reperibile, è stata sviluppato il servizio Slack Account Configurator, che ha lo scopo di rendere user friendly la configurazione dell'account di contatto Slack nel Gestore Personale. Esso rappresenta il Backend per l'applicazione Butterfly di Slack. Il codice di questa applicazione è scritta in TypeScript ed è disponibile in *./user-manager/slack-account-configurator*.
La libreria principale utilizzata per la realizzazione di questo microservizio è Bolt, ovvero la libreria JavaScript/TypeScript ufficiale per l'interazione con Slack.

7.3.0.2 Comandi Slack

Slack Account Configurator accetta un solo comando, `/email`, che richiede di inserire l'email di un utente esistente nel Gestore Personale, e può ritornare i seguenti messaggi:

- **Caso di successo**:
  - **"Successfully added a new Slack Account for $email"**
- **Casi d'insuccesso**:
  - **"Another Slack Account is already configured for the user identified by"**: ciò avviene quando l'utente identificato dall'email usata nel comando (che è stata validata con successo) ha già creato un account di contatto Slack in Butterfly;
  - **"Can't contact Butterfly's User Manager service. Please try again later."**: questo messaggio viene presentato quando non è stato possibile contattare lo User Manager entro il tempo massimo stabilito (attualmente fissato ad 1 secondo);
  - **"Unexpected error. Please contact an administrator."**: questo errore si presenta in situazioni errore non gestite.

Per informazioni più dettagliate sui comandi personalizzati per applicazioni Slack, si prega di far riferimento alla documentazione ufficiale degli Slash Commands di Slack.

7.3.0.3 Inoltro messaggi Slack

Tutti i messaggi in arrivo allo Slack Consumer sono inoltrati a Slack tramite messaggio privato all'utente individuato dal Gestore Personale. Il modo con cui Slack

permette l'invio automatizzato di messaggi privati agli utenti è tramite bot. Il bot butterflydstackbot è stato creato allo scopo, e la sua logica riesiede nel modulo `./butterfly/slack-consumer`.
La libreria SimpleSlackAPI è stata utilizzata per la realizzazione del client del bot Slack, e il suo utilizzo è stato incapsulato all'interno della classe SlackBotAdapter, che svolge il ruolo di wrapper. Grazie all'applicazione dell'Adapter Pattern, qualora si rendesse necessario cambiare libreria di terze parti per la gestione del bot, è sufficiente cambiare solo l'implementazione della classe SlackBotAdapter, senza doverne alterare i metodi pubblici.

Per informazioni più dettagliate sulle funzionalità offerte dai bot per Slack, si prega di far riferimento alla documentazione ufficiale dei bot Slack.

7.3.1 Diagramma delle classi

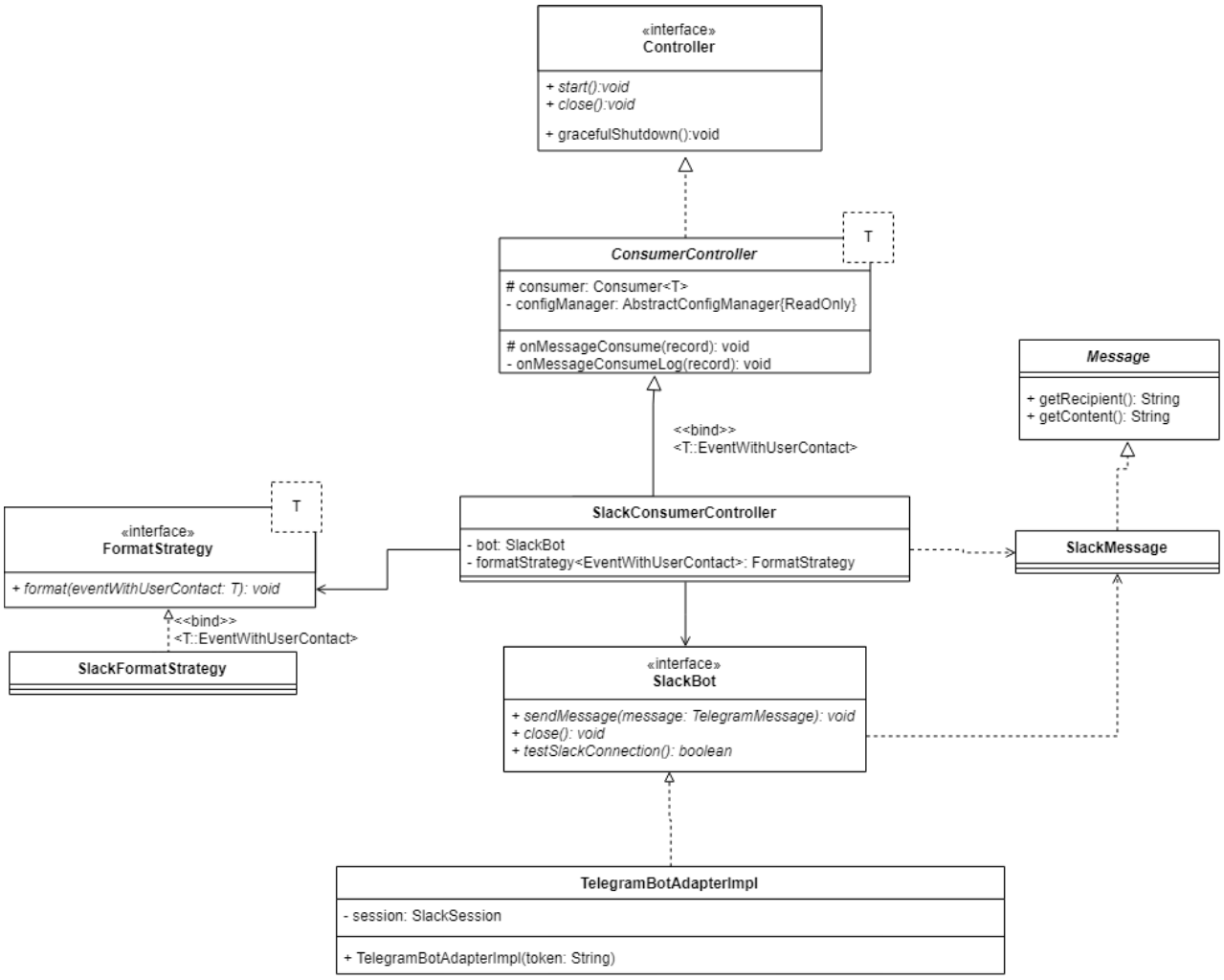

Figura 19: Diagramma delle classi java: *Slack consumer*

8 Middleware dispatcher

Il Middleware dispatcher è il servizio che si occupa di prelevare le segnalazioni dai topic: "gitlab-service", "redmine-service", "sonarqube-service". In particolare, esso consuma gli eventi normalizzati di GitLab, Redmine e SonarQube inseriti nella coda di messaggi di Kafka, li inoltra al *Gestore Personale* (non prima di averne convertito la struttura in formato compatibile con le REST API, ovvero JSON), e produce tanti nuovi messaggi quanti sono i sistemi di contatto associati a tutti gli utenti ritornati dalla richiesta HTTP al Gestore Personale.

Il codice applicativo del Middleware Dispatcher è presente nella cartella "./butterfly/middleware-dispatcher".

8.0.1 Diagramma dei package

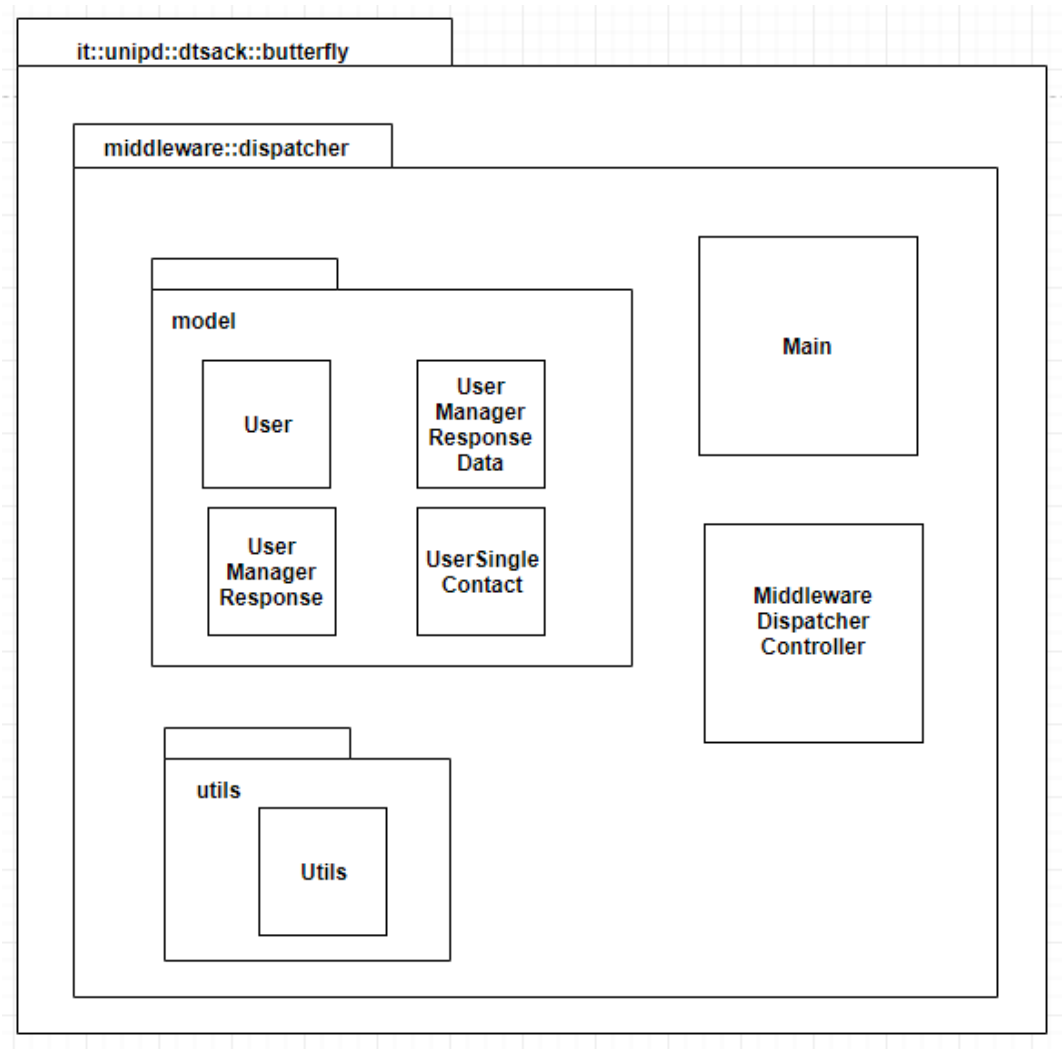

Figura 20: Diagramma dei package: *Middleware Dispatcher*

8.1 Diagramma delle classi

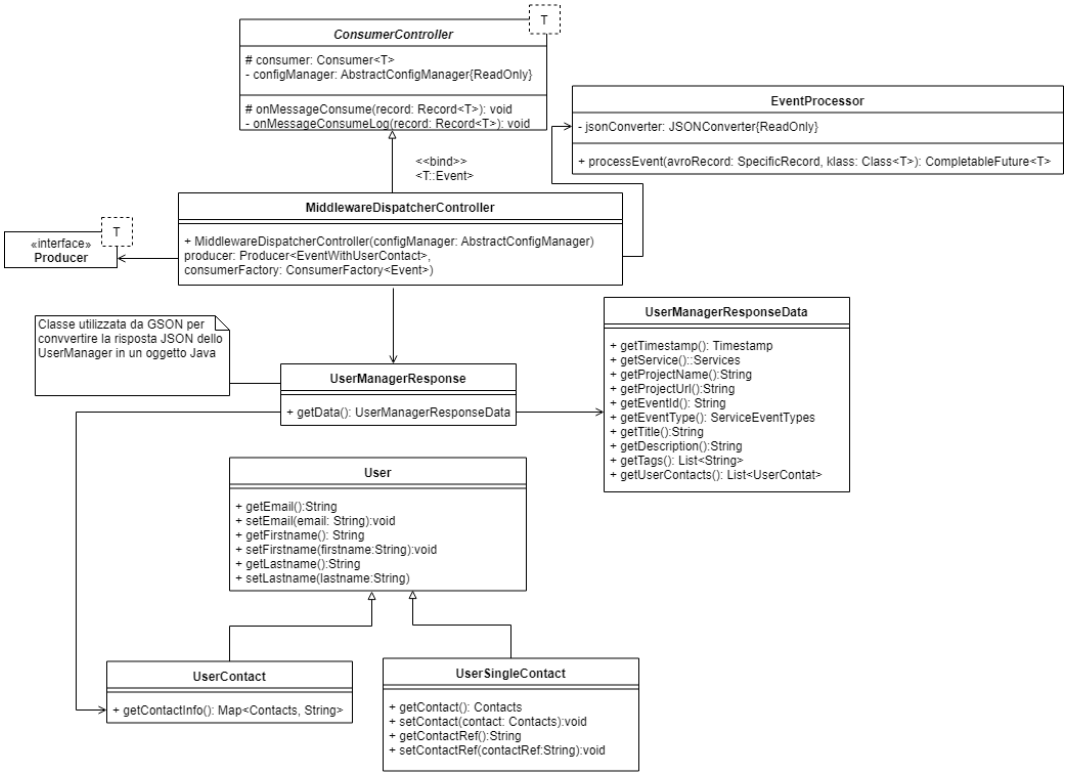

Figura 21: Diagramma delle classi: *Middleware Dispatcher*

9 Documentazione Gestore Utente

9.1 Moduli Gestore Personale

La logica operativa degli endpoint REST dello User Manager è contenuta nella sotto cartella *src/modules*. Essa contiene tante sottocartelle quanti sono i principali gruppi di risorse che il Gestore Personale permette di gestire:

