AlphaSix

Manuale Sviluppatore

Informazioni sul documento

Nome Documento	ManualeSviluppatore v2.0.0.pdf
Versione	2.0.0
Data di Creazione	23 marzo 2019
Data ultima modifica	13 maggio 2019
Stato	Approvato
Redazione	Timoty Granziero Ciprian Voinea Laura Cameran
Verifica	Samuele Gardin Matteo Marchiori
Approvazione	Timoty Granziero
Uso	Esterno
Distribuzione	AlphaSix
Destinato a	Prof. Tullio Vardanega, Prof. Riccardo Cardin, Imola Informatica
Email di riferimento	alpha.six.unipd@gmail.com

Descrizione

Questo documento è il manuale sviluppatore inerente al prodotto Butterfly del gruppo AlphaSix, utile agli sviluppatori per la manutenzione e l'estensione.

Registro delle modifiche

Versione	Descrizione	Ruolo	Nominativo	Data
2.0.0	Approvazione per il rilascio	Responsabile	Timoty Granziero	2019-05-13
1.1.0	Verifica finale	Verificatore	Matteo Marchiori	2019-05-12
1.0.4	Verifica diagramma dei package e delle classi	Verificatore	Samuele Gardin	2019-05-06
1.0.3	Correzione diagramma delle classi in fig 9 per incorrettezza sintattica	Amministratore	Nicola Carlesso	2019-04-29
1.0.2	Correzione diagramma dei package in fig 2, fig 5, fig 8, fig 13 e fig 15 per associazioni sbagliate	Amministratore	Nicola Carlesso	2019-04-29
1.0.1	Correzione diagramma delle attività in §11	Amministratore	Nicola Carlesso	2019-04-23
1.0.0	Approvazione per il rilascio	Responsabile	Nicola Carlesso	2019-04-11
0.3.0	Verifica finale	Verificatore	Matteo Marchiori	2019-04-09
0.2.5	Aggiornate parole in glossario §A	Programmatore	Timoty Granziero	2019-04-07
0.2.4	Aggiunto paragrafo test §5	Programmatore	Timoty Granziero	2019-04-04
0.2.3	Ampliato §4.4 per descrivere come è possibile estendere MessageProcessor	Programmatore	Laura Cameran	2019-04-04
0.2.2	Aggiunto §4	Programmatore	Timoty Granziero	2019-04-03
0.2.1	Aggiunto §3.2.2 per introdurre la componente connessa a Mongo	Programmatore	Laura Cameran	2019-04-01
0.2.0	Verifica	Verificatore	Samuele Gardin	2019-03-29
0.1.2	Aggiunto paragrafo Architettura di Butterfly in §3	Programmatore	Samuele Gardin	2019-03-28
0.1.1	Aggiunte tecnologie utilizzate in §2	Programmatore	Samuele Gardin	2019-03-27
0.1.0	Verifica	Verificatore	Matteo Marchiori	2019-03-25
0.0.2	Aggiunta introduzione §1	Programmatore	Timoty Granziero	2019-03-24
0.0.1	Creazione template documento	Programmatore	Timoty Granziero	2019-03-23

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Glossario

Al fine di rendere il documento più chiaro possibile, i termini che possono assumere un significato ambiguo o che richiedono una spiegazione avranno una G a pedice (e.g. WEBHOOK$_G$), e saranno riportati nell'appendice §A.

1.2 Scopo del documento

Il documento ha lo scopo di proporsi come guida per lo sviluppatore che in futuro vorranno contribuire a mantenere ed estendere il prodotto.
Verranno analizzate le tecnologie interessate, l'architettura nel dettaglio e le scelte progettuali.

1.3 Scopo del prodotto

Lo scopo del prodotto è creare un applicativo per poter gestire i messaggi o le segnalazioni provenienti da diversi prodotti per la realizzazione di software. Queste segnalazioni passano attraverso un BROKER$_G$ che gestisce i canali a loro dedicate per poi distribuirle ad applicazioni di messaggistica.
Il software dovrà inoltre essere in grado di riconoscere il TOPIC$_G$ dei messaggi in input per poterli inviare a determinati canali a cui i destinatari dovranno iscriversi.
È anche richiesto di creare un canale specifico per gestire le particolari esigenze dell'azienda. Questo dovrà essere in grado, attraverso la lettura di particolari METADATI$_G$, di reindirizzare i messaggi ricevuti al destinatario più appropriato.

2 Tecnologie interessate

In questa sezione descriviamo le tecnologie che dopo una fase di analisi, abbiamo deciso di utilizzare per lo sviluppo di Butterfly.

2.1 Strumenti per lo sviluppo

2.1.1 Python

I linguaggi consigliati per l'implementazione di Butterfly sono: Python, Java e NodeJS. Dopo un'analisi che abbiamo svolto sui vari linguaggi proposti, siamo arrivati a scegliere Python perché è un linguaggio di programmazione dinamico, orientato agli oggetti, distribuito con licenza Open-Source e utilizzabile per molti tipi di sviluppo software. Offre inoltre un forte supporto all'integrazione con altri linguaggi e programmi, è fornito di una estesa libreria standard e può essere imparato in poco tempo.

2.1.2 MongoDB

Abbiamo deciso di utilizzare MongoDB$_G$ piuttosto di un database relazionale quale MySQL$_G$, perché non sfrutta una struttura tradizionale basata su tabelle ma si basa sui documenti in stile BSON$_G$ con schema dinamico, rendendo l'integrazione di dati di alcuni tipi di applicazioni più facile e performante.

2.2 Strumenti per la gestione dei container

2.2.1 Docker

Abbiamo deciso di utilizzare Docker$_G$ per la semplicità di utilizzo e per mantenere le componenti del tutto isolate. La sua configurazione avverrà tramite un Dockerfile$_G$ in cui verranno specificate informazioni come sistema operativo, script di avvio, numero di istanze ed altri parametri specifici.
Docker oltretutto ci consente di creare dei container$_G$ aventi la capacità di simulare un ambiente virtuale dov'è possibile testare e mantenere le proprie applicazioni, permettendo di aumentare l'efficienza riducendone i costi e simulando l'esecuzione di sistema operativo su una macchina con risorse$_G$ condivise.
A differenza delle macchine virtuali$_G$, dove lo stato dell'ambiente viene salvato su disco, occupando memoria, i container si adattano in maniera più performante all'applicativo richiesto, in quanto il loro scopo è quello di massimizzare la quantità delle applicazioni in esecuzione riducendo al minimo il numero delle macchine per eseguirla. Sono quindi più leggeri, occupando meno memoria su disco e impiegando meno risorse.

2.2.2 Rancher

Per la gestione dei container in remoto, la proponte Imola Informatica ci ha messo a disposizione un cluster$_G$ con due server sui quali è stato installato Rancher. Questo è un software di gestione di oggetti di Kubernetes$_G$. Da qui possiamo quindi gestire i nostri container installando le immagini direttamente da DockerHub$_G$ senza aver bisogno di file di configurazione come ad esempio quello necessario al docker-compose$_G$.

2.3 Librerie esterne

Le librerie esterne usate per Python3 scelte sono tutte installabili con il gestore dei pacchetti di Python pip$_G$.
Queste saranno già installate nel cluster fornito dalla proponente.

2.3.1 Requests

Libreria per la gestione delle richieste HTTP. Grazie a essa è possibile effettuare ogni tipo di richiesta (GET, POST, DELETE, PUT). Viene utilizzata principalmente per effettuare richieste tramite la API di Telegram.
Maggiori informazioni al link

http://docs.python-requests.org/en/master/

Installabile con il comando `pip install requests`.

2.3.2 Flask

Flask è un framework lightweight per l'installazione di web server, scritto in Python. Viene utilizzato dai PRODUCER$_G$ per restare in ascolto degli WEBHOOK$_G$ e dal Gestore Personale per istanziare l'interfaccia web per la gestione del personale.
Maggiori informazioni al link:

http://flask.pocoo.org/docs/1.0/

Installabile con il comando `pip install flask`.

2.3.3 Flask-restful

Libreria che estende Flask e semplifica la gestione di API Rest, automatizzando il collegamento tra risorse e le relative tipologie di richiesta. Viene usata nel Gestore Personale per implementare lo standard API Rest.
Maggiori informazioni al link:

https://flask-restful.readthedocs.io/en/latest/

Installabile con il comando `pip install flask-restful`.

2.3.4 PyMongo

Libreria per l'utilizzo di MongoDB con Python. Il database viene utilizzato nel Gestore Personale.
Maggiori informazioni al link:

https://api.mongodb.com/python/current/

Installabile con il comando `pip install pymongo`.

2.3.5 Pytest

Pytest è uno dei framework più utilizzati in Python per l'esecuzione dei test. Semplifica di molto la loro gestione e implementazione rispetto al modulo `unittest` della libreria standard.
Maggiori informazioni al link:

https://docs.pytest.org/en/latest/

Installabile con il comando `pip install pytest`.

3 Architettura

In questa sezione descriviamo brevemente l'architettura del sistema Butterfly.
L'architettura adottata è EVENT-DRIVEN$_G$: ogni componente è isolato e asincrono, e comunica attraverso interfacce scambiandosi messaggi tramite il broker Apache Kafka.
Il sistema si divide principalmente in tre insiemi di componenti:

- Producers
- Gestore Personale
- Consumers

3.1 Producers

Il Producer è il componente che resta in ascolto degli webhook provenienti dal suo applicativo specifico (e.g. Redmine, GitLab). Ha lo scopo di immettere i messaggi su Kafka in formato JSON$_G$, conservando solo i campi di interesse e aggiungendone eventualmente di propri.
Al momento della stesura di questo manuale, i Producer implementati sono due:

- RedmineProducer
- GitlabProducer

L'algoritmo di invio dei messaggi differirà solamente nella modalità in cui verrà formato il messaggio finale: per questo motivo abbiamo deciso di utilizzare il design pattern TEMPLATE METHOD$_G$ per l'implementazione dell'algoritmo. Le classi concrete avranno esclusivamente il compito di implementare il parser degli webhook e il metodo del Producer `webhook_kind()`.

