softwareonion@gmail.com

Manuale sviluppatore

Informazioni sul Documento

Versione	v2.0.0
Data approvazione	2019-07-12
Responsabili	Matteo Lotto
Redattori	Alessio Lazzaron Federico Omodei Nicola Zorzo
Verificatori	Nicola Pastore
Stato	Approvazione provvisoria
Lista distribuzione	*prof. Tullio Vardanega* *prof. Riccardo Cardin* *Onion Software* *Imola Informatica S.p.A.*

Scopo del Documento

Lo scopo del documento è di presentare le tecnologie e l'architettura del prodotto software "Butterfly" agli sviluppatori interessati.

Registro delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
v2.0.0	2019-07-12	Matteo Lotto	Responsabile	Documento approvato per il rilascio
v1.1.0	2019-06-29	Nicola Pastore	Verificatore	Verifica superata
v1.0.2	2019-06-28	Nicola Zorzo	Progettista	Rettifica documento dei §3 §4 §5
v1.0.1	2019-06-24	Nicola Zorzo	Progettista	Integrazioni documento dei §2 §3
v1.0.0	2019-06-08	Linpeng Zhang	Responsabile	Documento approvato per il rilascio provvisorio
v0.1.0	2019-06-07	Nicola Pastore	Verificatore	Verifica superata
v0.0.6	2019-06-03	Alessio Lazzaron	Progettista	Correzione derivanti dalla verifica
v0.0.5	2019-05-31	Nicola Pastore	Verificatore	Verifica non superata
v0.0.4	2019-05-30	Alessio Lazzaron	Progettista	Stesura capitoli §5 e integrazione dei vari diagrammi
v0.0.3	2019-05-27	Nicola Zorzo	Progettista	Stesura capitoli §3 §4
v0.0.2	2019-05-25	Federico Omodei	Progettista	Stesura capitoli §1 §2 §6
v0.0.1	2019-05-22	Alessio Lazzaron	Progettista	Creazione del documento

Indice

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Il fine del documento è di presentare l'architettura del prodotto Butterfly fornendo tutte le informazioni necessarie per ampliarlo, correggerlo e migliorarlo.
Verranno descritte le tecnologie utilizzate, le librerie coinvolte, l'architettura di dettaglio e le scelte progettuali.

1.2 Scopo del prodotto

Il prodotto nasce dalla necessità di accentrare e standardizzare le segnalazioni di strumenti di terze parti, i quali Redmine e Gitlab; questi, infatti, forniscono messaggi di notifica specifici e di difficile gestibilità per l'utente finale. Inoltre, in caso di segnalazioni di bug è doveroso assicurarsi che il problema sia risolto tempestivamente, senza che debba accedere ad interfacce specifiche per comprendere. Butterfly si pone lo scopo di risolvere queste problematiche, garantendo una maggiore flessibilità e permettendo una gestione automatizzata e personalizzabile delle segnalazioni.

1.3 Prerequisiti

Per poter comprendere i contenuti di questo documento, facendo in particolar modo riferimento ai diagrammi presenti, è opportuno che il lettore possegga una conoscenza di base del linguaggio UML2.0, del concetto di API REST, e di che cosa sia un Message Broker.

1.4 Glossario

Al fine di evitare possibili ambiguità relative al linguaggio utilizzato nel documento, viene fornito un glossario nell'appendice §A, in questa vengono definiti e descritti tutti i termini con un significato specifico. Per facilitare la comprensione, i termini saranno contrassegnati da una 'G' a pedice.

1.5 Compatibilità

Garantiamo il corretto funzionamento dei servizi che compongono il prodotto Butterfly solo nel caso in cui vengono usate le versioni esatte delle librerie e dei *framework*$_G$ di sviluppo, che saranno elencati nel documento. Il gruppo non si assume quindi nessuna responsabilità nel caso in cui il prodotto risenta di problematiche riconducibili ad una dipendenza di terze parti la cui versione non è supportata.

2 Requisiti minimi di sistema

Di seguito sono elencati i requisiti di sistema minimi richiesti per l'installazione del prodotto; tali requisiti si suddividono in *obbligatori* e *consigliati*. Con *consigliati* si intendono gli strumenti utili allo sviluppo e compilazione del codice in locale, ma che non sono richiesti per l'installazione.

- Requisiti obbligatori
  - Docker v18.06.1;
  - Docker Compose v1.23.2;
  - connessione ad internet.
- Requisiti consigliati
  - Python v3.7.3;
  - PyCharm v2019.1.3.