- **users**: contiene la logica operativa CRUD per la gestione degli utenti;
- **projects**: contiene la logica operativa CRUD per la gestione dei progetti;
- **subscriptions**: contiene la logica operativa CRUD per la gestione delle preferenze degli utenti riguardo a specifici eventi dei progetti disponibili;
- **userContacts**: contiene la logica operativa CRUD per la gestione degli account di contatto degli utenti tramite le piattaforme TELEGRAM, SLACK e EMAIL;
- **search**: contiene la logica per la ricerca degli utenti che possono essere interessati ad un certo tipo di evento.

Esiste inoltre il modulo **health**, il cui scopo è quello di permettere di capire il consumo di risorse del server e se il server sia pronto a ricevere richieste o meno.

9.2 Database

Il database utilizzato dal Gestore Personale è Postgres 11, e la definizione delle tabelle, procedure e quant'altro componga la logica lato database dello User Manager risiede nel file user-manager/user-manager-database/sql/ddl.sql. Andiamo di seguito a presentare i tipi e le tabelle più importanti della definizione del database di questo servizio.

9.2.1 Enum

Per convenzione, ogni valore che compone un tipo enumerabile è salvato nel formato *SCREAMING_CAMEL_CASE*, ovvero in maiuscolo e con ogni parola separata da un simbolo di underscore (_). Inoltre, per tutti i tipi enumerabili presentati di seguito, è possibile utilizzare la stored function `get_enum_values(regtype)` per ottenere la lista completa e aggiornata degli elementi che definiscono il tipo enumerabile passato come argomento.

9.2.1.1 public.producer_service

Questo tipo enumera la lista dei servizi di produzione supportati. Attualmente, i servizi di produzione supportati sono i seguenti:

- REDMINE;
- GITLAB;
- SONARQUBE.

Query per ottenere la lista aggiornata degli elementi di questo tipo:

```
SELECT * FROM get_enum_values('public.producer_service');
```

9.2.1.2 public.consumer_service

Questo tipo enumera la lista dei servizi di consumo supportati. Attualmente, i servizi di consumo supportati sono i seguenti:

- TELEGRAM;
- EMAIL;
- SLACK.

Query per ottenere la lista aggiornata degli elementi di questo tipo:

```
SELECT * FROM get_enum_values('public.consumer_service');
```

9.2.1.3 public.user_priority

Questo tipo enumera la lista di priorità supportate per un utente. Attualmente, le priorità utente supportate sono le seguenti:

- LOW;
- MEDIUM;
- HIGH.

Query per ottenere la lista aggiornata degli elementi di questo tipo:

```
SELECT * FROM get_enum_values('public.producer_service');
```

9.2.1.4 public.service_event_type

Questo tipo enumera la lista di tipologie di evento supportate per ogni servizio di produzione registrato nel sistema. Ogni tipologia di evento è salvata con il prefisso `${SERVICE}_`, dove ${SERVICE} indica un servizio di produzione tra quelli supportati. Di seguito sono presentate le tipologie di evento supportate, raggruppate per servizio di produzione relativo.

- Tipologie di evento per **REDMINE**:
  - **REDMINE_TICKET_CREATED**: corrisponde all'evento di creazione di una nuova segnalazione in Redmine;
  - **REDMINE_TICKET_EDITED**: corrisponde all'evento di modifica di una segnalazione in Redmine.
- Tipologie di evento per **GITLAB**:
  - **GITLAB_COMMIT_CREATED**: a partire da un singolo evento di push del codice sorgente in una repository ospitata su GitLab, Butterfly emette tanti eventi di creazione commit quanti sono i commit coinvolti nel push corrente, con un limite massimo di 20 commit;[7]

[7] https://docs.gitlab.com/ee/user/project/integrations/webhooks.html#push-events

  - **GITLAB_ISSUE_CREATED**: a partire da un evento relativo ad una issue di GitLab, se non è presente una data di modifica, Butterfly interpreta tale evento come uno di creazione issue;
  - **GITLAB_ISSUE_EDITED**: a partire da un evento relativo ad una issue di GitLab, se è presente una data di modifica, Butterfly interpreta tale evento come uno di modifica issue;
  - **GITLAB_MERGE_REQUEST_CREATED**: corrisponde all'evento di apertura di una nuova merge request in GitLab;
  - **GITLAB_MERGE_REQUEST_EDITED**: corrisponde all'evento di modifica del contenuto di una merge request in GitLab;
  - **GITLAB_MERGE_REQUEST_MERGED**: corrisponde all'evento di merge di una merge request in GitLab;
  - **GITLAB_MERGE_REQUEST_CLOSED**: corrisponde all'evento di chiusura senza merge di una merge request in GitLab.

- Tipologie di evento per **SONARQUBE**:
  - **SONARQUBE_PROJECT_ANALYSIS_COMPLETED**: corrisponde all'evento di completamento dell'analisi statica di un progetto con SonarQube.

9.2.2 Tabelle

Nella seguente sezione, con la sigla **PK** identificheremo la chiave primaria, mentre con **FK** andremo ad indicare una chiave esterna.

9.2.2.1 public.service

La tabella `public.service` assegna un ID relativo ai servizi di comunicazione come *TELEGRAM* per poterli collegare in seguito con la relazione rappresentata dalla tabella `public.x_service_event_type`. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
service_id	`bigint`	`2`	**PK**
producer_service_key	`public.service`	`GITLAB`	Servizio di produzione, unico

Tabella 1: public.service

9.2.2.2 public.event_type

La tabella `public.event_type` assegna un ID relativo a tipologie di evento come *GITLAB_COMMIT_CREATED* per poterli collegare in seguito con la relazione rappresentata dalla tabella `public.x_service_event_type`. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
event_type_id	`bigint`	`10`	**PK**
event_type_key	`public.service_event_type`	`GITLAB_COMMIT_CREATED`	Tipologia di evento di produzione, unico

Tabella 2: public.event_type

9.2.2.3 public.x_service_event_type

La tabella `public.x_service_event_type` rappresenta l'associazione molti a molti tra i servizi di comunicazione di `public.service` e le tipologie di evento di `public.event_type`. Ad esempio, questa tabella permette di dire che l'evento *GITLAB_COMMIT_CREATED* è uno degli eventi supportati dal servizio di produzione *GITLAB*. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
x_service_event_type_id	`bigint`	`5`	**PK**
service_id	`bigint`	`2`	**FK** di `public.service.service_id`. ID di un servizio di comunicazione, unico insieme a event_type_key
event_type_id	`bigint`	`10`	**FK** di `public.event_type.event_type_id`. ID di una tipologia di evento di produzione, unico insieme a service_id

Tabella 3: public.x_service_event_type

9.2.2.4 public.project

La tabella `public.project` modella l'entità "progetto" del Gestore Utente. Ogni progetto è identificato da un nome e può avere una URL associata per ogni servizio di produzione supportato. Ad esempio, il progetto chiamato "Butterfly" può essere citato sia in eventi provenienti da GitLab (ad esempio: `https://gitlab.com/dstack/butterfly`) sia da eventi di analisi provenienti da SonarQube (ad esempio: `https://sonarcloud.io/dstack/butterfly`). Non può esistere un progetto senza almeno una URL associata. Le URL dei progetti sono salvate in una serializzazione binaria del formato JSON, e sono costruite come una mappa chiave-valore in cui la chiave è il nome del sistema di produzione (ad esempio GITLAB) e il valore è la rispettiva URL (ad esempio: `https://gitlab.com/dstack/butterfly`). È compito delle query e delle funzioni che interagiscono con la tabella `public.project` assicurarsi che tale struttura JSON sia rispettata. Il vincolo di univocità per il nome di un progetto è case insensitive, grazie all'indice `project_name_unique_idx`. Solo il campo `modified` è nullable, e solo il campo `created` ha un valore di default (corrispondente alla data e all'ora cor-

rente di creazione). La cancellazione di un progetto comporta la distruzione di tutti i record della tabella `public.subscription` in cui è coinvolto il progetto eliminato.

Nome	Tipo	Esempio	Commento
project_id	`bigint`	`1`	**PK**
project_name	`varchar(30)`	`'Butterfly'`	Nome del progetto, unico
project_url	`jsonb`	`'{"GITLAB": "https://gitlab.com/dstack/butterfly"}'`	Mappa delle URL associate al progetto
created	`timestamp`	`'2019-05-02 14:48:25.7927+00'`	Data e ora di creazione
modified	`timestamp`	`NULL`	Data e ora di ultima modifica

Tabella 4: public.project

9.2.2.5 public.user

La tabella `public.user` modella l'entità "utente" del Gestore Utente. Ogni utente è identificato da una email e possiede un nome ed un cognome. Gli utenti possono essere abilitati o disabilitati grazie al flag booleano `enabled`, che assume il valore `true` come impostazione predefinita. Gli utenti disabilitati non compaiono nelle query di ricerca esposte dalle REST API dello User Manager. Il vincolo di univocità per l'email di un utente è case insensitive, grazie all'indice `user_email_unique_idx`. Solo il campo `modified` è nullable, e solo il campo `created` ha un valore di default (corrispondente alla data e all'ora corrente di creazione). La cancellazione di un utente comporta la distruzione di tutti i record delle tabelle `public.subscription` e `public.user_contact` in cui è coinvolto l'utente eliminato.

Nome	Tipo	Esempio	Commento
user_id	`bigint`	`1`	**PK**
email	`varchar(30)`	`'johndoe@gmail.com'`	Email dell'utente, unica
firstname	`varchar(30)`	`'John'`	Nome dell'utente
lastname	`varchar(30)`	`'Doe'`	Cognome dell'utente
enabled	`boolean`	`true`	Vero solo se l'utente corrente è abilitato
created	`timestamp`	`'2019-05-02 14:48:25.7927+00'`	Data e ora di creazione
modified	`timestamp`	`NULL`	Data e ora di ultima modifica

Tabella 5: public.project

9.2.2.6 public.user_contact

La tabella `public.user_contact` modella l'entità "account di contatto" del Gestore Utente. Butterfly permette la registrazione di tanti account di contatto diversi quanti sono i sistemi di contatto enumerati in `public.consumer_service`. I sistemi di contatto attualmente supportati sono `TELEGRAM`, `EMAIL` e `SLACK`. La definizione degli account di contatto permette al servizio `middleware-dispatcher` di inoltrare i messaggi degli eventi verso il Consumer corretto, e permette a tali Consumer di sapere a chi inviare tali messaggi.

Per ogni account di contatto è necessario un riferimento di contatto, ovvero un identificativo che contraddistingua l'utente corrente all'interno della piattaforma di contatto scelta. Nel caso l'utente voglia ricevere notifiche su Telegram, il riferimento del sistema di contatto è rappresentato dall'ID della conversazione tra il bot Telegram di Butterfly e l'utente corrente. Nel caso l'utente voglia ricevere notifiche tramite email, il riferimento del sistema di contatto è rappresentato dall'email specificata dall'utente.