3.1.1 RedmineProducer

Il Producer di Redmine si occupa di ascoltare gli webhook provenienti dai progetti di Redmine che hanno configurato la porta relativa al componente, e di immettere nella coda *redmine* di Kafka i messaggi.
Nel seguente diagramma di sequenza è possibile vedere il flusso di esecuzione, che parte dall'arrivo di un webhook da Redmine e si conclude con l'inoltro del messaggio sulla coda *redmine* di Kafka.

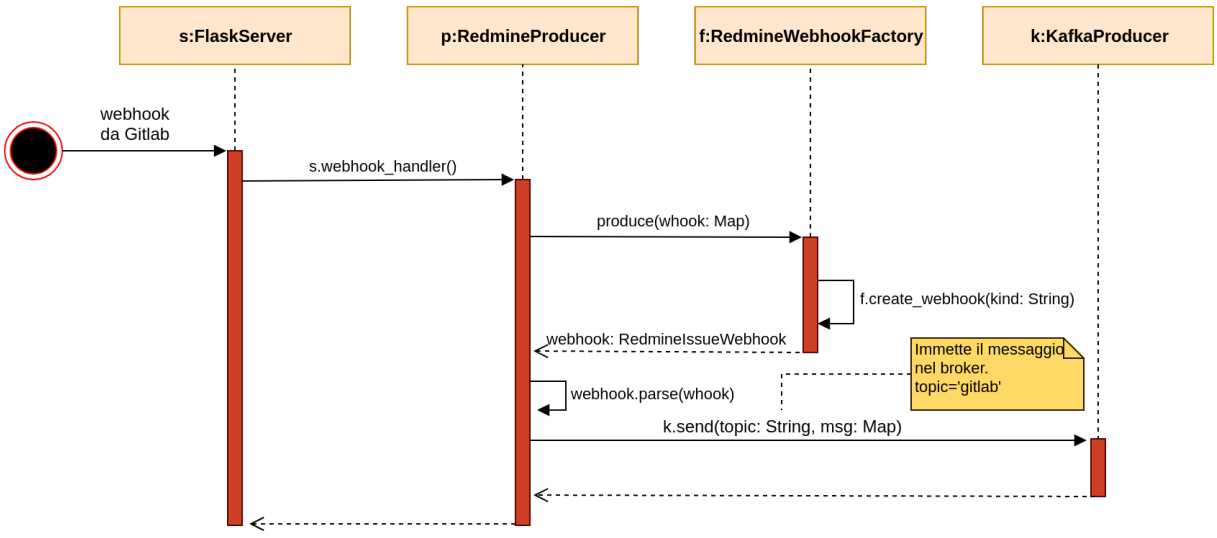

Figura 1: Diagramma di sequenza per la ricezione di un messaggio su RedmineProducer e invio del messaggio su Kafka

3.1.1.1 Diagramma dei package

Il package `producer` ha tre dipendenze esterne:

- `Flask`, classe concreta del package di libreria `flask`: è il server che si mette in ascolto degli webhook provenienti da Redmine.
- `KafkaProducer`, classe concreta del package di libreria `kafka-python`: ha il compito di interfacciarsi con il broker Kafka e di immettere in esso i messaggi. È costruito su di esso un ADAPTER$_G$, in cui KafkaProducer viene adattato all'astrazione `Producer`.
- Package `webhook`: ha il compito di effettuare il parsing dei messaggi in entrata.