3 Tecnologie utilizzate

Di seguito verranno descritte brevemente tutte le tecnologie che, dopo una accurata fase di analisi, abbiamo scelto di utilizzare per lo sviluppo del prodotto software.

3.1 Strumenti di sviluppo

3.1.1 Python

Il proponente ha consigliato per l'implementazione del prodotto i seguenti linguaggi:

- Python;
- Java;
- NodeJS.

Dopo un'attenta analisi sui vari vantaggi e svantaggi che ciascun linguaggio comporta abbiamo scelto Python. Le motivazioni principali sono le seguenti:

- è un linguaggio open-source ed è possibile usarlo e distribuirlo senza restrizioni di copyright;
- è un linguaggio multi-paradigma, che supporta sia la programmazione procedurale, sia la programmazione ad oggetti, sia la programmazione funzionale;
- è un linguaggio portabile, ciò è possibile perché si tratta di un linguaggio interpretato, quindi lo stesso codice può essere eseguito su qualsiasi piattaforma purché abbia l'interprete Python installato;
- è un linguaggio molto diffuso e supportato da una community incredibilmente vasta;
- è un linguaggio di alto livello semplice da imparare, e in poco tempo permette di sviluppare software complessi;
- è un linguaggio che a nostro parere ha una bella sintassi.

3.1.2 MongoDB

MongoDB è un database *NoSQL*$_G$ che fornisce un approccio nell'organizzazione e gestione dei dati in grado di coprire la quasi totalità delle esigenze svolte dai database relazionali, garantendo però:

- migliori performance e una maggiore scalabilità orizzontale;
- un'attività di sviluppo più semplice e veloce rispetto ai database tradizionali.

3.1.3 Apache Kafka

Sistema di messaggistica open source distribuito che consente di creare applicazioni in tempo reale tramite flussi di dati. Il progetto 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.

3.1.4 Ruby

È il linguaggio utilizzato per la codifica di uno script che permette l'integrazione del prodotto con Redmine. Proprio come Python, Ruby:

- è un linguaggio open source;
- è un linguaggio semplice e flessibile;
- possiede una comunità molto attiva, che mette a disposizione vaste librerie di qualità.

3.2 Strumenti di gestione

3.2.1 Docker

Il proponente ha suggerito di utilizzare *Docker* per la semplicità di utilizzo e per mantenere le componenti totalmente isolate. *Docker* è un progetto open-source che automatizza il *deployment*$_G$ di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo. Docker utilizza le funzionalità di isolamento delle risorse del kernel per consentire a *container*$_G$ indipendenti di coesistere sulla stessa istanza del sistema operativo, evitando l'installazione, la manutenzione e l'overhead di una macchina virtuale. La sua configurazione avverrà tramite un file in cui verranno specificate informazioni come sistema operativo, script di avvio, numero di istante e altre specifiche, chiamato *Dockerfile*.

3.3 Strumenti condivisi tra i servizi

3.3.1 JSON

JSON è il formato di serializzazione testuale scelto per l'interfacciamento con le REST API esposte dal componente Gestore Personale. Esso è un formato estremamente diffuso per lo scambio dei dati in applicazioni client-server. Si basa su dizionari, ovvero coppie chiave/valore, e supporta i tipi booleano, stringa, numero, e lista. È semplice e facilmente leggibile dalle persone, inoltre non necessita di alcun processo di compilazione particolare per essere modificato.

3.4 Librerie esterne

Le librerie elencate saranno già installate nel cluster fornito dalla proponente.

3.4.1 Flask

È un web *framework*$_G$ leggero scritto in Python. Viene utilizzato:

- dai componenti *Producer*$_G$ per restare in ascolto degli *webhook*$_G$;
- per fornire un'interfaccia web e delle REST API.

Per maggiori informazioni visitare il link:

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

Flask è installabile con il comando

```
pip install Flask
```

si consiglia la versione 1.0.2.

3.4.2 PyMongo

È una libreria per l'utilizzo di MongoDB con Python. Il database viene utilizzato:

- dai componenti TelegramBot, SlackBot per mantenere la corrispondenza tra username di Telegram e email di Slack con i relativi id per l'invio di messaggi;
- dal Gestore Personale che deve interagire con il database per recuperare le informazioni per processare i messaggi;
- dalle interfacce utente per permettere l'interazione con il database.

Per maggiori informazioni visitare il link:

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

Pymongo è installabile con il comando

```
pip install PyMongo
```

la versione consigliata è la 3.8.0.

3.4.3 Mongoengine

È una libreria che fornisce un'astrazione del database. MongoEngine possiede diverse funzionalità che permettono una maggiore flessibilità e semplicità nell'interazione con il database, nascondendone i dettagli di basso livello.
Per maggiori informazioni visitare il link:

http://docs.mongoengine.org/apireference.html

Mongoengine è installabile con il comando

```
pip install mongoengine
```

la versione consigliata è la 0.17.0.