Non possono esistere account appartenenti ad utenti diversi in cui la piattaforma e il riferimento di contatto siano uguali. Ad esempio, due utenti non possono avere entrambi un contatto Telegram il cui ID della conversazione con il bot Telegram sia il medesimo. Gli utenti che non hanno ancora alcun sistema di contatto associato non compariranno mai tra i possibili candidati alla ricezione della notifica di un evento di un sistema di produzione. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
user_contact_id	`bigint`	`1`	**PK**
user_id	`bigint`	`1`	**FK** di `public.user.user_id`. ID dell'utente associato all'account di contatto
contact_type	`public.consumer_service`	`TELEGRAM`	Piattaforma di contatto scelta dall'utente per ricevere notifiche di Butterfly
contact_ref	`varchar(30)`	`'38442289'`	Riferimento di contatto per la piattaforma scelta dall'utente

Tabella 6: public.user_contact

9.2.2.6.1 Funzioni associate

Per semplificare la gestione e l'interazione con la tabella `public.user_contact`, il database di Butterfly offre alcune stored function dedicate, presentate di seguito.

Alcune delle funzioni relative agli account di contatto hanno come tipo di ritorno `public.user_contact_denormalized_type`, la cui definizione è la seguente:

```
CREATE TYPE public.user_contact_denormalized_type AS (
  "userContactId" BIGINT,
  "userEmail" text,
  "contactService" public.consumer_service,
  "contactRef" text
);
```

`public.create_user_contact(text, public.consumer_service, text)` è la funzione utilizzata per creare un nuovo account di contatto per un particolare utente, identificato dalla sua email. L'intestazione della funzione è:

```
FUNCTION public.create_user_contact(
  -- Email dell'utente a cui sarà associato il nuovo
  -- account di contatto
  in_user_email text,

  -- Nome della piattaforma di contatto scelta: TELEGRAM,
  -- EMAIL, SLACK
  in_contact_type public.consumer_service,

  -- Identificativo univoco della destinazione del
  -- destinatario associata alla piattaforma
  -- di contatto
  in_contact_ref text
)
RETURNS public.user_contact_denormalized_type
```

`public.delete_user_contact(text, public.consumer_service)` è la funzione utilizzata per eliminare una particolare iscrizione. L'intestazione della funzione è:

```
FUNCTION public.delete_user_contact(
  -- Email dell'utente a cui è associato l'account
  -- di contatto da eliminare
  in_user_email text,

  -- Nome della piattaforma di cui andare ad eliminare
  -- l'account per l'utente corrente: TELEGRAM,
  -- EMAIL, SLACK
  in_contact_type public.consumer_service
)
RETURNS INT
```

9.2.2.7 public.keyword

La tabella `public.keyword` contiene la definizione di tutte le parole chiavi associate alle iscrizioni di ogni utente. Visto che le parole chiavi possono essere molteplici e spesso possono ripetersi (si pensi alle parole chiavi "BUG" e "FIX"), per ogni iscrizione sono salvati solo gli id delle parole chiavi coinvolte. La tabella che determina l'associazione vera e propria, di tipo molti a molti, è `public.x_subscription_keyword`. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
keyword_id	`bigint`	`1`	**PK**
keyword_name	`varchar(15)`	`'BUG'`	Valore della keyword, unico

Tabella 7: public.keyword

9.2.2.8 public.subscription

La tabella `public.subscription` modella l'entità "iscrizione ad un evento" del Gestore Utente. L'iscrizione ad un evento è uno dei concetti chiave di Butterfly, perché è ciò che permette all'utente di essere candidato a ricevere le notifiche di un evento che coinvolge la tipologia di evento e il progetto della subscription corrispondente. Con un'iscrizione, un certo utente esprime la volontà di ricevere notifiche riguardanti una singola tipologia di evento (ad esempio solo notifiche sulla creazione di nuove issue in GitLab, piuttosto che notifiche sull'aggiornamento di una segnalazione in Redmine) che avviene in un determinato progetto. Ogni utente può esprimere molteplici iscrizioni, a patto che ognuna di esse riguardi allo stesso tempo progetti e tipi di evento diversi. Poiché più utenti diversi possono decidere di iscriversi alla stessa tipologia di evento riguardante lo stesso progetto, esiste la possibilità di specificare la priorità dell'utente corrente. Nella ricerca degli utenti a cui inoltrare le notifiche di un particolare evento, Butterfly sceglie sempre gli utenti con la priorità più alta.

Esistono due relazioni molti a molti che non sono direttamente desumibili dalla tabella sottostante. La tabella `public.x_subscription_user_contact` è utilizzata per registrare i molteplici sistemi di contatto in cui l'utente corrente desidera ricevere notifiche, mentre la tabella `public.x_subscription_keyword` è impiegata per il salvataggio delle parole chiavi di interesse per la tipologia di evento e per il progetto coinvolto da ogni subscription.

In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
subscription_id	`bigint`	`1`	**PK**
user_id	`bigint`	`1`	**FK** di `public.user.user_id`. ID dell'utente associato all'iscrizione. Unico insieme a `project_id` e `x_service_event_type_id`
project_id	`bigint`	`1`	**FK** di `public.project.project_id`. ID del progetto associato all'iscrizione. Unico insieme a `user_id` e `x_service_event_type_id`
x_service_event_type_id	`bigint`	`1`	**FK** di `public.x_service_event_type.x_service_event_type_id`. ID del progetto associato all'iscrizione. Unico insieme a `user_id` e `project_id`

Tabella 8 continuata dalla pagina precedente

Nome	Tipo	Esempio	Commento
user_priority	`public.user_priority`	`HIGH`	Priorità dell'utente associato a questa iscrizione. Se un altro utente è registrato alla stessa tipologia di evento e allo stesso progetto, allora solo l'utente disponibile con priorità più alta riceverà il messaggi

Tabella 8: public.subscription

9.2.2.8.1 Funzioni associate

Per semplificare la gestione e l'interazione con la tabella `public.subscription`, che come visto richiede più associazioni molti a molti con diverse tabelle, il database di Butterfly offre delle stored function dedicate, presentate di seguito.

Alcune delle funzioni relative alle iscrizioni a tipologie di evento hanno come tipo di ritorno `public.subscription_denormalized_type`, la cui definizione è la seguente:

```
CREATE TYPE public.subscription_denormalized_type AS (
  "subscriptionId" BIGINT,
  "userEmail" text,
  "projectName" text,
  "eventType" public.service_event_type,
  "userPriority" public.user_priority,
  "contacts" jsonb,
  "keywordList" text[]
);
```

`public.create_subscription(text, text, public.service_event_type, public.user_priority, public.consumer_service[], text[])` è la funzione utilizzata per creare una nuova iscrizione ad una tipologia di evento per un particolare progetto. L'intestazione della funzione è:

```
-- public_create_subscription.sql
FUNCTION public.create_subscription(
  -- Email dell'utente che sta creando una nuova iscrizione
  in_user_email text,

  -- Nome del progetto
  in_project_name text,

  -- Nome della tipologia di evento a cui l'utente si
  -- sta iscrivendo, come GITLAB_COMMIT_CREATED,
  -- GITLAB_ISSUE_CREATED, ...
  in_event_type_key public.service_event_type,

  -- Priorità dell'utente identificato `in_user_email`:
  -- LOW, MEDIUM, HIGH
  in_user_priority public.user_priority,

  -- Lista dei sistemi di contatto in cui l'utente vuole
  -- essere notificato: TELEGRAM, EMAIL, SLACK
  in_contact_type_list public.consumer_service[],

  -- Lista di parole chiave che, se appaiono in un nuovo
  -- evento della tipologia e del progetto specificato da
  -- questa iscrizione, comportano l'invio di una notifica
  -- all'utente corrente usando i sistemi di contatto
  -- specificati
  in_keyword_name_list text[]
)
-- Vista denormalizzata dell'iscrizione appena creata
RETURNS public.subscription_denormalized_type
```

`public.find_subscription(text, text, public.service_event_type)` è la funzione utilizzata per la ricerca di una particolare iscrizione. L'intestazione della funzione è:

```
FUNCTION public.find_subscription(
  -- Email dell'utente
  in_user_email text,

  -- Nome del progetto
  in_project_name text,

  -- Nome della tipologia di evento, come GITLAB_COMMIT_CREATED,
  -- GITLAB_ISSUE_CREATED, ...
  in_event_type_key public.service_event_type
)
-- Vista denormalizzata dell'iscrizione individuata
RETURNS public.subscription_denormalized_type
```

`public.find_subscriptions_by_email(text)` è la funzione utilizzata per la ricerca di tutte le iscrizioni di un particolare utente. L'intestazione della funzione è:

```
-- public.find_subscriptions_by_email.sql
FUNCTION public.find_subscriptions_by_email(
-- Email dell'utente
  in_user_email text
)
-- Vista denormalizzata delle iscrizioni individuate
RETURNS public.subscription_denormalized_type
```

`public.update_subscription(text, text, public.service_event_type, public.user_priority, public.consumer_service[], text[])` è la funzione utilizzata per effettuare aggiornamenti parziali ad un'iscrizione ad una tipologia di evento per un particolare progetto. L'intestazione della funzione è:

```
FUNCTION public.update_subscription(
  -- Email dell'utente che sta creando una nuova iscrizione
  in_user_email text,

  -- Nome del progetto
  in_project_name text,

  -- Nome della tipologia di evento a cui l'utente
  -- si sta iscrivendo, come GITLAB_COMMIT_CREATED,
  --  GITLAB_ISSUE_CREATED, ...
  in_event_type_key public.service_event_type,

  -- Priorità dell'utente identificato 'in_user_email':
  --LOW, MEDIUM, HIGH
  -- Se è NULL, viene mantenuta la prioritàa utente
  -- precedente
  -- Se non è NULL, la prioritàa utente per questa iscrizione
  -- viene sovrascritta con il nuovo valore
  in_user_priority public.user_priority,

  -- Lista dei sistemi di contatto in cui l'utente vuole essere
  -- notificato: TELEGRAM, EMAIL, SLACK
  -- Se è NULL, vengono mantenuti i sistemi di contatto
  -- precedenti.
  -- Se non è NULL, i sistemi di contatto per questa
  -- iscrizioni vengono eliminati e sostituiti con i
  -- nuovi valori
  in_contact_type_list public.consumer_service[],


  -- Lista di parole chiave che, se appaiono in un nuovo
  -- evento della tipologia e del progetto specificato da
  -- questa iscrizione, comportano l'invio di una notifica
  -- all'utente corrente usando i sistemi di contatto
  -- specificati.
  -- Se è NULL, vengono mantenute le parole chiave precedenti.
  -- Se non è NULL, la precedente associazione di parole
  -- chiavi per questa iscrizione viene eliminata e sostituita
  -- con i nuovi valori
  in_keyword_name_list text[]
)
-- Vista denormalizzata dell'iscrizione dopo le modifiche
-- appena effettuate
RETURNS public.subscription_denormalized_type
```

`public.delete_subscription(text, text, public.service_event_type)` è la funzione utilizzata per eliminare una particolare iscrizione. L'intestazione della funzione è:

```
-- public.delete_subscription.sql
FUNCTION public.delete_subscription(
  -- Email dell'utente
  in_user_email text,

  -- Nome del progetto
  in_project_name text,

  -- Nome della tipologia di evento, come
  -- GITLAB_COMMIT_CREATED, GITLAB_ISSUE_CREATED, ...
  in_event_type_key public.service_event_type
)
-- Numero delle entità eliminate, assume valore 1
-- se esisteva un'iscrizione corrispondente ai
-- parametri in input, 0 altrimenti
RETURNS INT
```

9.2.2.9 public.x_subscription_user_contact

La tabella `public.x_subscription_user_contact` rappresenta l'associazione molti a molti tra l'entità "iscrizione ad un evento" rappresentata da `public.subscription` e gli account di contatto dell'utente iscritto, contenuti nella tabella `public.user_contact`. Questa tabella è utilizzata per registrare i molteplici sistemi di contatto in cui l'utente corrente desidera ricevere notifiche. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
subscription_id	`bigint`	1	**PK** insieme a `user_contact_id`
user_contact_id	`varchar(15)`	1	**PK** insieme a `subscription_id`

Tabella 9: public.x_subscription_user_contact

9.2.2.10 public.x_subscription_keyword

La tabella `public.x_subscription_user_contact` rappresenta l'associazione molti a molti tra l'entità "iscrizione ad un evento" rappresentata da `public.subscription` e le parole chiavi scelte dall'utente per una particolare iscrizione, contenute nella tabella `public.keyword`. Questa tabella è impiegata per il salvataggio delle parole chiavi di interesse per la tipologia di evento e per il progetto coinvolto da ogni subscription. Questa tabella è utilizzata per registrare i molteplici sistemi di contatto in cui l'utente corrente desidera ricevere notifiche. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
subscription_id	`bigint`	1	**PK** insieme a `keyword_id`
keyword_id	`bigint`	1	**PK** insieme a `subscription_id`

Tabella 10 continuata dalla pagina precedente

Nome	Tipo	Esempio	Commento

Tabella 10: public.x_subscription_keyword

9.2.2.11 public.event_sent_log

La tabella `public.event_sent_log` è utilizzata per il salvataggio (opzionale) degli eventi ricevuti dai servizi di produzione. Questa tabella fornisce un semplice esempio di come sia possibile espandere il sistema Butterfly storicizzando gli eventi ricevuti, in modo tale da poter permettere l'ottenimento di statistiche nel tempo. La scrittura in tale tabella è regolata dal parametro booleano `in_save_event` dalla stored function `public.search_event_receivers(json, boolean)`, il quale è impostato di default a `false`. In questa tabella, nessun elemento è nullable e non è presente alcun valore di default.