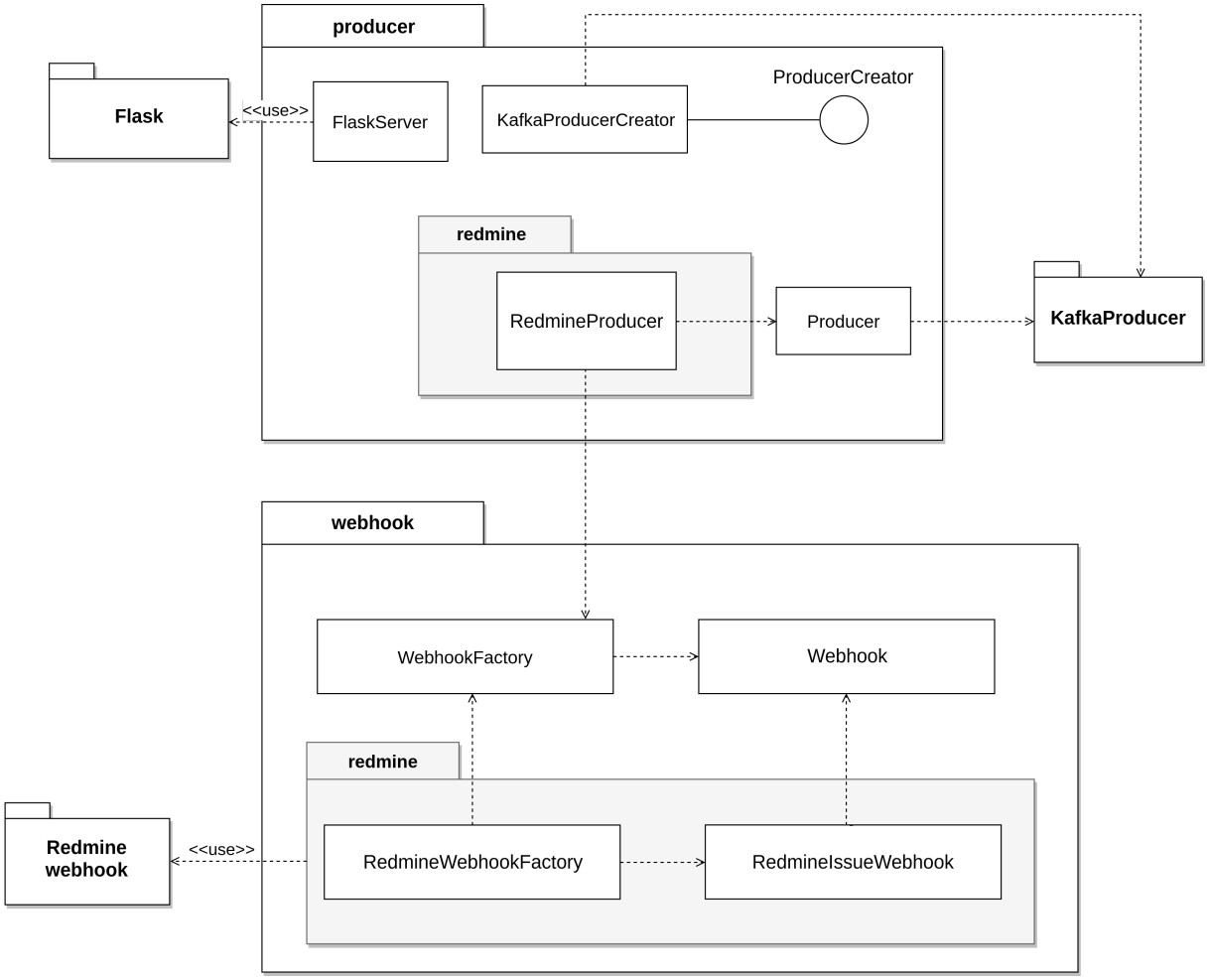

Figura 2: Diagramma dei package di RedmineProducer

3.1.1.2 Diagramma delle classi

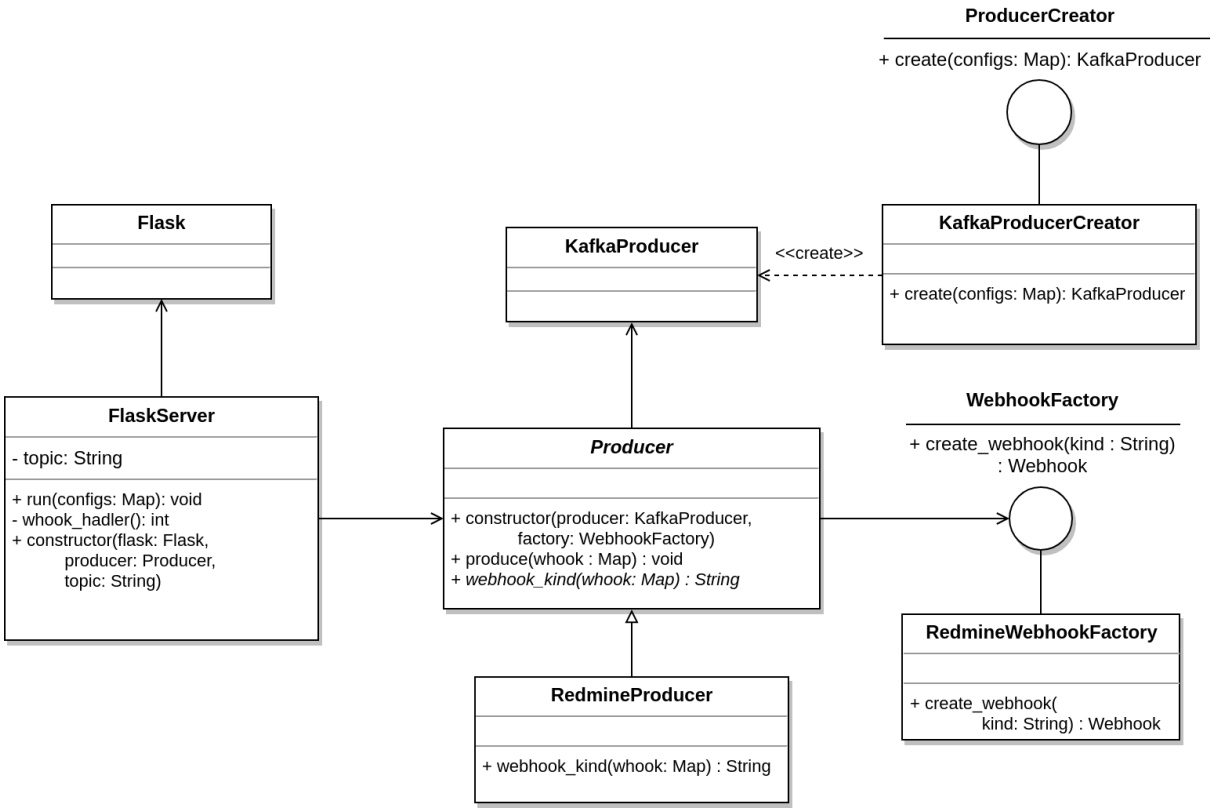

Figura 3: Diagramma delle classi di RedmineProducer

Per fare il parsing dei webhook, `RedmineProducer` si appoggia alla gerarchia `Webhook`. Ogni istanza di `RedmineProducer` ha un riferimento a una `WebhookFactory`: quando viene catturato un webhook da Redmine, chiama il metodo `create_webhook(kind)` della factory associata, passandogli come parametro la tipologia di webhook, in formato stringa.
Essa si occuperà di creare il parser di webhook adeguato al messaggio in entrata. La classe concreta derivata da `Webhook` costruisce il messaggio con i campi di interesse da immettere nella coda di Kafka, e lo restituisce al Producer.
Per Redmine, è presente una sola tipologia di webhook. Tenendo a mente l'OPEN-CLOSED PRINCIPLE$_G$, è stata implementata la gerarchia relativa ai webhook per permettere agli sviluppatori di poterne facilmente aggiungere di nuovi.

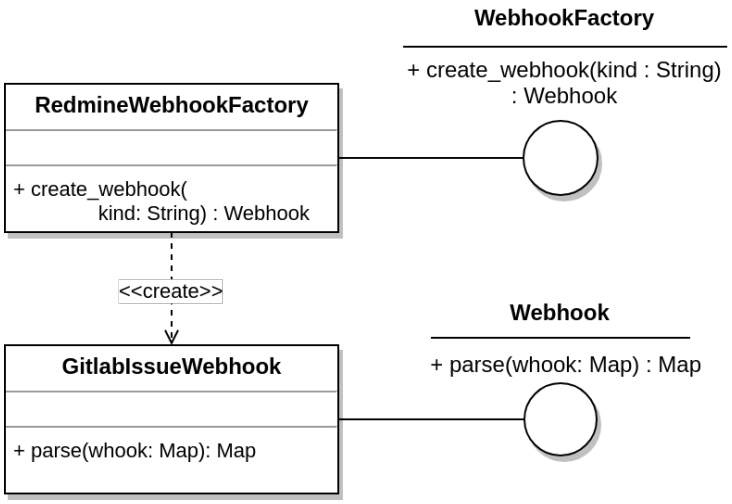

Figura 4: Diagramma delle classi di RedmineWebhookFactory

3.1.2 GitlabProducer

Il Producer di GitLab si occupa di ascoltare gli webhook provenienti dai progetti di GitLab che hanno configurato la porta relativa al componente, e di immettere nella coda gitlab di Kafka i messaggi.
Esso è implementato allo stesso modo di quello di Redmine, per cui non verrà ripetuta l'analisi delle funzionalità che sono identiche al Producer di Redmine.
Per tenere isolate le componenti, i diagrammi sono stati totalmente separati, anche se le classi risiedono nello stesso package. Sarà compito dei Dockerfile copiare solamente i file necessari al componente specifico per poter mantenere isolato il container.

3.1.2.1 Diagramma dei package

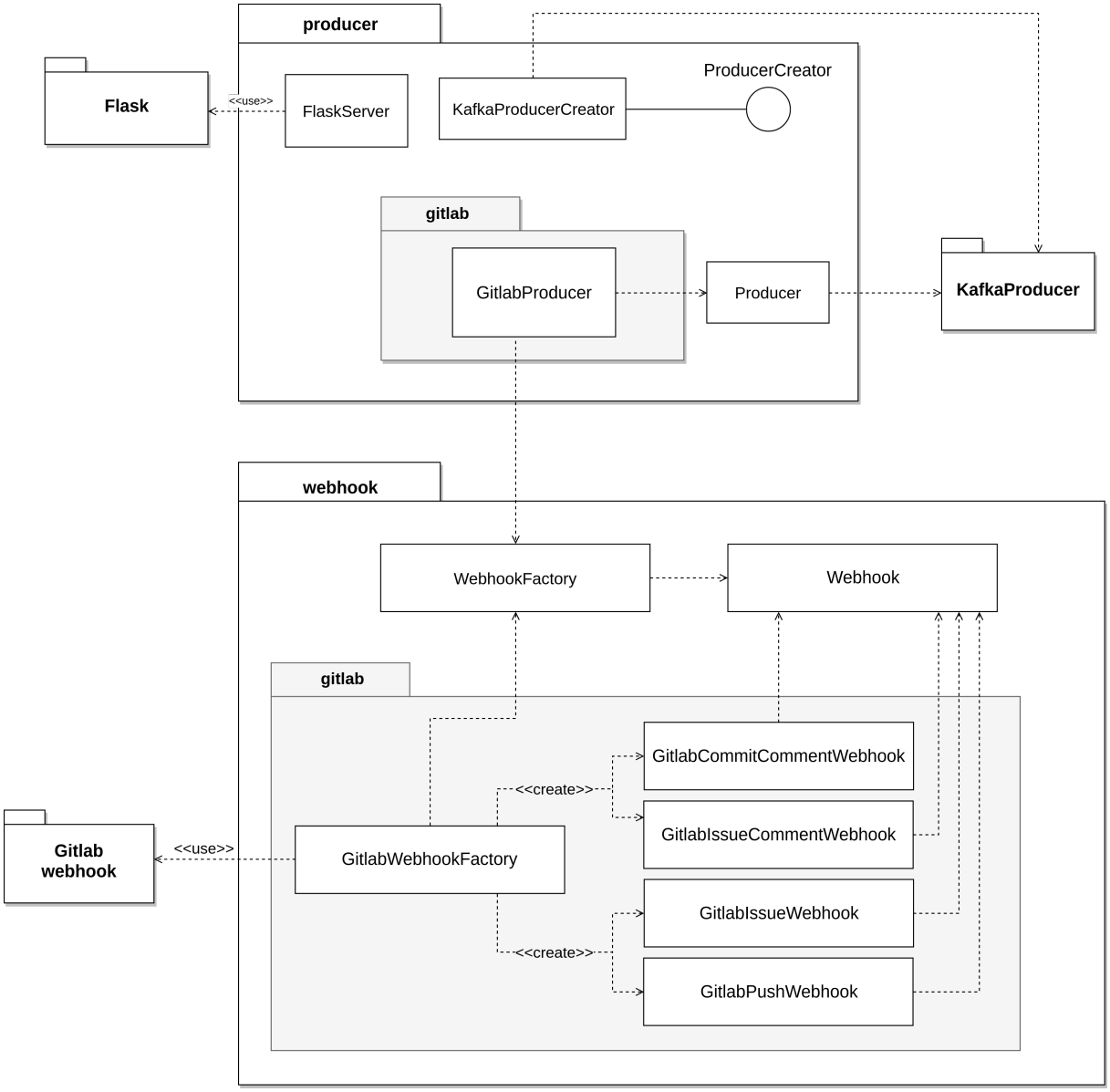

Figura 5: Diagramma dei package di GitlabProducer

3.1.2.2 Diagramma delle classi

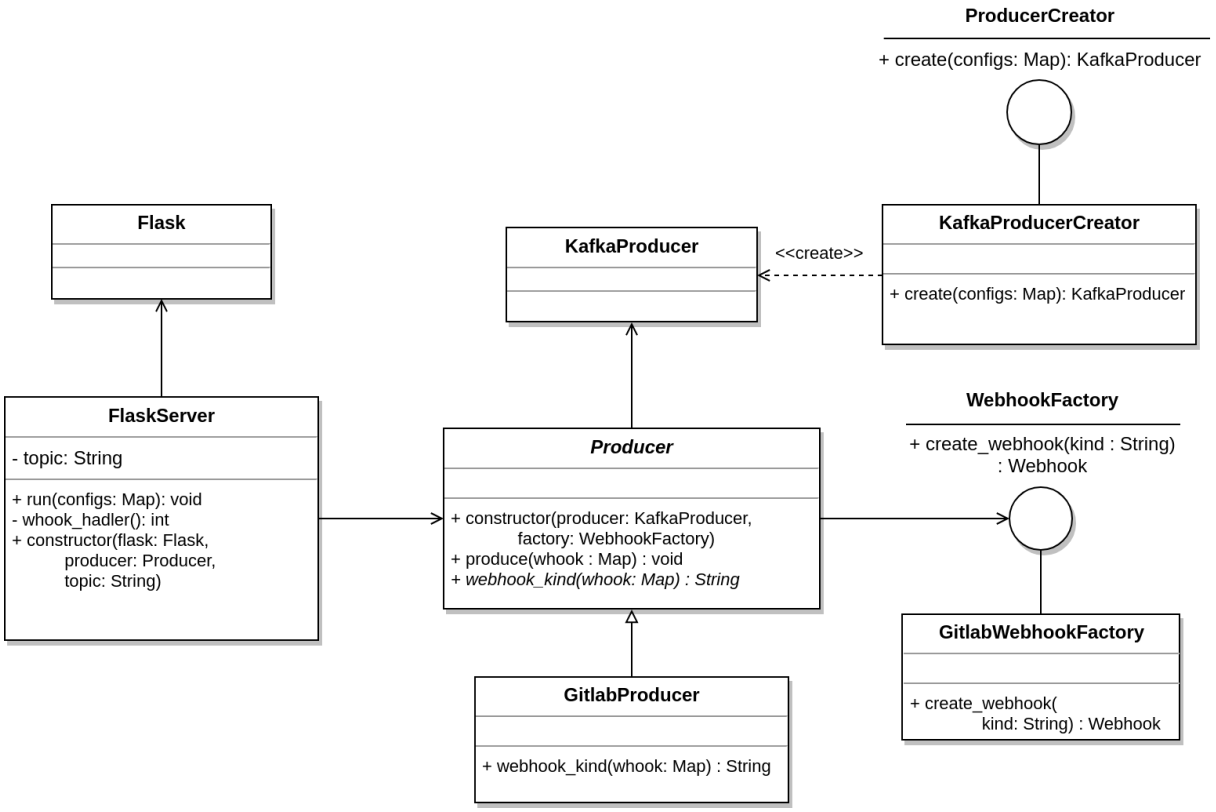

Figura 6: Diagramma delle classi di GitlabProducer

Abbiamo implementato per `GitlabProducer` 4 tipologie di `Webhook`. Il parametro `kind` del metodo `create_webhook(kind)` può quindi assumere 4 valori, in base al quale `GitlabWebhookFactory` costruirà un parser di `Webhook` apposito:

- '`issue`': verrà istanziato un `GitlabIssueWebhook`.
- '`push`': verrà istanziato un `GitlabPushWebhook`.
- '`issue-comment`': verrà istanziato un `GitlabIssueCommentWebhook`.
- '`commit-comment`': verrà istanziato un `GitlabCommitCommentWebhook`.