3.4.4 PyTest

È uno dei framework più utilizzati di Python per l'esecuzione dei test. In particolare permette una più facile gestione e implementazione dei test rispetto alla libreria standard *unittest*.
Per maggiori informazioni visitare il link:

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

Pytest è installabile con il comando

```
pip install pytest
```

la versione consigliata è la 4.4.1.

3.4.5 Requests

È una libreria per la gestione di richieste HTTP. Rende possibile effettuare ogni tipo di richiesta (GET, POST, DELETE, PUT). Viene utilizzata per testare le API Rest e per simulare la ricezione dei webhook da parte dei Producer. Per maggiori informazioni visitare il link:

https://2.python-requests.org//en/master/

Requests è installabile con il comando

```
pip install requests
```

3.4.6 PyTelegramBotApi

È una libreria che permette di interfacciarsi in modo semplice alle API per la gestione dei bot di Telegram.
Per maggiori informazioni visitare il link:

https://github.com/eternnoir/pyTelegramBotAPI

PyTelegramBotApi è installabile con il comando

```
pip install pyTelegramBotApi
```

la versione consigliata è la 3.6.6.

3.4.7 Kafka-python

È una libreria in Python che contiene un client per il sistema di elaborazione del flusso distribuito Apache Kafka. kafka-python è progettato per funzionare in modo molto simile al client java ufficiale.
Per maggiori informazioni visitare il link:

https://github.com/dpkp/kafka-python

Kafka-python è installabile con il comando

```
pip install Kafka-python
```

la versione consigliata è la 1.4.6.

3.4.8 Jinja

Jinja2 è un motore di template completo per Python, usato per l'interfaccia web. Ha un supporto unicode completo, un ambiente di esecuzione *sandbox*$_G$ integrato e ampiamente utilizzato dalla community.
Per maggiori informazioni visitare il link:

http://jinja.pocoo.org/

Jinja è integrato con Flask, quindi non necessita di nessun comando extra per essere installato.

3.4.9 Pylint

È una libreria che permette l'analisi di codice sorgente, bug e qualità per il linguaggio di programmazione Python.
Per maggiori informazioni visitare il link:

https://www.pylint.org/

Pylint è installabile con il comando

```
pip install pylint
```

la versione consigliata è la 2.3.1.

3.4.10 Slackclient

È un kit di sviluppo per l'interfaccia con l'API Slack Web e l'API Real Time Messaging su Python 3.6 e versioni successive.
Per maggiori informazioni visitare il link:

https://github.com/slackapi/python-slackclient

Slackclient è installabile con il comando

```
pip install slackclient
```

la versione consigliata è la 2.0.1.

3.4.11 Slack-bot

È una libreria che permette di creare e gestire un bot di Slack.
Per maggiori informazioni visitare il link:

https://pypi.org/project/slack-bot/

Slack-bot è installabile con il comando

```
pip install slack-bot
```

la versione consigliata è la 0.0.7.

4 Architettura

L'architettura software utilizzata è *event-driven*$_{G}$ *con topologia broker*$_{G}$.
Ogni componente che fa parte del sistema conosce solamente l'esistenza del *broker* e comunica con esso attraverso l'invio o la ricezione di messaggi che rispecchiano l'avvenimento di un determinato evento. Proprio per questo motivo ogni componente è trattato come un'unità isolata e asincrona.

Questo tipo di architettura è stata scelta in base alle seguenti considerazioni:

- **dominio del problema**: il progetto *Butterfly* prevede l'implementazione di componenti specifici (*producer e consumer*) per servizi e canali di comunicazione completamente scollegati tra loro, che non hanno bisogno di conoscere la presenza l'uno dell'altro;
- **scalabilità**: l'indipendenza dei servizi/canali di comunicazione e il comportamento asincrono dei componenti permette una buona scalabilità del sistema.

L'architettura dell'applicativo si suddivide in tre macro componenti analizzati nelle sezioni successive:

- ***producers***: rappresentano i componenti che rimangono in ascolto del *webhook*$_{G}$ proveniente dal servizio loro assegnato (*Redmine o GitLab*), producono il messaggio relativo all'evento descritto da quest'ultimo e lo inviano al *broker*;
- ***consumers***: rappresentano i componenti che ricevono il messaggio relativo a un determinato evento dal *broker* e lo inoltrano ai destinatari finali attraverso il canale di comunicazione designato (*email, Telegram o Slack*);
- ***gestore personale***: rappresenta i componenti che si occupano della manipolazione dei messaggi inviati dai *producer* al *broker* e da quest'ultimo ai *consumer*. Nello specifico, hanno il compito di filtrare i messaggi in entrata per stabilire il *consumer* appropriato a cui viene inviato il messaggio e la lista dei destinatari finali.