Nome	Tipo	Esempio	Commento
event_sent_log_id	`bigint`	`1`	**PK**
producer_service_key	`public.producer_service`	`1`	**PK** insieme a `subscription_id`
event_record	`json`	`{"projectName": "Butterfly", "eventId": "2", ..., "userEmail": "admin@mail.it"}`	Snapshot in formato JSON dell'evento inoltrato dai servizi di produzione

Tabella 11: public.event_sent_log

10 Struttura messaggi interni

10.1 Descrizione

Il formato utilizzato per lo scambio di messaggi interni tra i microservizi di Butterfly collegati al broker Kafka è Apache Avro. Gli schemi Avro permettono di definire, usando la sintassi JSON, le strutture dati tipizzate inviate dai servizi di produzione e ricevute dai servizi di consumo. Per ogni schema Avro, a tempo di compilazione è generata una classe Java equivalente, che offre sia un modo pratico di creare una nuova istanza di tale classe (tramite Builder Pattern), sia metodi getter e setter per leggere e modificare ogni campo dato definito nello schema. Per un esempio pratico su come creare un oggetto Avro in Java, si rimanda alla relativa sezione 10.2.

Avro supporta i più comuni tipi primitivi di dato, di cui quelli più utilizzati in Butterfly sono:

- `long`: numero intero a 64 bit con segno;
- `string`: sequenza di caratteri Unicode;
- `null`: nessun valore.

La definizione delle strutture dei messaggi scambiati si basa sul tipo complesso Record, che è analogo alla definizione di un oggetto JSON. Avro supporta inoltre le enumerazioni di tipi e alcune semplici collezioni di dati, come gli array, anch'essi utilizzati da Butterfly.

Per una documentazione più esaustiva sui tipi primitivi e complessi supportati da Avro, si rimanda al documento ufficiale della specifica di Apache Avro 1.8.1.

10.2 Esempio creazione oggetto Avro in Java

Lo schema Avro di riferimento di questo esempio è `it.unipd.dstack.butterfly.producer.avro.Event`.

```
import java.util.list;
import it.unipd.dstack.butterfly.producer.avro.Event;
import it.unipd.dstack.butterfly.producer.avro.Services;
import it.unipd.dstack.butterfly.producer.avro.ServiceEventTypes;

// ...
Event.Builder eventBuilder = Event.newBuilder();
eventBuilder.setTimestamp(1557325419);
eventBuilder.setService(Services.GITLAB);
eventBuilder.setProjectName(mergeRequestEvent.getProject().getName
    ());
eventBuilder.setProjectURL(mergeRequestEvent.getProject().
    getWebUrl());
eventBuilder.setEventId("123");
eventBuilder.setEventType(ServiceEventTypes.GITLAB_ISSUE_CREATED);
eventBuilder.setUserEmail("admin@dstack.it");
eventBuilder.setTitle("title");
eventBuilder.setDescription("description");
eventBuilder.setTags(List.of("Bug", "Fix"));
Event event = eventBuilder.build();
```

10.3 Schemi Avro più importanti

Di seguito è presentata la lista degli schemi Avro utilizzati da Butterfly per la propagazione dei messaggi relativi agli eventi di produzione dai producer ai consumer.

10.3.1 Services

Services è un tipo enumerabile Avro che definisce la lista dei servizi di produzione supportati da Butterfly. Il namespace completo di questo schema è `it.unipd.dstack.butterfly.producer.avro.Services`.
I servizi supportati sono:

- REDMINE;
- GITLAB;
- SONARQUBE.

10.3.2 ServiceEventTypes

ServiceEventTypes è un tipo enumerabile Avro che definisce la lista degli eventi supportati da Butterfly. Il namespace completo di questo schema è `it.unipd.dstack.butterfly.producer.avro.ServiceEventTypes`.

Le tipologie di eventi supportati sono:

- GITLAB_COMMIT_CREATED;
- GITLAB_ISSUE_CREATED;
- GITLAB_ISSUE_EDITED;
- GITLAB_MERGE_REQUEST_CREATED;
- GITLAB_MERGE_REQUEST_EDITED;
- GITLAB_MERGE_REQUEST_MERGED;
- GITLAB_MERGE_REQUEST_CLOSED;
- REDMINE_TICKET_CREATED;
- REDMINE_TICKET_EDITED;
- SONARQUBE_PROJECT_ANALYSIS_COMPLETED.

10.3.3 Event

Event è una struttura Avro che contiene tutte le informazioni relative ad un evento prodotto da un sistema di produzione supportato da Butterfly. Il namespace completo di questo schema è `it.unipd.dstack.butterfly.producer.avro.Event`.
Poiché gli eventi prodotti da GitLab hanno una struttura diversa rispetto a quelli prodotti da Redmine e da SonarQube, è necessario fattorizzarli, unificandone la struttura ed estraendone solo le informazioni di effettivo interesse e il cui significato sia comune a prescindere dal sistema di produzione originale. Event, infatti, funge da massimo comune denominatore tra le strutture degli eventi supportati. Solo una

minima parte dei campi di Event è nullable.

I campi dello schema Event sono i seguenti:

Nome	Tipo	Esempio	Commento
timestamp	`long`	`1557325419`	Timestamp relativo all'evento espresso in formato UNIX
service	`Services`	`'GITLAB'`	Nome del servizio di terze parti da cui proviene l'evento originale
projectName	`string`	`'Butterfly'`	Nome del progetto associato con l'evento corrente
projectURL	`string`	`'NULL'`	http://dstack.gitlab.io/butterfly
eventId	`string`	`123`	ID dell'evento corrente
eventType	`ServiceEventTypes`	`GITLAB_ISSUE_CREATED`	Tipologia di evento associata all'evento corrente
userEmail	`string`	`NULL`	`admin@dstack.it`
title	`string`	`'New issue'`	Titolo riassuntivo dell'evento
descripion	`string`	`Some new bugs to fix`	Descrizione estesa dell'evento
tags	`string[]`	`['Bug']`	Lista di etichette associate all'evento, per esempio estratte dalle issue di GitLab o dai tracker di Redmine

Tabella 12: campi schema Event

10.3.4 Contacts

Contacts è un tipo enumerabile Avro che definisce la lista delle piattaforme di contatto supportate da Butterfly. Il namespace completo di questo schema è `it.unipd.dstack.butterfly.consumer.avro.Contacts`.

Le piattaforme di contatto supportate sono:

- TELEGRAM;
- EMAIL;
- SLACK.

10.3.5 UserSingleContact

UserSingleContact è una struttura Avro che contiene una breve anagrafica dell'utente a cui inviare l'evento, la piattaforma di contatto scelta dall'utente per l'inoltro di un evento, e il riferimento di contatto per identificare univocamente il destinatario all'interno del sistema di contatto scelto. Nel caso in cui la piattaforma di contatto scelta dall'utente sia `EMAIL`, il riferimento di contatto è l'indirizzo email specificato dall'utente, che non è necessariamente lo stesso usato per registrarsi. Nel caso in cui la piattaforma di contatto scelta sia `TELEGRAM`, il riferimento di contatto è l'id della conversazione tra il Bot di Telegram di Butterfly e l'utente, impostato al momento della configurazione dell'account di contatto di Telegram (ulteriori informazioni disponibili nella sezione 7.2). Infine, se la piattaforme di contatto scelta è SLACK, il riferimento di contatto è l'id dell'utente Slack che ha configurato l'account di contatto Butterfly per questa piattaforma (ulteriori informazioni disponibili nella sezione 7.3.0.1). Il namespace completo di questo schema è `it.unipd.dstack.butterfly.consumer.avro.UserSingleContact`.

I campi dello schema UserSingleContact sono i seguenti:

Nome	Tipo	Esempio	Commento
firstname	`string`	`Davide`	Nome dell'utente che deve ricevere una notifica dell'evento corrente
lastname	`string`	`Zanetti`	Cognome dell'utente che deve ricevere una notifica dell'evento corrente
contact	`Contacts`	`'Slack'`	Piattaforma di contatto scelta dall'utente destinatario per ricevere una notifica dell'evento corrente
contactRef	`string`	`'DUHV896A'`	Riferimento di contatto dell'utente destinatario per la piattaforma di contatto scelta

Tabella 13: campi schema UserSingleContacts

10.3.6 EventWithUserContact

EventWithUserContact è una struttura Avro che aggiunge le informazioni relative al destinatario del messaggio all'evento originale da inviare. Il namespace completo di questo schema è `it.unipd.dstack.butterfly.consumer.avro.EventWithUserContact`. EventWithUserContact è una composizione di altri due record Avro, `event` e `userContact`.

I campi dello schema EventWithUserContact sono i seguenti:

Nome	Tipo	Esempio	Commento
event	`Event`	`{ "timestamp": 1557325419, "service": "GITLAB", "projectName": "Butterfly", "projectURL": "http://dstack.gitlab.io/butterfly", "eventId": "123", "eventType": "GITLAB_ISSUE_CREATED", "userEmail": "admin@dstack.it", "title": "New issue", "description": "Some new bugs to fix", "tags": ["Bug"] }`	Record che definisce una struttura uniformata per tutti gli eventi di produzione possibili, estraendo le informazioni più rilevanti dall'evento originale
userContact	`UserSingleContact`	`{ "firstname": "Davide", "lastname": "Zanetti", "contact": "SLACK", "contactRef": "DUHV896A" }`	Record che contiene le informazioni base dell'utente destinatario, la piattaforma di contatto scelta e un identificativo univoco di contatto

Tabella 14: campi schema EventWithUserContact

11 Gestione task di sviluppo

Poiché Butterfly è frammentato in molteplici microservizi, il team DStack ha creato uno script bash per automatizzare le operazioni più comuni, ovvero compilare, eseguire, spegnere e testare Butterfly. Ogni task eseguito dallo script richiede solo che ci siano Docker e Docker Compose installati, e che il terminale di esecuzione sia in grado di interpretare correttamente i comandi della Bash shell. Lo script risiede nel file *./run.sh*, e accetta solo due opzioni:

- `-b | --build`: crea dei nuovi file Jar a partire dal codice sorgente Java nella cartella *./butterfly*;
- `-h | --help`: mostra il manuale del comando in inglese.

Di seguito sono presentati i comandi disponibili, da inserire obbligatoriamente dopo le opzioni, che invece sono facoltative.