Questo è un pattern creazionale specifico: Factory Method$_G$.

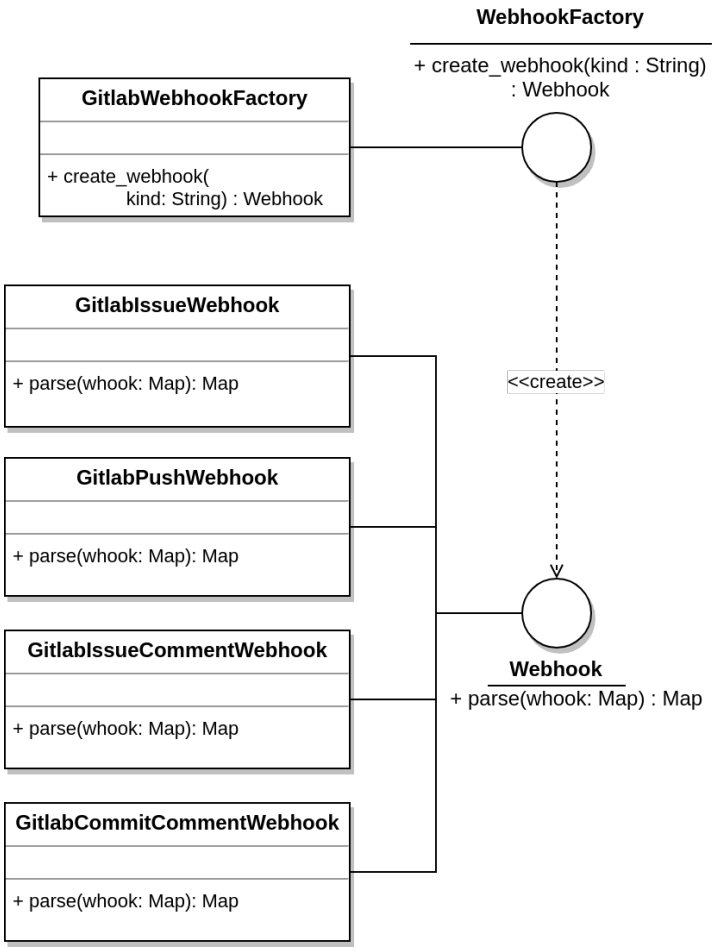

Figura 7: Diagramma delle classi di GitlabWebhookFactory

3.2 Gestore Personale

3.2.1 Diagramma dei package

Il Gestore Personale è la componente con la logica più complessa del sistema Butterfly. Esso ha un proprio KafkaProducer e KafkaConsumer che si occupano rispettivamente di ricevere i messaggi da Kafka e reinserirli nelle code relative ai Consumer finali. Per stabilire il destinatario e il Consumer finale appropriato, il Client del Gestore Personale interroga MongoDB per ottenere le informazioni relative ad i giorni di indisponibilità, la priorità dei progetti e la piattaforma di messaggistica sul quale ricevere la notifica per ogni utente. Tutte queste informazioni vengono inserite in MongoDB attraverso la View.
Questa prima parte riguarda la sezione del gestore personale che va a interfacciarsi con l'utente adottando un'architettura MVC pull model, ovvero `rest_hadler` nella figura 8. L'unica cosa che fa le View è inoltrare i comandi ricevuti dall'utente all'API REST_{G}.
Esiste poi invece anche un'altra parte chiamata `message_processor` che è collegata al MongoDB. Questo perchè deve poter analizzare i dati dei vari utenti in modo da capire dove inserire su Kafka il mesaggio elaborato.

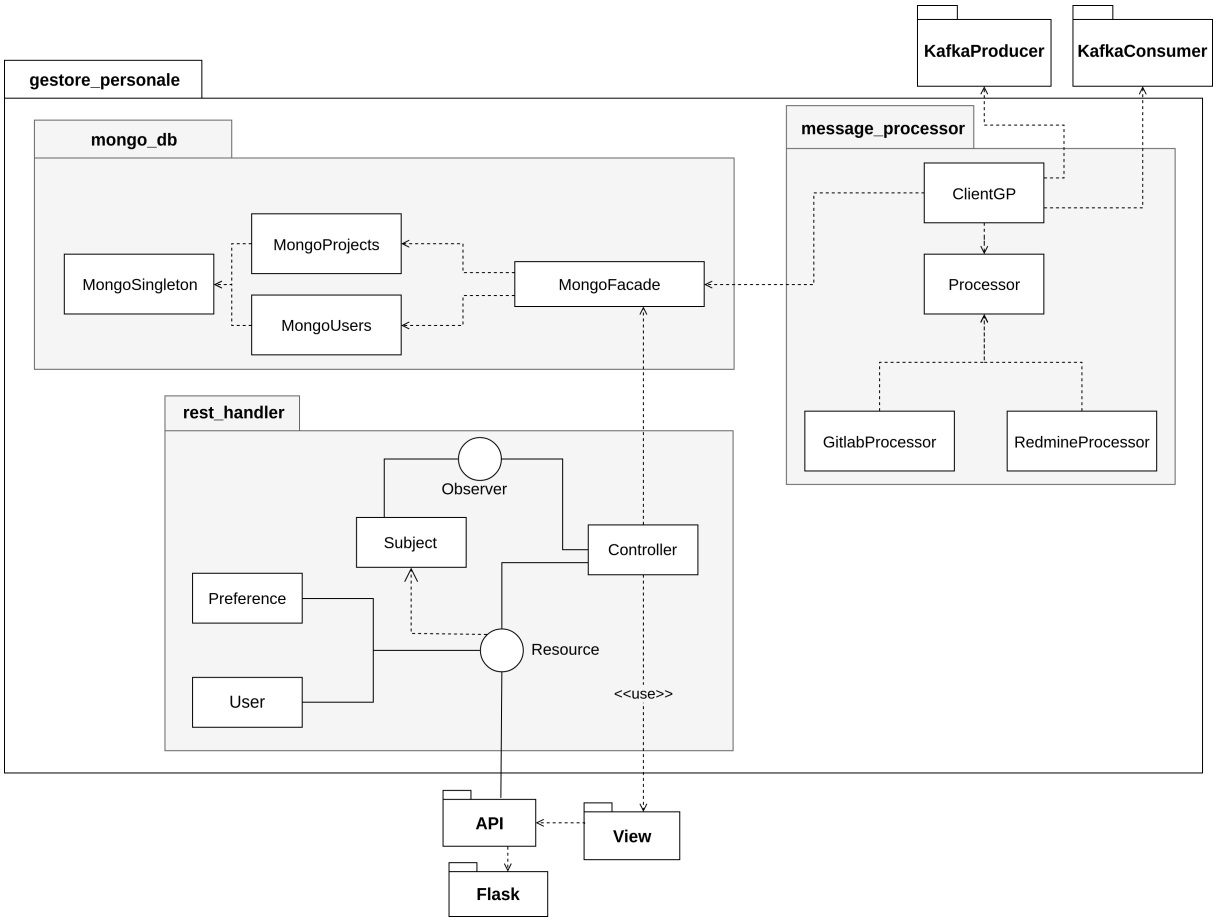

Figura 8: Diagramma dei package del Gestore Personale

3.2.2 Mongo

MongoDB$_G$ è un database non relazionale orientato all'utilizzo di documenti, e non di tabelle come nei database relazionali. Ci è molto utile in questo progetto perché ci dà la possibilità di aggiungere dati ai documenti senza modificarne la struttura. Questo lo rende un database molto estensibile, quindi uno sviluppatore può facilmente ampliare le informazioni in esso contenute.