- **install**: installa le dipendenze di terze parti dei servizi Java e dei servizi scritti in Node.js. Grazie all'utilizzo dei volumi Docker, le dipendenze sono salvate in una cache su disco fisico;
- **prod**: esegue tutti i servizi in modalità "production", ovvero non istanzia i servizi utili solo allo sviluppo;
- **dev**: esegue tutti i servizi in modalità sviluppatore, lanciando anche strumenti di utilità quali PgAdmin e OpenAPI;
- **prune**: ferma tutti i servizi di Butterfly attualmente in esecuzione, elimina tutte le sottoreti virtuali create da Docker, e rimuove tutti i volumi di Docker (cancellando quindi i file salvati di Butterfly);
- **stop**: ferma l'esecuzione di tutti i servizi del prodotto;
- **ps**: visualizza la lista dei processi di Butterfly in esecuzione e ne verifica lo stato;
- **logs**: mostra i log per tutti i servizi di Butterfly in esecuzione;
- **test**: esegue tutti i test di unità e d'integrazione di Butterfly.
- **sonarcloud**: raccoglie le metriche relative ai test Java e le inoltre a SonarCloud per l'analisi statica.

Ecco il risultato di `./run.sh --help`:

```
Usage:
  ./run.sh [OPTIONS] COMMAND

A single entry point for building, running, stopping and testing
   Butterfly.

Options:
-b|--build            If specified, it creates the Java
                      services' jar executables.
-h|--help             Show this help.

Commands:
```

```
install                 Install Java services' and User Manager'
                        dependencies.
prod                    Executes the services in production mode.
dev                     Executes the services in development mode
                        and spins up utility services too.
                        The utility services are:
                        - pgadmin [localhost:8080];
                        - openapi [localhost:8082].
prune                   Deletes all the docker virtual networks,
                        volumes and stops the services if
                        they're currently running.
stop                    Stops the services if they're running.
ps                      Shows the list of services running.
logs                    Fetches the logs for a service.
test                    Executes the whole test battery.
sonarcloud              Collects code metrics and uploads them on
    SonarCloud.
```

L'esecuzione di `./run.sh prod` esegue tutti i servizi necessari al corretto funzionamento di Butterfly, di cui i seguenti espongono porte TCP all'host (alla macchina in cui sta eseguendo Docker):

- **GitLab Producer**: porta 3000;
- **Redmine Producer**: porta 4000;
- **Sonarqube Producer**: porta 6000;
- **User Manager**: porta 5000;
- **Configuratore Account Slack**: porta 5200.

I seguenti servizi, inoltre, sono lanciati quando `./run.sh dev` viene eseguito:

- **OpenAPI**: porta 8082;
- **PgAdmin**: porta 8080.

In particolare, PgAdmin è configuration con le seguenti credenziali:

- *Nome utente*: user@domain.com;
- *Password*: secret;
- *Database*: butterfly_user;
- *Password Database*: butterfly_user.

12 Ricerca utenti interessati alle notifiche di un dato evento

Di seguito sono illustrati i passaggi di cui è composta la stored procedure `public.search_event_receivers(json, boolean = false)` che dato un oggetto Event in formato JSON, ritorna la lista di utenti che sono interessati a ricevere le notifiche dell'evento corrente nelle piattaforme di contatto da loro specificate, ovvero quegli utenti iscritti alla stessa tipologia di evento e al progetto citati dall'oggetto JSON

in input, e che hanno specificato una lista di parole chiave che compare nel titolo, nella descrizione o nelle etichette associate all'evento corrente. Le informazioni di contatto ritornate per ogni utente sono presentate in una struttura a mappa chiave valore, in cui le chiavi rappresentano le piattaforme di contatto supportate in cui di cui gli utenti relativi hanno specificato l'account di contatto relativo (TELEGRAM, EMAIL, SLACK), e in cui i valori sono i riferimenti di contatto relativi. Opzionalmente è inoltre possibile passare alla procedura di ricerca un parametro booleano, `in_save_event`, che se assume valore `true` salva l'evento corrente nella tabella `public.event_sent_log` a scopo di storicizzare gli eventi processati dal Gestore Utente.

12.1 Esempio di evento

```
{
  "service": "GITLAB",
  "projectName": "Butterfly test",
  "projectURL": "https://localhost:10443/dstack/butterfly",
  "eventId": "1",
  "eventType": "GITLAB_ISSUE_CREATED",
  "userEmail": null,
  "title": "New bug to fix and test",
  "description": "Breaking issue that affects the Middleware Dispatcher.",
  "tags": ["BUG", "FIX", "TESTING"]
}
```

12.2 Esempio di invocazione query

Si assuma che, nella seguente query, *${event}* faccia riferimento all'oggetto JSON usato nell'esempio di evento precedente.

12.2.1 Query

```
SELECT "userContactList"
FROM public.search_event_receivers(\${event}::json);
```

12.2.2 Esempio di risposta

```
[
  {
    "firstname": "Jon",
    "lastname": "Snow",
    "contactInfo": {
    "TELEGRAM": "120923823",
    "EMAIL": "jon.snow@got.co.uk"
    }
  },
  {
```

```
    "firstname": "Daenerys",
    "lastname": "Targeryen",
    "contactInfo": {
      "SLACK": "DHV89016"
    }
  }
]
```

12.2.3 Passaggi della procedura di ricerca

1. Individuazione del progetto associato all'evento in base all'URL o al nome del progetto;
2. Estrazione dell'identificativo della tipologia di evento;
3. Individuazione della lista di iscrizione correlate al progetto e alla tipologia di evento del JSON in ingresso;
4. Estrazione della lista di parole chiave distinte che sono associate alle iscrizione individuate al punto precedente;
5. Per ogni parola chiave individuata al punto precedente, estrae la lista di utenti che l'hanno scelta in una delle *subscriptions*;
6. Le parole chiave individuate vengono filtrate in base al match (*case insensitive*) su titolo, descrizione ed etichette dell'evento corrente;
7. Esclusione degli utenti che non hanno alcuna keyword in comune con i match individuati. Per ogni utente rimasto viene generato un contatore del numero di parole chiave scelte da quell'utente citate dall'evento corrente;
8. La lista di utenti viene ordinata per priorità ($HIGH > MEDIUM > LOW$) e per numero decrescente di match delle parole chiave (ovvero il contatore del punto precedente);
9. Per ogni utente filtrato vengono estratte le informazioni di contatto: nome, cognome, piattaforme di contatto scelte nell'iscrizione relativa alla tipologia di evento e progetto correnti e i relativi riferimenti di contatto.

13 Gestore Personale

Tutti i percorsi di file riferiti nella sezione corrente saranno intesi come relativi al path *./user-manager/user-manager-rest-api/src*.

13.1 Interfacciamento al Database

La definizione di tutti i metodi che definiscono le varie policy di interfacciamento al Database del Gestore Personale risiedono nell'interfaccia `DatabaseConnection`, definita nel file *./database/DatabaseConnection.ts*. Quest'interfaccia impone che sia possibile definire dei segnaposti all'interno delle query, e che tali segnaposti possano essere sostituiti a tempo d'esecuzione con dei valori effettivi. La struttura che contiene i valori effettivi è `DatabaseConnectionValues`, un oggetto chiave valore in cui la

chiave è il nome del placeholder all'interno della query, e in cui il valore è l'effettivo valore da sostituire.
Tutti i metodi di `DatabaseConnection` sono asincroni, e la loro segnatura è la seguente:

- `none(query: string, values?: DatabaseConnectionValues): Promise<void>`: esegue una query che non si aspetta alcun risultato di ritorno. Se per caso la query ritorna dei risultati, deve essere lanciata un'eccezione;
- `one<T>(query: string, values?: DatabaseConnectionValues): Promise<T|null>`: esegue una query che si aspetta esattamente un risultato. Se nessuna riga è restituita dal database, ritorna un valore nullo. Se più di una riga è restituita dal database, deve essere lanciata un'eccezione;
- `any<T>(query: string, values?: DatabaseConnectionValues): Promise<T[]>`: esegue una query che si aspetta un numero imprecisato di risultati, e li ritorna;
- `result(query: string, values?: DatabaseConnectionValues): Promise<number>`: esegue una query e ritorna il numero di righe coinvolte da tale operazione;
- `close(): Promise<void>`: chiude la connessione al database.

L'implementazione correntemente usata di `DatabaseConnection` è `PgDatabaseConnection`, definita nel file *./database/DatabaseConnection.ts* e che usa la libreria `pg-promise` per interfacciarsi con il database di tipo **Postgres** impiegato dallo User Manager. `pg-promise` richiede che i placeholder usati all'interno dei file di query siano definiti in uno dei seguenti modi:

- **${NOME_CHIAVE}** (preferibile)
- *$[NOME_CHIAVE]*

Per convenzione, tutti i segnaposto all'interno delle query dello User Manager seguono la prima modalità. Segue un esempio di esecuzione di una query SQL rivolta ad un database di tipo Postgres con sostituzione dei valori placeholder.
Si assuma che la stringa `query` contenga il seguente testo:

```
SELECT public.delete_subscription(
    ${userEmail},
    ${projectName},
    ${eventType}
) AS "count";
```

Si assuma inoltre che l'oggetto `values` abbia il seguente contenuto, con le chiavi combacianti ai placeholder impiegati nella query precedente:

```
{
  "userEmail": "jon.snow@got.co.uk",
  "projectName": "Butterfly",
  "eventType": "GITLAB_ISSUE_CREATED"
}
```

Per effettuare la sostituzione, basta lanciare uno qualsiasi dei metodi di DatabaseConnection (tranne ovviamente il metodo close):

```
databaseConnection.result(query, values) // Promise<number>
```

13.2 Router

Ogni file di tipo Router contiene una funzione che una volta invocata ritorna la definizione delle rotte HTTP per un certo modulo di Butterfly. Ogni modulo fa riferimento ad una particolare entità del database. Ad esempio, il file *./modules/user/router.ts* elenca tutte le rotte HTTP per la gestione CRUD dell'entità "Utente", e specifica quale metodo del controller deve essere invocato. È compito di ogni router istanziare le istanzi delle classi Repository, Manager e Controller appropriate. Nel caso si voglia aggiungere il supporto per nuovi endpoint, o sia necessario modificare gli endpoint esistenti, è opportuno partire dal file `router.ts`.

13.3 Entity

Ogni file entity.ts contiene una serie di definizioni di tipi e interfacce usate dalle classi Controller per tipizzare gli input letti dalle richieste HTTP. Non contengono nulla a livello di logica.

13.4 Controller

Nell'ambito del Gestore Personale, un Controller è una classe che determina quali parametri sono letti dall'endpoint, dalle querystring e dal payload della richiesta HTTP associata ad un particolare metodo. È inoltre compito del Controller invocare le corrette funzioni di validazione dei parametri letti dalle richieste HTTP. Le classi di questo tipo sono contenute all'interno del file controller.ts. L'istanza della classe di tipo "*Manager*" usata dal controller è iniettata nel costruttore tramite Dependency Injection. Ogni Controller espone delle funzionalità di interazione con il database ad un livello astratto, affindandosi ad una classe di tipo Manager per operazioni di lettura, creazione, aggiornamento e cancellazione. Un altro compito di ogni classe Controller è definire il codice HTTP di ritorno che, se non specificato, è di default 200 (HTTP OK). Ogni metodo pubblico esposto dalle classi controller è di tipo Middleware, e utilizza il metodo `execute<T>(RouteCommand<T>)` definito nella classe astratta padre `RouteController`. I comandi passati a tale funzione sono definiti come metodi privati della classe Controller. Il metodo `execute<T>(RouteCommand<T>)` prende il risultato di tali comandi e li usa per rispondere alla richiesta HTTP. Questo processo è indipendente dal framework HTTP utilizzato grazie all'impiego di una `RouteContextReplierFactory`, che ritorna un'istanza di `RouteContextReplier`, che astrae la logica di risposta del server HTTP.

13.5 Manager

Le classi di tipo Manager hanno il compito di inoltrare le richieste delle classi Controller alle classi Repository, che hanno piena conoscenza del database di riferimento e di come lanciare query ad esso. Eventuali errori nelle risposte possono essere gestiti dalle classi Manager, senza propagarli ai Controller. Dal momento che molte delle operazioni richieste dalle REST API di Butterfly sono ripetitive (pur avendo come target entità diversi, quasi ogni modulo richiede operazioni CRUD), è stata creata la classe astratta `AbstractCRUDManager`, che definisce le operazioni:

- `create<P, R>(params: P): Promise<R>`: crea una nuova risorsa in maniera asincrona e ne ritorna il contenuto. La segnatura di questo metodo è definita dall'interfaccia `Write`;

- `update<P, R>(params: P): Promise<R|null>`: aggiorna una risorsa esistente in maniera asincrona e ne ritorna il contenuto modificato. Se la risorsa da modificare non viene trovata, ritorna un valore nullo. La segnatura di questo metodo è definita dall'interfaccia `Write`;
- `find<P, R>(params?: P): Promise<R[]>`: ritorna in maniera asincrona una lista di risorse in base ai parametri opzionali in ingresso; La segnatura di questo metodo è definita dall'interfaccia `Read`;
- `findOne<P, R>(params: P): Promise<R|null>`: cerca in maniera asincrona una singola risorsa a partire dai parametri in ingresso. Se la risorsa cercata non può essere trovata, ritorna un valore nullo. La segnatura di questo metodo è definita dall'interfaccia `Write`;
- `delete<P>(params: P): Promise<boolean>`: cerca di cancellare in maniera asincrona la risorsa definita dall'oggetto dei parametri in ingresso. Se l'operazione ha successo ritorna *true*, altrimenti ritorna *false*. La segnatura di questo metodo è definita dall'interfaccia `Delete`;

13.6 Repository

Le classi di tipo Repository sono le uniche a comunicare direttamente con il database grazie ad un riferimento ad un oggetto di tipo `Database`, passato tramite Dependency Injection al costruttore. Le classi Repository richiedono che ogni query da lanciare sia definito all'interno di una file SQL separato. La convenzione impiegata nel Gestore Utente richiede che tali file SQL siano contenuti all'interno di *./modules/${entity}/sql/*.sql*, dove entity è il nome dell'entità modellata dal modulo corrente (ad esempio, la definizione delle query impiegate dal modulo `subscriptions`, quello relativo alle iscrizioni, si trova nella cartella *./modules/subscriptions/sql*). Ognuno di tali file dovrebbe avere lo stesso nome dei metodi di AbstractCRUDManager (`create.sql` per la creazione di una nuova risorsa, `update.sql` per l'aggiornamento della stessa, e via dicendo). È compito delle classi di tipo "QueryProvider" definire la mappatura tra il nome di un'operazione e il file con la definizione della query. Queste classi sono passate tramite Dependency Injection ai costruttori delle classi di tipo Repository. Le query definite nei file SQL sono lette solo una volta nella fase di avvio del server, e mantenute in memoria per tutti i successivi accessi (in modo da evitare l'overhead di effettuare una lettura su disco fisso ogni volta che il server del Gestore Personale deve rispondere ad una richiesta HTTP).

13.7 Validazione

È fortemente consigliato validare tutto l'input proveniente dall'esterno, soprattutto nel caso di lettura da richieste HTTP. Il processo di validazione consiste nell'asserire che i parametri in ingresso debbano avere una certa struttura e debbano rispettare determinati vincoli. Il successo della validazione è indispensabile per il proseguimento dell'intento espresso dall'utente (l'esecuzione di una certa query deve avvenire solo se l'input è stato validato con successo). Se quest'asserzione fallisce a runtime, il Gestore Personale deve costruire un errore di validazione che va mostrato all'utente per indicargli i motivi per cui la richiesta HTTP precedente è stata rifiutata. La libreria di validazione attualmente utilizzata dallo User Manager è *Joi*. La sua istanza statica nominata `Validator` è accessibile dal file *./common/Validation.ts*. All'interno dei singoli moduli, all'interno del file *./modules/${entity}/validator.ts*, sono definite le

funzioni utilizzate dalle classi Controller per validare l'input in ingresso. Un esempio di validazione è la seguente:

```
// Lunghezza massima del nome di un progetto
const maxProjectNameLength = 50;

const keywordsLength = {
  max: 5, // Massimo numero di parole chiavi per una singola
      iscrizione
  min: 1, // Minimo numero di parole chiavi per una singola
      iscrizione
};

const contactServicesLength = {
  max: 3, // Massimo numero di piattaforme di contatto per una
      singola iscrizione
  min: 1, // Minimo numero di piattaforme di contatto per una
      singola iscrizione
};

const validateBody = (validator: ValidatorObject) => ({
  // `contactServices` e' un array di stringhe di dimensione 1 <=
      x <= 3.
  // Tali stringhe possono essere una tra 'TELEGRAM', 'EMAIL' e '
      SLACK'.
  contactServices: validator.array()
    .min(contactServicesLength.min)
    .max(contactServicesLength.max)
    .items(validator.string()
    .valid('TELEGRAM', 'EMAIL', 'SLACK'),
    ),
  // `keywords` e' un array di stringhe di dimensione 1 <= x <= 5
  keywords: validator.array()
   .min(keywordsLength.min)
   .max(keywordsLength.max)
   .items(validator.string()),
  // `userPriority` e' una stringa che puo' assumere un valore tra
       'HIGH', 'MEDIUM' e 'LOW'.
  userPriority: validator.string()
   .valid(['HIGH', 'MEDIUM', 'LOW']),
});
```

Un esempio di oggetto che passerebbe questa validazione è il seguente:

```
{
  "contactServices": ["TELEGRAM", "EMAIL"],
  "keywords": ["BUG"],
  "userPriority": "HIGH"
}
```

14 Estensione delle funzionalità

14.1 Editor consigliati

Per effettuare le modifiche al codice, consigliamo l'adozione dei seguenti editor:

- **IntelliJ IDEA**: editor usato per la scrittura dei file Java. Consente il supporto di Maven, attraverso il quale è possibile importare direttamente le dipendenze sul codice;
- **Microsoft Visual Studio Code**: usato per la scrittura dei file Typescript, SQL e file di configurazione per le variabili d'ambiente.

14.2 Cose possibili da modificare/aggiungere

É possibili estendere le funzionalità di Butterfly aggiungiendo nuove implementazioni di Producer e Consumer:

- Per quanto concerne l'aggiunta di un nuovo Consumer occore creare un nuovo package in cui implementare una nuova classe di Consumer che estende la classe ConsumerController, presente nel package `it.unipd.dstack.butterfly.consumer.consumer.controller`. Inoltre, è vantaggioso continuare a "passare" tramite Dependency Injection tutte le dipendenze necessarie al corretto funzionamento del nuovo Consumer.
- Per quanto concerne l'aggiunta di un nuovo Producer, il discorso è analogo, occore creare un nuovo package in cui implementare una nuova classe di Producer che estende la classe ProducerController, presente nel package `it.unipd.dstack.butterfly.producer.producer.controller`, continuando a utilizzare la Dependency Injection.

15 Test

La stesura dei test statici e dinamici del codice sorgente di Butterfly è attualmente in fase implementativa, è sarà terminata da parte del gruppo entro la Revisione d'Ammissione. Di seguito andiamo a presentare l'organizzazione dei test esistenti suddivisi in due grossi blocchi, come due sono i linguaggi utilizzati all'interno del progetto: *Test Gestore Personale* e *Test Servizi Java*.

15.1 Test Gestore Personale

15.1.1 Test d'unità

Per quanto concerne lo *User Manager*, la suite di test d'unità è gestita con lo strumento Jest. Il comando da lanciare per eseguire il test d'unità è volutamente separato dal comando per eseguire il test d'integrazione. Infatti, una delle caratteristiche principali dei test d'unità è che, oltre ad essere molto compatti e riguardare solo un metodo per volta, devono anche essere molto performanti da eseguire. I test d'integrazione, al contrario, richiedono dei flussi di esecuzione più complessi che introducono latenze. I test d'unità per il *Gestore Personale* vengono eseguiti nei seguenti casi:

- Manualmente, lanciando il comando `yarn test:unit`;
- Durante le fasi preliminari dell'istanziazione del servizio tramite Docker, lanciando il comando `docker-compose up -build` dalla cartella "user-manager";
- Contestualmente all'esecuzione dell'intera suite di test, lanciando lo script `./run.sh test` dalla cartella root del progetto;
- Pubblicando un nuovo commit, così da far partire la Continuous Integration tramite Travis CI.

15.1.2 Test d'integrazione

I test d'integrazione (o *integration tests* in inglese) per lo *User Manager* sono gestiti dallo strumento Jest per l'esecuzione dei metodi veri e propri, a cui è accostata la libreria di terze parti supertest. Quest'ultima è usata per simulare richieste HTTP all'applicazione server del *Gestore Personale*. Le integrazioni da verificare nel Gestore Utenti sono le seguenti:

- Il codice di stato HTTP ritornato come risposta da una richiesta deve essere coerente con il successo o il fallimento dell'operazione richiesta;
- Il servizio Node.js deve essere in grado di comunicare con il database relazionale per effettuare operazioni di ricerca, creazione, modifica e cancellazione;
- I parametri e i payload JSON inviati agli endpoint delle REST API esposte dallo User Manager devono essere validati correttamente;
- I risultati ritornati dalle API devono essere coerenti con i dati inseriti a database: è quindi necessario garantire che il Gestore Personale sappia esattamente il contenuto del database.

Quest'ultimo punto è garantito in quanto viene applicato il seguente flusso di operazioni per ogni test d'integrazione:

- All'inizio di ogni test Jest crea e istanzia un nuovo server, rappresentato dalla classe `Server` del modulo `http` della libreria standard di Node.js;
- Tale server si occupa di instaurare una connessione al database configurato, che è diverso dal database di produzione;
- Su tale database Jest invoca `public.trunc_data()`, una stored procedure SQL che elimina tutte le righe delle colonne definite dall'utente, preservando solo la definizioni di tabelle, viste, tipi custom, procedure e funzioni;
- All'interno dei singoli test vengono invocate query di creazione al database utilizzando recordi noti, e dunque sfruttabili per le asserzioni;
- Tramite *supertest* all'interno del test viene verificato che la risposta dell'endpoint sotto test sia coerente con i dati appena scritti a database, che per le precondizioni precedenti sono anche gli unici dati salvati in quel momento;
- Oltre alla struttura della risposta del database, sempre tramite *supertest* viene verificato il codice HTTP ritornato, e il test termina;
- Infine, Jest spegne il server, il quale a sua volta rilascia la connesione con il database.

Questo continuo processo di istanziazione server, asserzioni e terminazione server da un lato causa un allungamento dei tempi di esecuzione dei test d'integrazione rispetto ai test d'unità, dall'altro garantisce che l'integrazione tra applicativo server e database sia corretta e coerente con lo stato creato ad hoc da ogni test. In altre parole, i test d'integrazione creati nel gestore personale sono riproducibili al 100%, e non hanno alcuna dipendenza sull'ordine di esecuzione dei test stessi. Dal momento che i test d'integrazione del Gestore Personale richiedono l'esecuzione contemporanea di più servizi, il gruppo *DStack* ha semplificato il procedimento creando lo script `./test.sh` dalla cartella "user-manager". Tale script, che al suo interno sfrutta Docker Compose, fa le seguenti operazioni:

- Crea un database con la medesima definizione di schema (definita nel file "ddl.sql" all'interno della cartella "user-manager-database") del database "di produzione";
- Importa il codice sorgente del progetto "user-manager-rest-api";
- Installa le dipendenze di terze parti in un ambiente virtualizzato da Docker diverso da quello di esecuzione del servizio "in produzione";
- Esegue il comando `yarn test:ci` che, dopo aver effettuato operazioni di analisi statica e compilazione del codice, esegue tutti i test di integrazione e di unità.

La possibilità di effettuare operazioni all'inizio e alla fine di ogni test lanciato all'interno di un dato file è offerta dai seguenti due metodi di Jest:

1. `beforeEach(done: () => void): void`: tutto ciò che è definito all'interno di questo metodo viene eseguito appena prima dell'esecuzione di ogni singolo test;

2. `afterEach(done: () => void): void`: analogamente a sopra, tutto ciò che è definito all'interno di questo metodo viene eseguito appena dopo la conclusione di ogni singolo test.

I test d'integrazione per il *Gestore Personale* possono essere eseguiti nei seguenti modi:

- Contestualmente all'esecuzione dell'intera suite di test, lanciando lo script `./run.sh test` dalla cartella root del progetto;
- Pubblicando un nuovo commit, così da far partire la Continuous Integration tramite Travis CI.