3.2.2.1 Diagramma delle classi

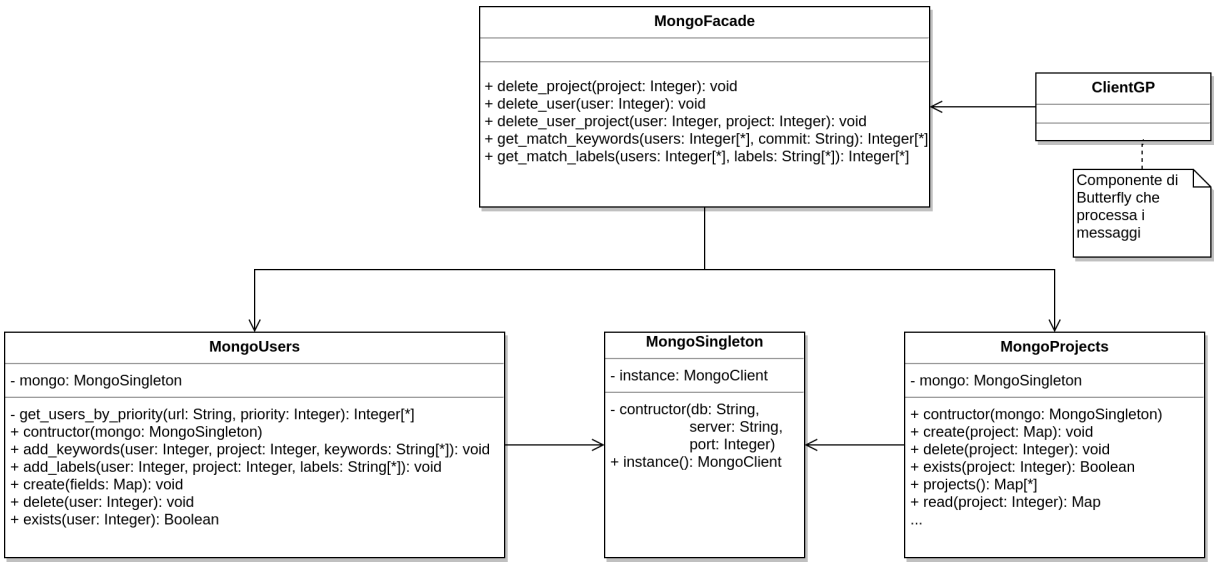

Figura 9: Interazione con MongoDB del Gestore Personale

Per quanto riguarda il database, abbiamo una classe singleton che permette al nostro sistema di avere un'unica istanza della connessione con PyMongo. I metodi che agiscono sulla base di dati sono implementati nelle classi `MongoUser` e `MongoProject`, e sono suddivisi in base alle risorse di interesse. Tali classi vengono raccolte in un facade che espone solamente i metodi che servono verso l'esterno, per renderli più facilmente fruibili.
Nella figura 9 vengono mostrate queste classi e le loro dipendenze.

3.2.3 Controller e View

Il controller recupera gli input che gli utenti inseriscono attraverso la View, le interpreta, per poi poterle elaborare correttamente, interagendo eventualmente con il database. Nel momento in cui cambiano i dati relativi agli utenti nel database il controller aggiorna la View.

3.2.3.1 Diagramma delle classi

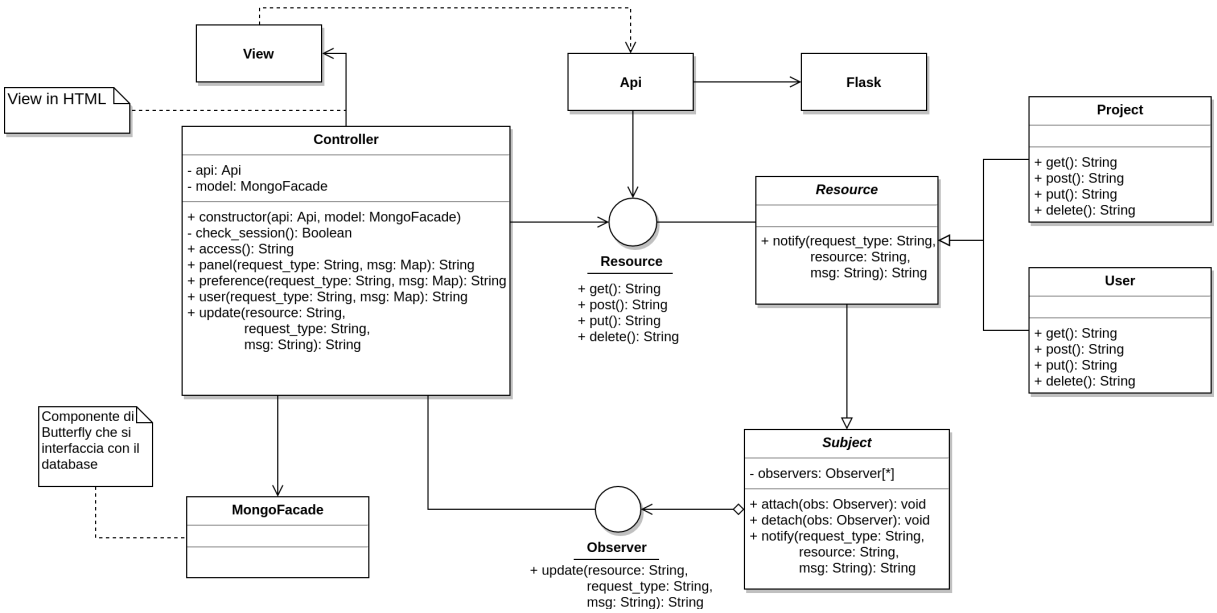

Figura 10: Logica del Controller del Gestore Personale

Il Controller del Gestore Personale serve per collegare la componente del Model con la componente View. Una volta istanziati il server Flask, l'API REST e MongoDB vengono passati al Controller che resta in ascolto.
Quando arriva una richiesta dall'API REST o dalla View, il Controller esegue le dovute operazioni su MongoDB e restituisce una risposta alla vista di provenienza, in formato JSON se si tratta dell'API REST o in formato HTML se si tratta della view consultabile da browser Web.

3.2.4 Client

Il suo compito è quello di recuperare i messaggi inseriti in Kafka dai Producer GitLab e Redmine attraverso un Consumer fittizio, analizzare il messaggio, associargli il destinatario corretto e inserirlo nella coda di Kafka appropriata (di Telegram o Email) attraverso un Producer fittizio.

Per analizzare il messaggio il Client opera in questo modo:

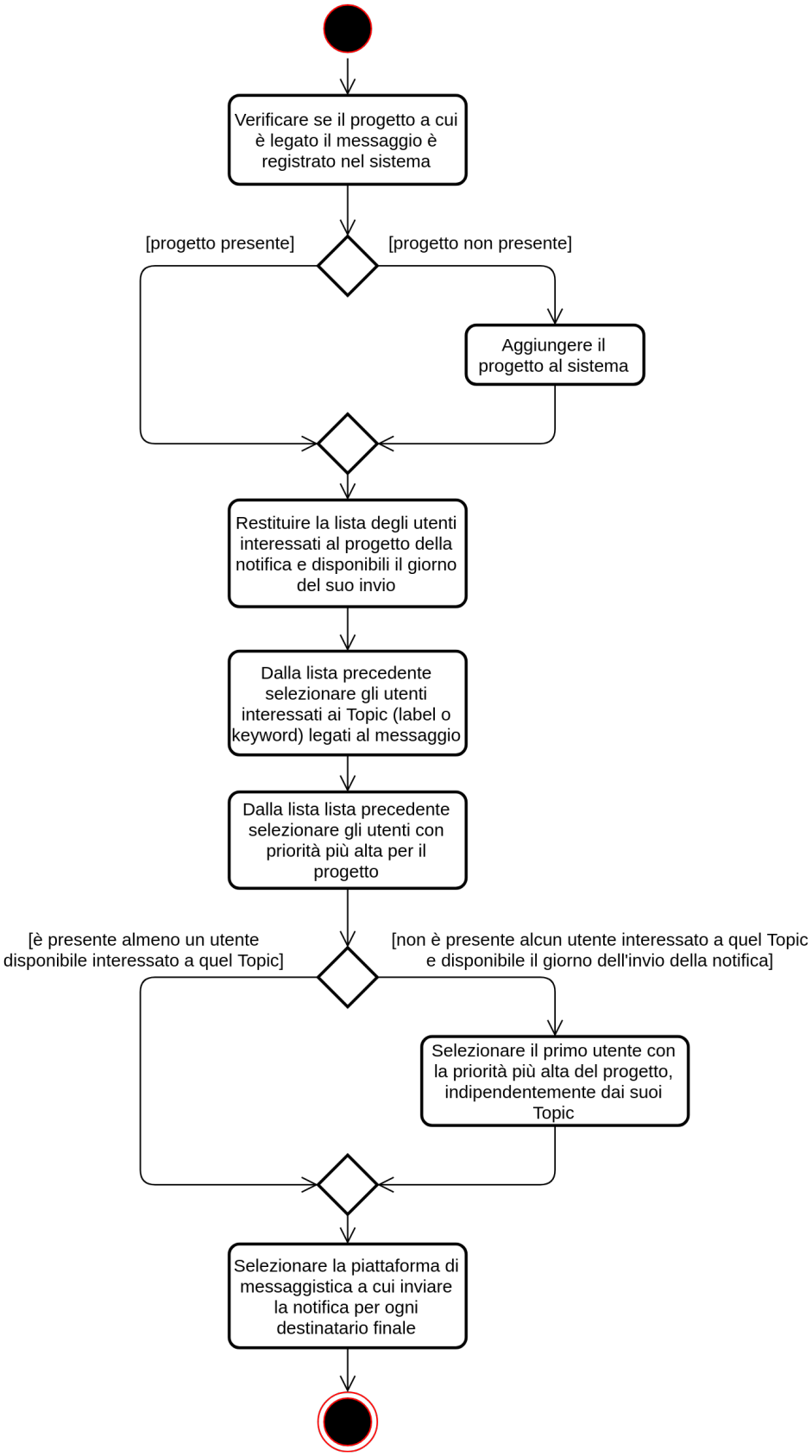

Figura 11: Diagramma di attività del Client

3.2.4.1 Diagramma delle classi

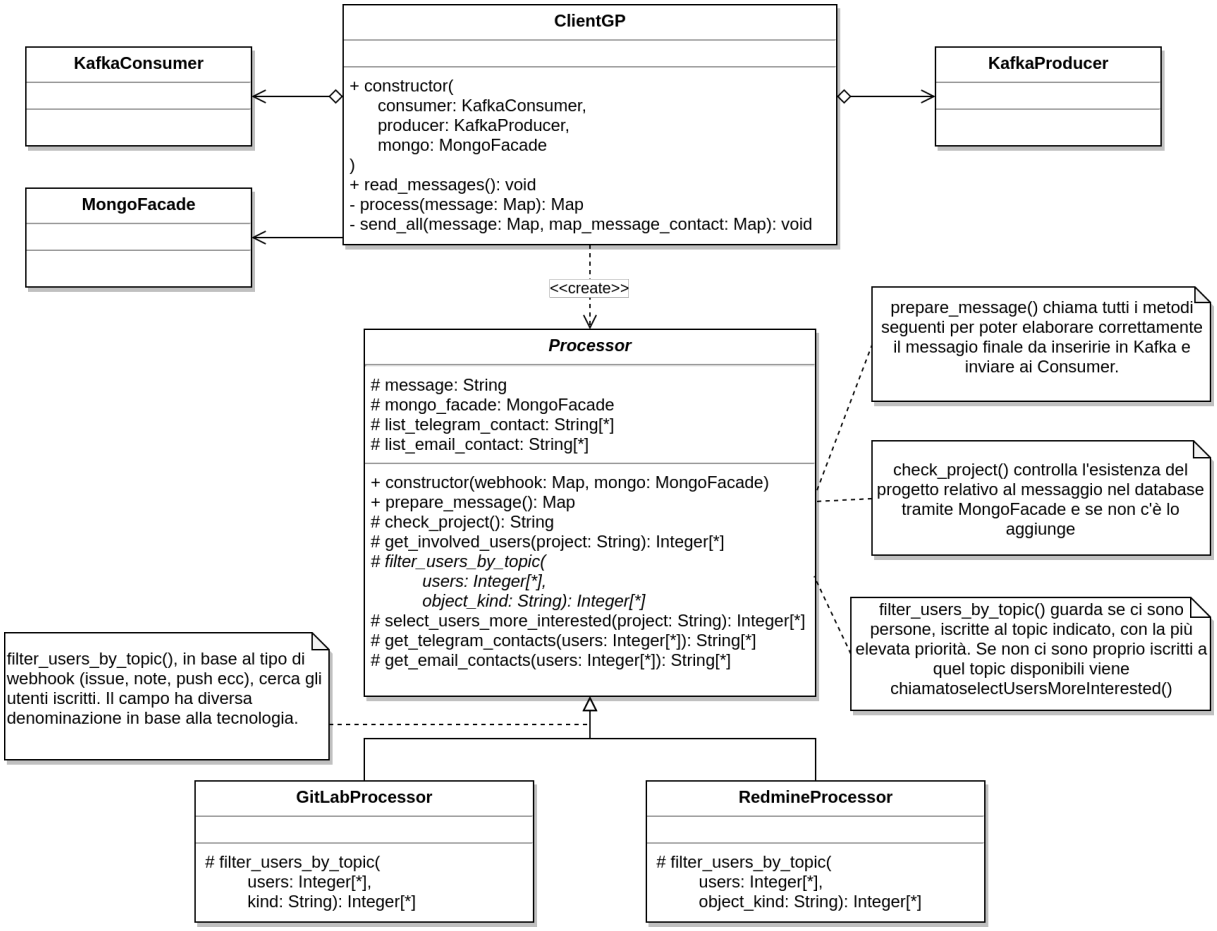

Figura 12: Message processor del Gestore Personale

Il Gestore Personale usa ora `ClientGP` che si interfaccia con Kafka attraverso `KafkaProducer` e `KafkaConsumer`, e per elaborare i messaggi che riceve crea un'implementazione di Processor. Questa, in base alle informazioni nel messaggio ricevuto, indica qual è il destinatario e attraverso quale applicazione notificarlo interrogando il database con `MongoFacade`.

3.3 Consumers

Il Consumer è il componente finale del sistema Butterfly. Esso resta in ascolto della sua coda specifica di Kafka per applicativo (e.g. telegram, email). Si occupa di inoltrare il messaggio al destinatario finale.
Al momento della stesura di questo manuale, i Consumer implementati sono due:

- TelegramConsumer
- EmailConsumer

L'algoritmo di ascolto dei messaggi è identico per ogni Consumer: come per i Producer, abbiamo utilizzato Template Method per l'implementazione di `listen()`. Le classi concrete avranno esclusivamente il compito di implementare il metodo `send()`, il quale invierà il messaggio al destinatario finale tramite le API dell'applicativo su cui il Consumer è basato.

3.3.1 TelegramConsumer

Nel seguente diagramma di sequenza è possibile vedere il flusso di esecuzione che parte dalla ricezione del messaggio dalla coda *telegram* di Kafka all'invio del messaggio all'utente finale su Telegram.

3.3.1.1 Diagramma dei package

Il TelegramConsumer ha due dipendenze esterne:

- `KafkaConsumer` dalla libreria esterna `kafka-python`: offre le funzionalità di ascolto di una o più code specifiche. È costruito su di esso un adapter, per adattare `KafkaConsumer` alle funzionalità di `Consumer`.
- Package `requests`: effettua le richieste POST e poter inviare i messaggi tramite l'API di Telegram al destinatario finale.

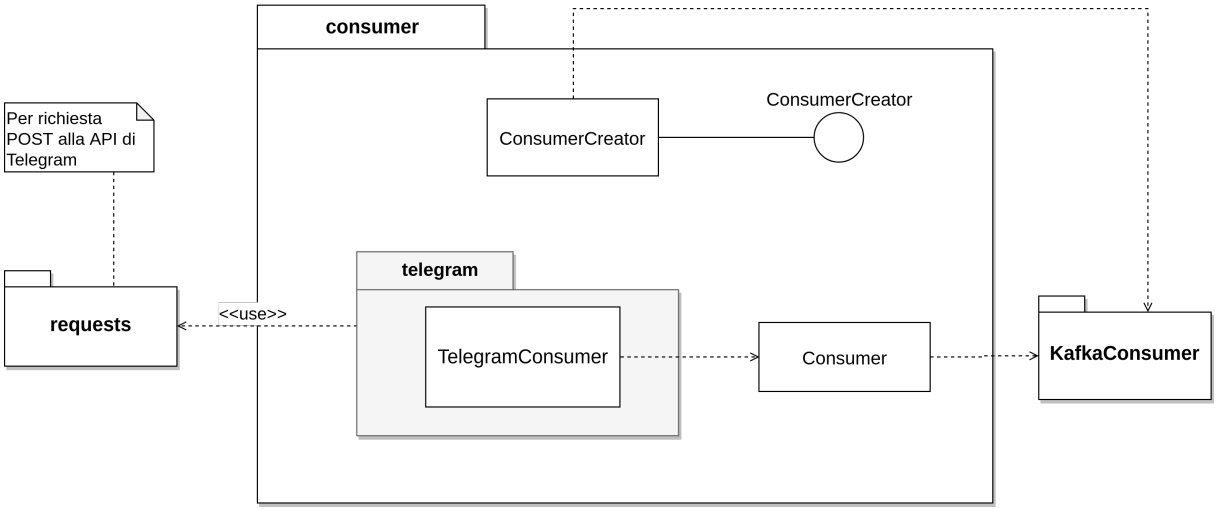

Figura 13: Diagramma dei package di TelegramConsumer

3.3.1.2 Diagramma delle classi

Il metodo `format()` costruisce il testo del messaggio finale utilizzando il formato MARKDOWN$_G$ specifico per Telegram.

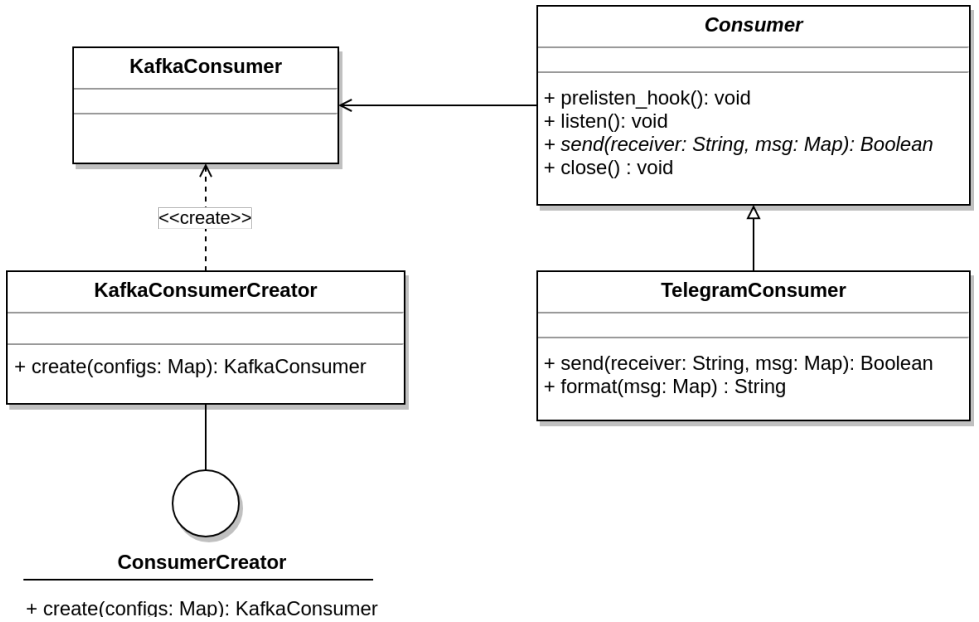

Figura 14: Diagramma delle classi di TelegramConsumer

3.3.2 EmailConsumer

3.3.2.1 Diagramma dei package

L'EmailConsumer ha due dipendenze esterne:

- `KafkaConsumer` della libreria esterna `kafka-python`: offre le funzionalità di ascolto dei messaggi proveniente da una o più code specifiche. È costruito su di esso un adapter, per adattare le funzionalità di `KafkaConsumer` a `Consumer`.
- `SMTP` della libreria esterna `smtplib`: server e-mail che, dopo aver effettuato il login, invia al destinatario una mail contenente il messaggio finale.