15.1.3 Analisi Statica

Per quanto concerne l'analisi statica del codice dello *User Manager*, il gruppo *DStack* ha utilizzato lo strumento tslint. Esso permette di verificare che il codice TypeScript scritto sia conforme alle regole della community di Microsoft, che sono le stess adottate in questo modulo del progetto. Il suo file di configurazione è "tslint.json", collocato nella cartella "user-manager/user-manager-rest-api". È possibile eseguire l'analisi statica del codice in modi diversi, a seconda del contesto:

- Manualmente, lanciando il comando `yarn lint` dalla cartella del progetto "user-manager-rest-api";
- Assieme al comando di traspilazione del codice TypeScript in codice JavaScript, lanciando `yarn build` dalla cartella del progetto "user-manager-rest-api";
- Durante le fasi preliminari dell'istanziazione del servizio tramite Docker, lanciando il comando `docker-compose up -build` dalla cartella "user-manager";
- Contestualmente all'esecuzione dell'intera suite di test, lanciando lo script `./run.sh test` dalla cartella root del progetto;
- Pubblicando un nuovo commit, così da far partire la Continuous Integration tramite Travis CI.

15.2 Test Servizi Java

15.2.1 Test d'unità

JUnit 5 è lo strumento individuato per eseguire l'analisi dinamica del codice dei servizi Java. Il comando da lanciare per eseguire il test d'unità è volutamente separato dal comando per eseguire il test d'integrazione, infatti, i test d'integrazione richiedono dei flussi di esecuzione più complessi che introducono latenze. I test d'unità vengono eseguiti quando viene lanciato il comando `mvn test`.

15.2.2 Test d'integrazione

JUnit 5 è lo strumento individuato per eseguire i test d' integrazione tramite il comando `mvn integration-test`. Tutti i test di integrazione hanno come precondizione che venga lanciata una nuova istanza del broker kafka eseguendo da terminale il comando `docker-compose -f docker-compose-test.yml up`, questo ci assicura una separazione netta tra l'ambiente di produzione e quello di integrazione. Tutti i test adottano il seguente flusso di operazioni:

- Viene avviato il servizio da testare (*Producer* oppure *Consumer*) dichiarato nel file `docker-compose-test.yml`;
- Il servizio è istanziato con una configurazione specifica per i test, identica a quella di produzione, che differisce soltanto sul numero di porta sulla quale questo viene avviato.
- Nel caso venga testato il *Producer*, si inserisce in *Kafka* un messaggio predefinito il quale verrà consumato "al volo" da un *Consumer* Mock che ne verifica l'integrità tramite `Assert`.
- Nel caso venga testato il *Consumer*, si inserisce in *Kafka* un messaggio predefinito tramite un *Producer* Mock creato "al volo", infine viene verificato che il messaggio consumato sia esattamente quello inserito tramite `Assert`.
- Infine viene terminata l'esecuzione dell' ambiente di integrazione.

15.2.3 Analisi Statica

Checkstyle è lo strumento scelto per eseguire l'analisi statica del codice dei servizi Java. Tutti i sottomoduli Java condividono infatti lo stesso file di definizione delle regole di analisi statica, ovvero il file "checkstyle.xml" presente all'interno della cartella "butterfly". Per verificare che le convenzioni di siano verificate, è sufficiente lanciare il comando `mvn checkstyle:check`.

A Importazione del progetto sugli editor consigliati

A.1 Importare su IntelliJ IDEA

Per evitare errori nell'importazione del progetto, viene illustrata una veloce guida per importate correttamente il progetto Butterfly su intelliJ IDEA.

1. Aprire l'IDE IntelliJ IDEA;
2. Una volta visualizzata la schermata iniziale, cliccare l'opzione *Import Project*;

Figura 22: Finestra di avvio di IntelliJ IDEA

3. Selezionare il percorso fino alla directory "butterfly" e confermare la scelta;
4. Spuntare la voce *Import project from external model*, selezionare *Maven* e confermare la scelta;

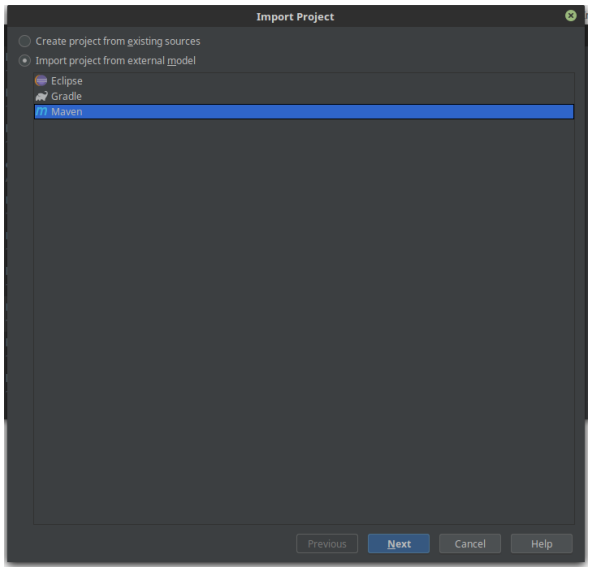

Figura 23: Finestra di importo da un modello esistente

5. Selezionare le voci *Import Maven projects automatically* e *Create module groups for multi-module Maven projects*;

6. Sempre nella stessa finestra, selezionare dalla combo box "Generated source folders" la voce *target/generated-sources*. Successivamente confermare la scelta;

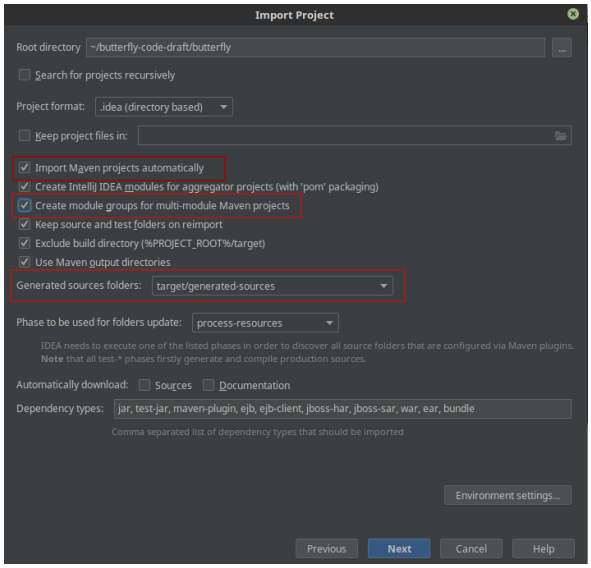

Figura 24: Finestra opzioni importo

7. Selezionare il progetto riportato e confermare la scelta;

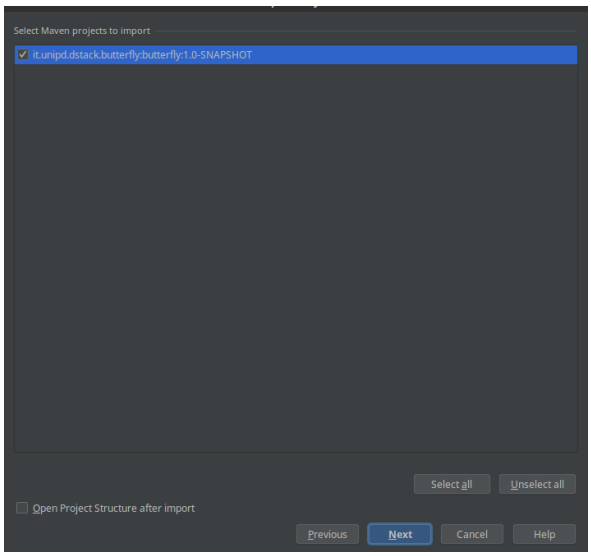

Figura 25: Finestra selezione progetto

8. Selezionare la versione 11 di Java e confermare la scelta;

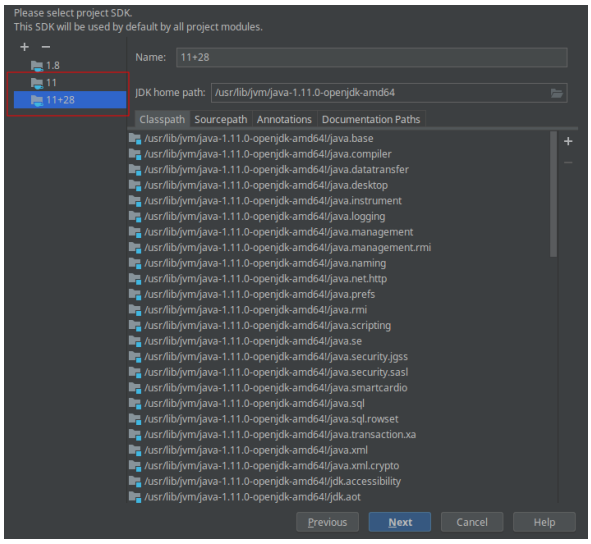

Figura 26: Finestra selezione SDK

9. Scegliere il nome del progetto e la posizione desiderata e successivamente confermare la scelta.

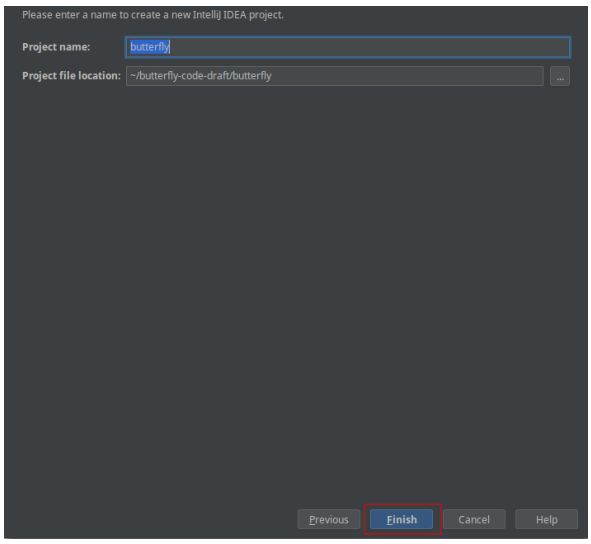

Figura 27: Finestra selezione percorso

B Creazione Webhook

B.1 Webhook Gitlab

Viene descritto come aggiungere un webhook al progetto di Gitlab:

- Accedere al servizio Gitlab;
- Accedere ad *Admin area* (Simbolo chiave inglese);

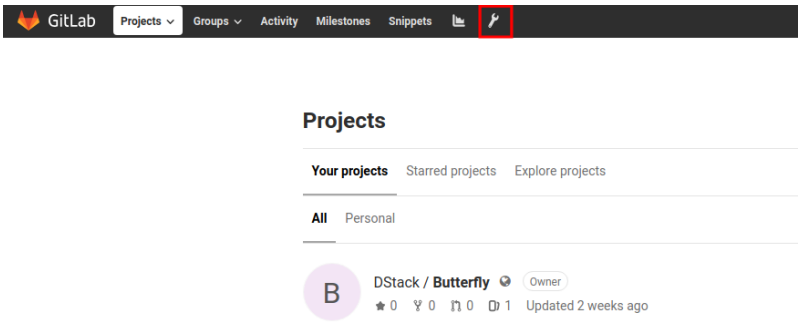

Figura 28: Selezione Admin Area

- Accedere al menu *Setting* » *Network*;

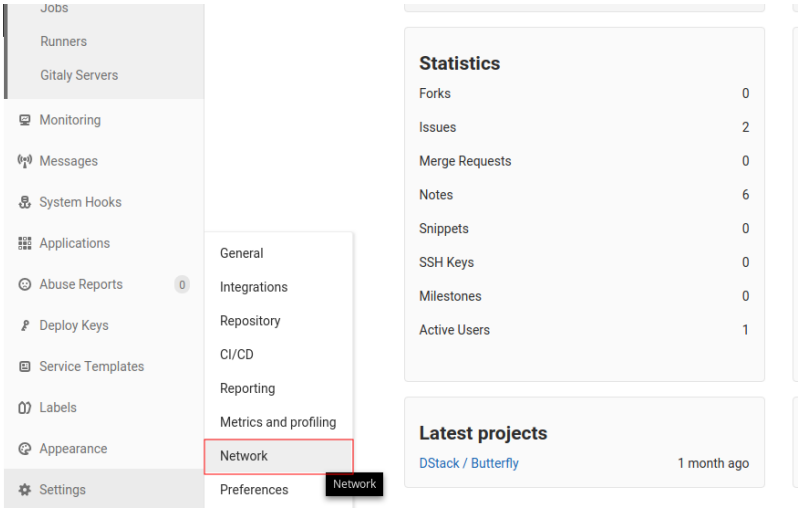

Figura 29: Accesso alle impostazioni del network

- Sotto la voce *Outbound request* spuntare la voce *Allow requests to the local network from hooks and services*;