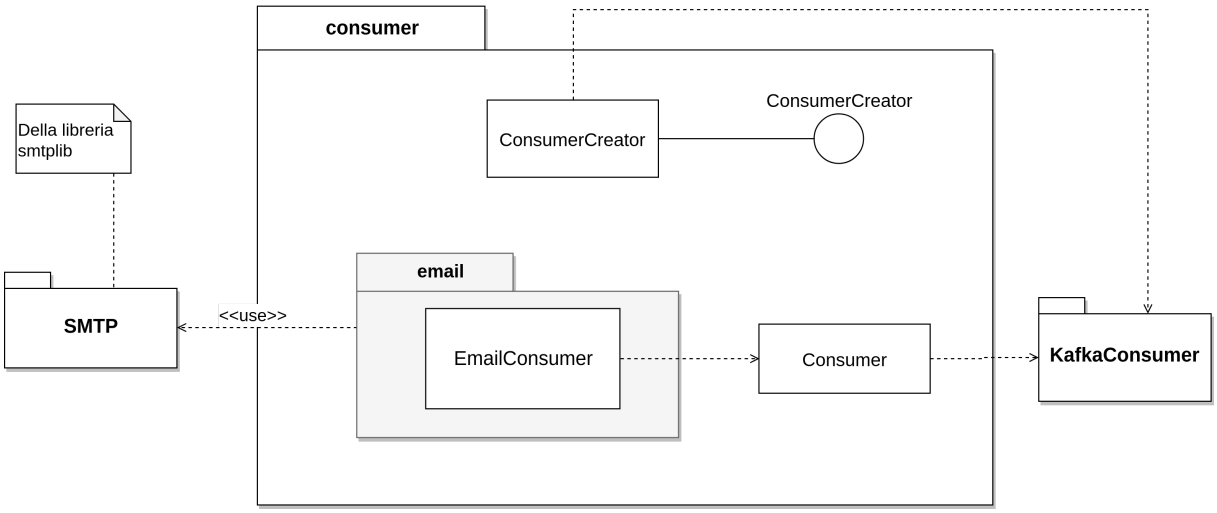

Figura 15: Diagramma dei package di EmailConsumer

3.3.2.2 Diagramma delle classi

- Il metodo `format_html()` costruisce il testo del messaggio finale in formato HTML

- Il metodo `format()` costruisce il testo per il messaggio senza i tag HTML, in caso il client mail del ricevente non sia in grado di interpretare tale linguaggio

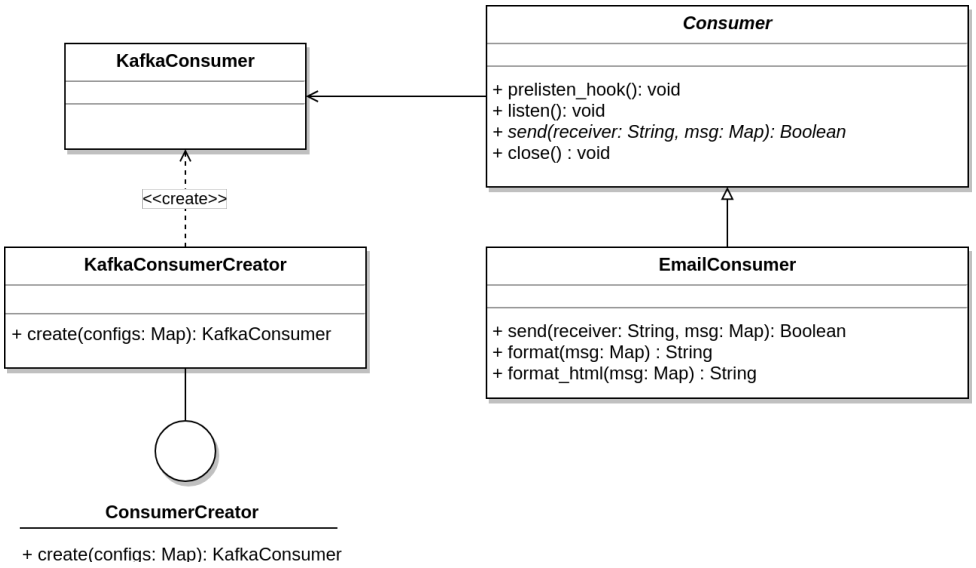

Figura 16: Diagramma delle classi di EmailConsumer

3.4 Impostazioni dei vari componenti

Per fare in modo che tutti i componenti vengano istanziati con i determinati parametri che permettano la giusta comunicazione tra le componenti, ci sono dei file di configurazioni che contengono per esempio il token del bot Telegram, l'email e la password per accedere al server Email. Queste configurazioni sono contenute nelle variabili d'ambiente descritte nel *ManualeSviluppatore v2.0.0*$_{\text{D}}$, oppure nei file "`config.json`" contenuti nelle cartelle dei vari componenti (e.g. producer, consumer ecc.).

3.5 Interazione tra i componenti

Vedendo Butterfly ad alto livello, si possono identificare sei componenti. Redmine e GitLab, per comunicare con i relativi Producer, utilizzano dei webhook in formato JSON, mettendosi in ascolto su determinate porte utilizzando il webserver Flask. Per quanto riguarda la comunicazione tra i Producer e Kafka utilizziamo delle istanze di KafkaProducer. Per far comunicare Kafka con i vari Consumer utilizziamo istanze di KafkaConsumer. Nell'ultimo passo, quello che riguarda i Consumer e le applicazioni finali, quali Telegram e Email, usiamo le API messe a disposizione dagli applicativi.

4 Estendere Butterfly

4.1 Aggiungere un Webhook

Per aggiungere una nuova tipologia di Webhook parser, è sufficiente estendere l'interfaccia `Webhook` e un eventuale `WebhookFactory` in caso si tratti di una nuova applicazione.

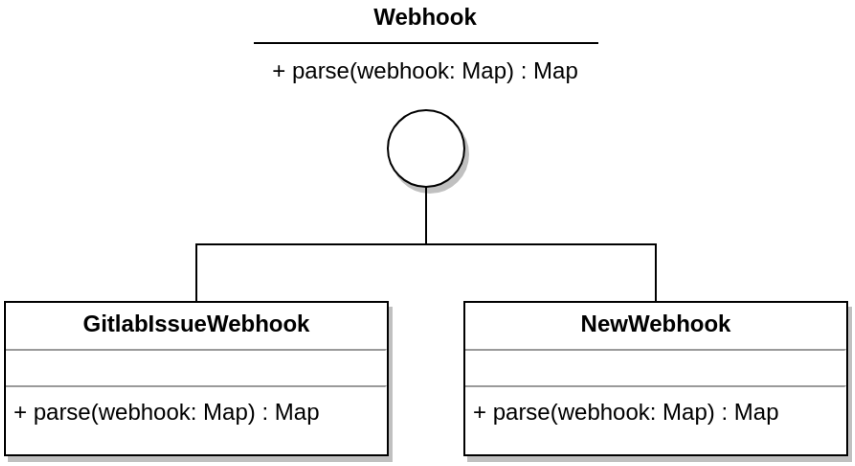

Figura 17: Aggiungere un Webhook parser

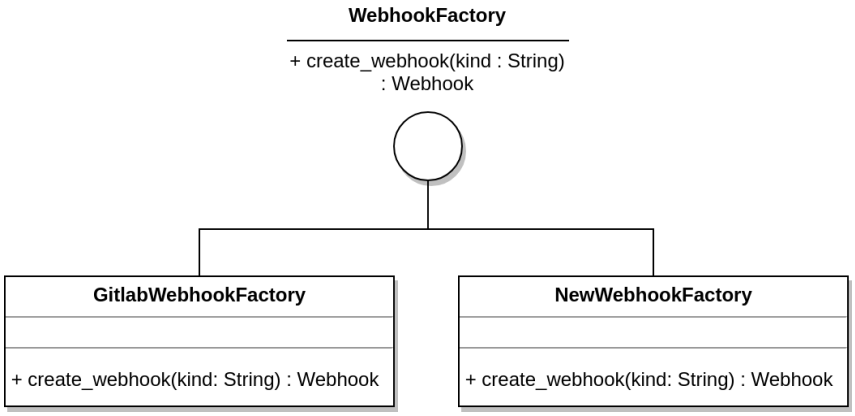

Figura 18: Aggiungere una Factory

Se è un nuovo webhook di un'applicativo esistente, è necessario che la factory conosca la tipologia di webhook. Aggiungere tale informazione al metodo `create_webhook(kind)`.

4.2 Aggiungere un Producer

Per aggiungere un Producer di una nuova applicazione, creare una nuova classe che estenda la classe astratta `Producer`. Definire il metodo astratto `webhook_kind(webhook)`.

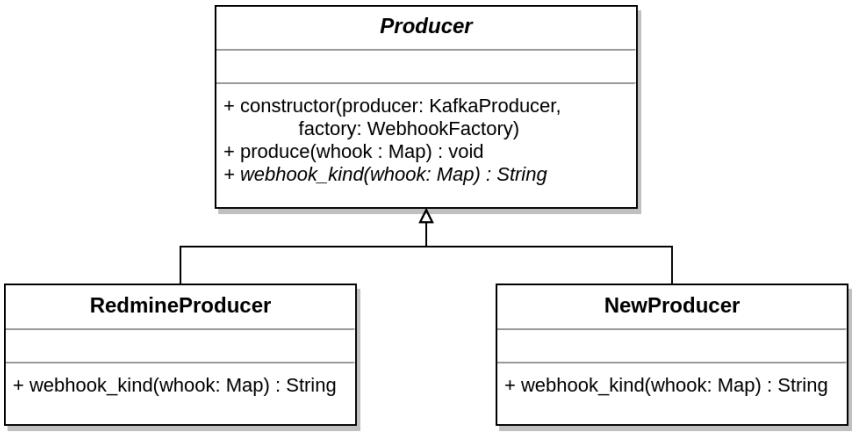

Figura 19: Aggiungere un Producer

4.3 Aggiungere un Consumer

Per aggiungere un Consumer di una nuova applicazione, creare una classe che estenda la classe astratta `Consumer`. Definire il metodo astratto `send(receiver: String, msg: Map)` che, dato in input un destinatario e un insieme di coppie chiave-valore, produca un messaggio formattato e lo invii al destinatario tramite le API dell'applicazione sulla quale inviare il messaggio. È consigliato creare un metodo a parte per la formazione del messaggio.

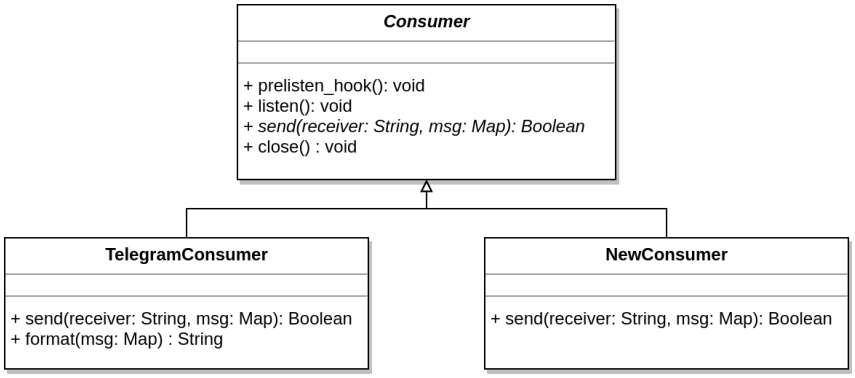

Figura 20: Aggiungere un Consumer

Viene inoltre offerta la possibilità di ridefinire il metodo `prelisten_hook()` che viene chiamato prima che il Consumer si metta effettivamente in ascolto del topic specifico su Kafka. È concreto nella class base, ma non fa nulla. Serve per aumentare la flessibilità delle classi che implementano la classe astratta `Consumer`.

4.4 Aggiungere un MessageProcessor

Per far conoscere al Gestore Personale la presenza di un nuovo tipo di Producer è necessario creare una classe concreta che erediti da `Processor` e, di conseguenza, implementare il metodo `filter_users_by_topic(users: Integer[*], object_kind: String)` che riceve la lista degli utenti interessati al progetto legato al messaggio e la sua tipologia. Questo perchè ogni tecnologia (come Redmine e Gitlab) può avere un tipo diverso di messaggio da elaborare (una messaggio di issue o un messaggio di push).

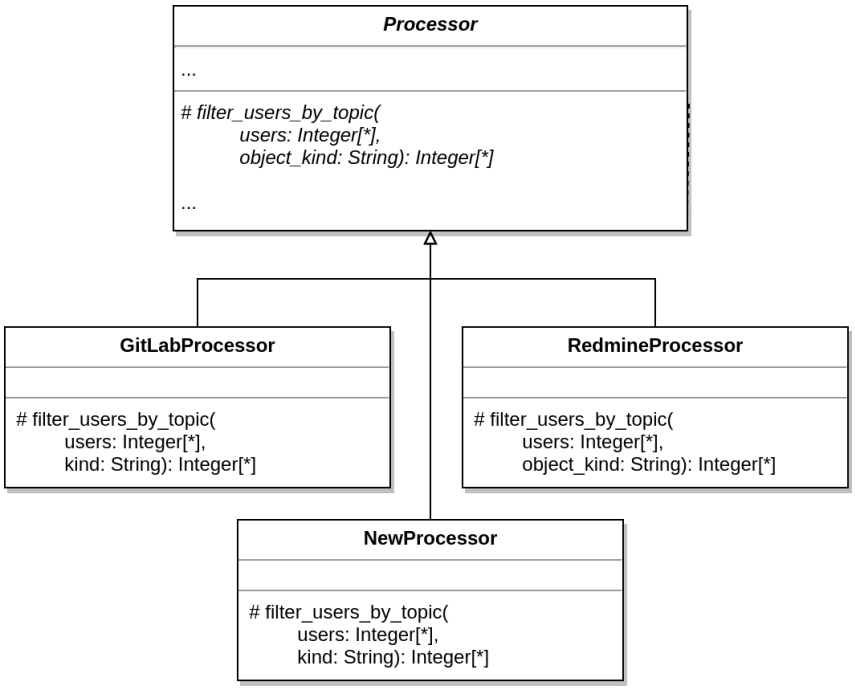

Figura 21: Aggiungere un Processor

5 Test

Questa sezione ha lo scopo di mostrare agli sviluppatori come testare il proprio codice. Con questo proposito verranno descritti i vari test che abbiamo implementato e come eseguirli.
Per eseguire la suite di test, bisogna posizionarsi nella root di Butterfly e eseguire il seguente comando:

```
python3 -m pytest
```

Questo eseguirà tutta la batteria di test del progetto, restituendo il numero di test passati e non passati ed eventuali informazioni relativi ai fallimenti. I test si appoggiano sulla libreria `pytest`.

A Glossario

A

Adapter
Design pattern strutturale con la funzione di adattare un'interfaccia a un'altra, utile quando si usano oggetti di librerie esterne che necessitano di essere adattate alle interfacce del proprio sistema.

Applicativo
Programma software con lo scopo di rendere possibili una o più funzionalità, servizi o strumenti utili.

API Rest
Metodi con cui è possibile fornire l'interazione con le componenti del sistema basate su representational state transfer, dove le risorse sono uniche e indirizzabili mediante URI.

B

Broker
Componente che gestisce i messaggi inviati da Publisher a Subscriber nel relativo modello architetturale. Mette a disposizione i Topic$_G$ nei quali i Publisher inviano i messaggi, mentre i Subscriber possono iscriversi a essi per ricevere i messaggi.

BSON
Formato di scambio dati utilizzato principalmente nel database MongoDB$_G$. È un formato binario per rappresentare semplici strutture.

C

Cluster
È un insieme di computer connessi tra loro tramite una rete telematica. Scopo del cluster è distribuire un'elaborazione molto complessa tra i vari computer, aumentando la potenza di calcolo del sistema e/o garantendo una maggiore disponibilità di servizio, a prezzo di un maggior costo e complessità di gestione dell'infrastruttura: per essere risolto il problema che richiede molte elaborazioni viene infatti scomposto in sottoproblemi separati i quali vengono risolti ciascuno in parallelo.

Container
Consiste nella capacità di eseguire più processi e applicazioni in modo separato per sfruttare al

meglio l'infrastruttura esistente pur conservando il livello di sicurezza che sarebbe garantito dalla presenza di sistemi separati.

D

Docker
Piattaforma che consente di automatizzare il DEPLOYMENT$_G$ di applicazioni all'interno di CONTAINER$_G$ software.

Docker-compose
Definisce le configurazioni necessarie per come devono essere eseguite le immagini contenute nei container DOCKER$_G$, i link e le porte esposte verso l'esterno.

Dockerfile
Definisce le configurazioni necessarie per il container sul quale si andrà a eseguire l'APPLICATIVO$_G$.

DockerHub
Repository di Docker che permette il versionamento delle immagini e la gestione delle build in maniera automatica innescata al push delle modifiche sulla repository del codice (necessita che queste siano collegate fra loro).

E

Event Driven
È un paradigma di programmazione dell'informatica. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica a eventi il flusso del programma è largamente determinato dal verificarsi di eventi esterni.

F

Factory Method
Design Pattern creazionale, in cui l'interfaccia di creazione lascia alle sottoclassi la decisione su quale oggetto istanziare.

J

JSON
Acronimo di JavaScript Object Notation, è un formato adatto all'interscambio di dati fra applicazioni client/server. È basato sul linguaggio JavaScript Standard ma ne è indipendente.

K

Kubernetes
Kubernetes è uno strumento open source di orchestrazione e gestione di container. È stato sviluppato dal team di Google ed è uno dei tool più utilizzati a questo scopo. Kubernetes permette di eliminare molti dei processi manuali coinvolti nel deployment e nella scalabilità di applicazioni contenute in container e di gestire in maniera semplice ed efficiente cluster di host su cui questi vengono eseguiti.

M

Macchina virtuale
Una macchina virtuale (VM) indica un software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale che emula il comportamento di una macchina fisica (PC client o server) grazie all'assegnazione di risorse hardware. In cui alcune applicazioni possono essere eseguite come se interagissero con tale macchina.

Markdown
Linguaggio di markup con una sintassi molto semplice, convertibile in altri formati quali HTML con un TOOL$_G$ omonimo.

Metadato
Particolare dato che descrive insiemi di altri dati.

MongoDB
È un DBMS$_G$ non relazionale, orientato ai documenti. MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON$_G$ con schema dinamico.

MySQL
È un software di tipo server avente il compito di gestire uno o più database. Il suo compito è quello di intervenire, in qualità di intermediario, in ogni operazione sui database, gestendo gli accessi ai dati (filtrando quelli non autorizzati), ad eseguire le interrogazioni ed a restituirne il risultato ove previsto.

O

Open-closed principle
Principio della programmazione ad oggetti che stabilisce quanto segue: una classe dovrebbe essere aperta alle estensioni ma chiusa alle modifiche.

P

PIP
Il Python Package Index è un repository che contiene decine di migliaia di package scritti in Python. Chiunque può scaricare package esistenti o condividere nuovi package su PIP.

Producer
Componente di Butterfly con lo scopo di raccogliere i messaggi e pubblicarli sotto forma di messaggi all'interno dei Topic adeguati.

R

Risorsa
Soggetto consumabile che può essere di varia natura. Nel caso di un progetto software una risorsa può consistere in ore di lavoro o tecnologie utilizzabili.

T

Template Method
Design pattern comportamentale che definisce la struttura di un algoritmo, lasciando alle sotto-classi il compito di definirne alcuni passi.

Topic
Equivalente di "argomento" in italiano. Nel contesto di un BROKER$_G$ si intende un canale di messaggi associato a uno specifico argomento.

W

Webhook
Metodo per aumentare o modificare il comportamento di una pagina o applicazione Web con chiamate HTTP esterne in modo semplice, standardizzato e intelligente (CALLBACK$_G$).