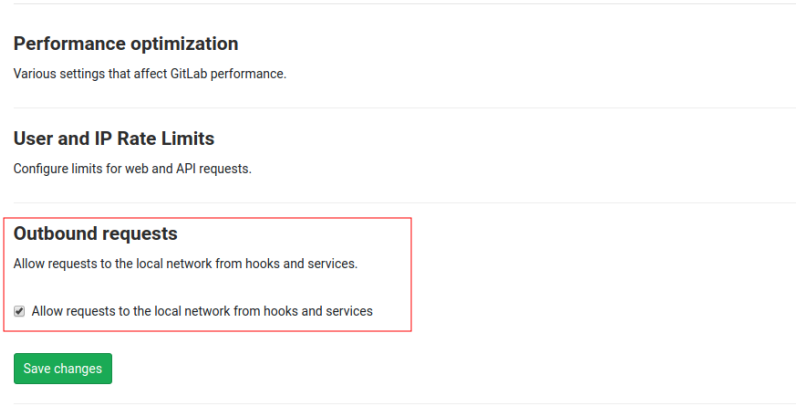

Figura 30: Spunta sull'impostazione

- Spostarti sul progetto da collegare;
- Selezionare *Settings* » *Integrations*;

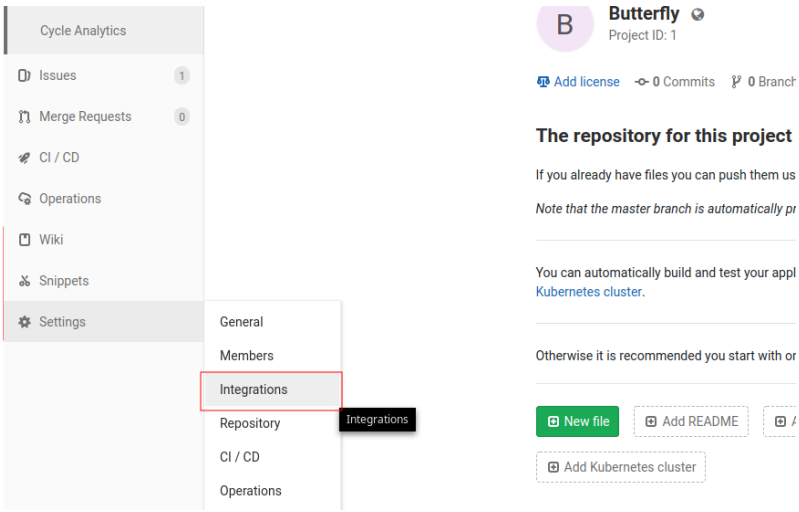

Figura 31: Selezione percorso webhook

- Inserire l'URL del webhook `http://gitlab-producer:3000/webhooks/gitlab` e il token segreto

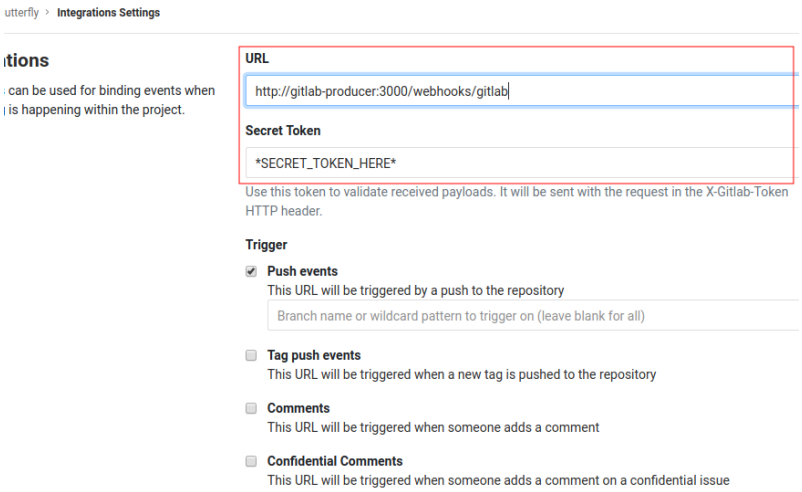

Figura 32: Inserimento URL e secret token

- Selezionare gli eventi che supportati da Butterflye ai quali si è interessati;
- Togliere la spunta a *SSL verification*;

Figura 33: Selezione eventi e deselezione SSL Verification

- Cliccare "Add webhook".

B.2 Webhook Sonarqube

Viene descritto come aggiungere un webhook su Sonarqube:

- Accedere al servizio Sonarqube;
- Accedere ad *Administration*;

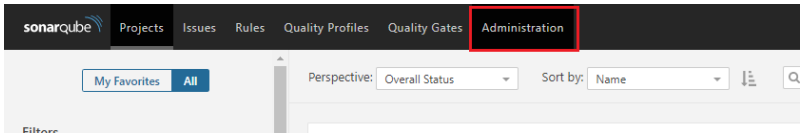

Figura 34: Selezione Administration

- Accedere al menu *Configuration* » *Webhook*;

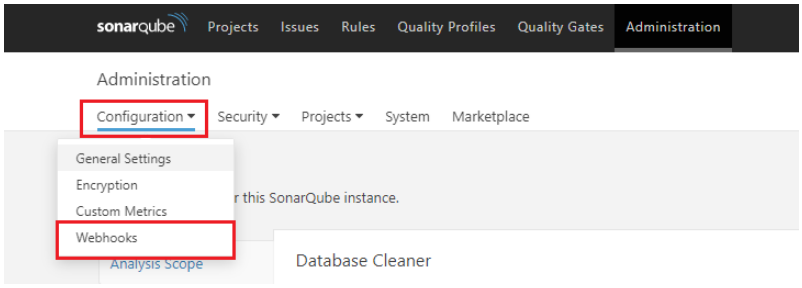

Figura 35: Accesso alle impostazioni del Webhook

- Fare click su *Create*;

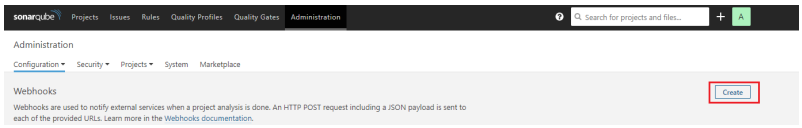

Figura 36: Create Webhook

- Inserire un nome per il webhook nel campo *Name*;
- Inserire l'URL del webhook `http://sonarqube-producer:6000/webhooks/sonarqube`;

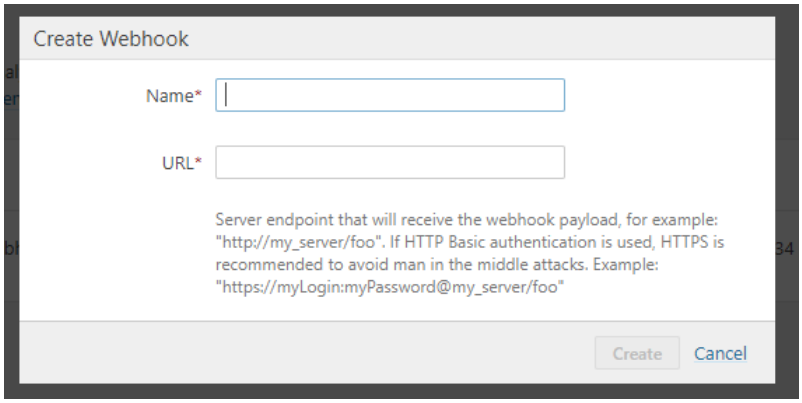

Figura 37: Inserimento Nome e URL

- Cliccare "Create".

C Licenza

Il prodotto Butterfly, sviluppato dal team DStack, è rilasciato al pubblico in maniera completamente *Open Source*. La licenza scelta è la *MIT License*, il cui contenuto è riportato di seguito.

```
MIT License

Copyright (c) 2019 DStack Group

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

D Definizioni utili

Di seguito viene riportata una lista di termini che possono aiutare la comprensione del lettore:

- **Checkstyle**: strumento di analisi statica del codice utilizzato nello sviluppo di software per verificare se il codice sorgente Java è conforme alle regole di codifica. I controlli eseguiti da CheckStyle sono limitati alla presentazione del codice, percui non possono confermare la correttezza o la completezza dello stesso.
- **Confluent Schema Registry**: strumento che fornisce un livello di servizio per i metadati. Si componne di un'interfaccia RESTful per l'archiviazione ed il recupero degli schemi Avro. Permette la memorizzazione di una cronologia delle versioni di tutti gli schemi, fornisce molteplici impostazioni di compatibilità e consente l'evoluzione degli schemi in base alle impostazioni di compatibilità configurate e al supporto Avro espanso. Inoltre fornisce serializzatori che si collegano ai client Kafka, che gestiscono lo storage e il recupero dello schema per i messaggi Kafka inviati nel formato Avro;

- **CRUD**: sono quattro operazioni di base che si possono effettuare su dati persistenti. Esse consistono in: create (inserimento nuovi dati), read (lettura dei dati), update (aggiornamento dei dati), delete (eliminazione dei dati);
- **LTS**: acronimo di **L**ong **T**erm **S**upport, indica una versione di un prodotto con periodo di supporto di 5 anni. Vengono rilasciate ogni 2 anni nell'Aprile degli anni pari. Inoltre sono ideali in tutti quei casi in cui si preveda di effettuare un'installazione duratura nel tempo dove non si ritiene di primaria importanza avere l'intero parco software aggiornato all'ultima versione. Tuttavia la presenza di repository dedicati e l'introduzione dei pacchetti Snap facilita l'installazione di versioni aggiornate dei software più popolari;
- **Message Broker**: nell'ambito di Kafka, i broker (o message broker) sono i contenitori dei topic e sono rappresentati solitamente da dei server. Un gruppo di più broker compone il Kafka Cluster;
- **Nullable**: campo di una classe o di una tabella SQL che può assumere un valore nullo;
- **Publisher-Subscriber**: design pattern che permette di implementare un sistema di messaggistica molti a molti tra un numero arbitrario di oggetti. Il pattern prevede la presenza di mittenti e destinatari di messaggi che dialogano tra loro attraverso un tramite, dispatcher o broker. Il funzionamento del pattern è il seguente:

  1. Il mittente di un messaggio (publisher) si occupa esclusivamente di pubblicare (publish) il proprio messaggio al dispatcher, non essendo consapevole di quale è l'identità dei destinatari (subscriber).
  2. I destinatari (subscriber) si rivolgono, a loro volta, al dispatcher abbonandosi (subscribe) alla ricezione dei messaggi.
  3. A questo punto il dispatcher inoltra ogni messaggio inviato da un publisher a tutti i subscriber interessati a quel messaggio.

  Il meccanismo di sottoscrizione consente ai subscriber di precisare a quali messaggi sono interessati;
- **Topic Kafka**: nell'ambito di Kafka è un canale di messaggi, presente all'interno di un *B*roker, associato ad uno specifico argomento.
- **Transaction log**: log delle transizioni presente in ogni database SQL in cui sono registrate tutte le transazioni e le modifiche apportate dalle transazioni stesse al database. Il log delle transazioni è un componente fondamentale del database. Se si verifica un errore di sistema, è necessario tale registro per ripristinare uno stato coerente del database. Le operazioni concesse riguardano:

  – Recupero di singole transazioni;
  – Recupero di tutte le transazioni incomplete all'avvio del database;
  – Rollforward di una pagina, file, filegroup o database ripristinato fino al punto in cui si è verificato l'errore;
  – Supporto della replica transazionale;
  – Supporto delle soluzioni di ripristino di emergenza e disponibilità elevata: Gruppi di disponibilità Always On, mirroring del database e log shipping.

- **Travis CI**: è un servizio di integrazione continua distribuito utilizzato per costruire e testare progetti software ospitati su GitHub. Un server CI esegue la build di un progetto software, quindi di solito, preleva l'ultima versione del progetto da un sistema di Version Control, come SVN o GIT ed esegue uno script che può spingersi fino al deploy del software in produzione, oppure creare semplicemente dei pacchetti binari pronti per essere installati su una o più macchine di produzione; Nello specifico Travis CI;
- **Webhook**: termine inglese usato per intendere un metodo, nella programmazione informatica, in grado di alterare il comportamento di una pagina web o di una applicazione web. Questa tecnica usa delle chiamate di ritorno (callback) che possono essere gestite, modificate da terze parti non necessariamente affiliate alla pagina web in questione.