4.1 Producer

I *producer* sono i componenti dell'architettura che hanno lo scopo di inviare messaggi su Kafka in formato *JSON*$_G$. Restando in ascolto degli *webhook*$_G$ specifici di ogni applicativo (e.g. Redmine e Gitlab), generano messaggi personalizzati secondo i campi d'interesse prestabiliti. I producer implementati nel package *producers* sono tre:

- RedmineProducer;
- GitLabProducer.

4.1.1 Diagramma delle classi

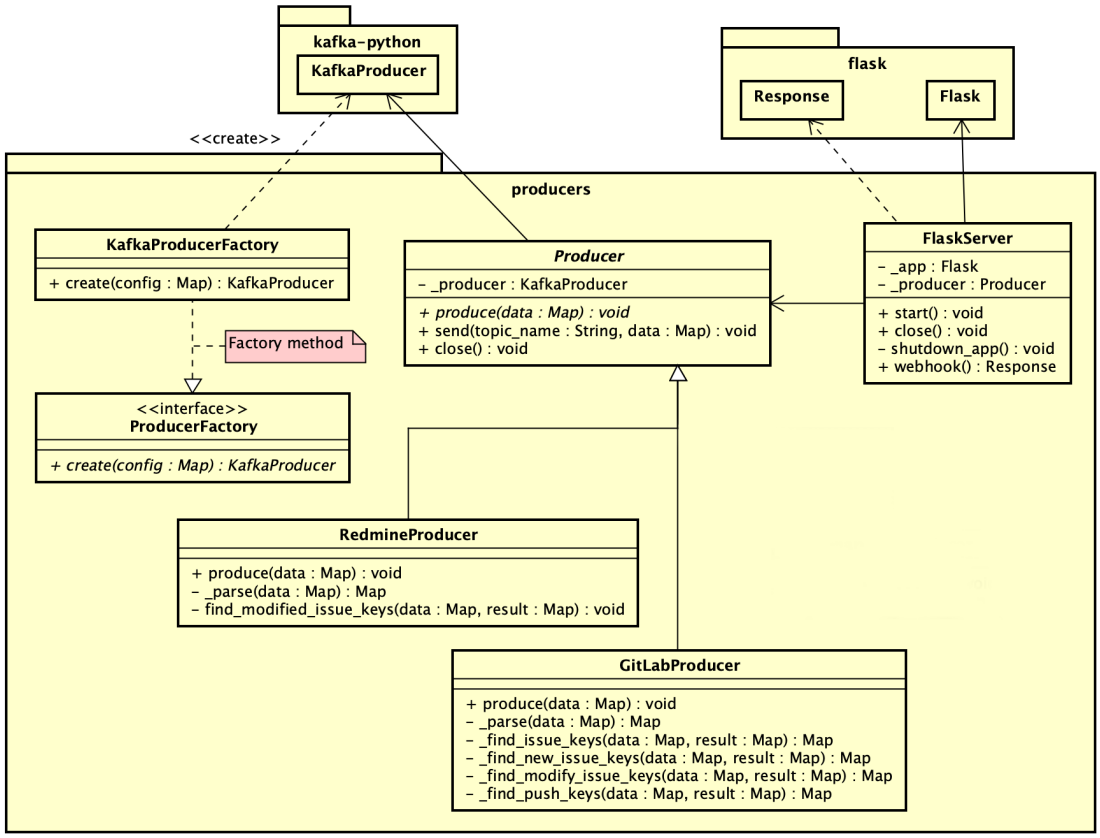

Figura 1: Diagramma delle classi dei componenti *producer*

4.1.2 Chiamate

4.1.3 Design pattern utilizzati

4.1.3.1 Factory Method

L'interfaccia `ProducerFactory` mette a disposizione un metodo astratto *`create(config: map)`* implementato nella classe concreta `KafkaProducerFactory`. Questa classe ha il compito di istanziare un KafkaProducer a partire da una mappa contenente le informazioni per la configurazione. L'utilizzo di questo pattern consente agli utilizzatori di avere un riferimento all'interfaccia, in modo che il codice sia più estendibile qualora vi fosse necessità di istanziare particolari tipi di KafkaProducer, o qualora dovesse cambiare il formato dei file di configurazione.

4.1.4 Diagrammi di sequenza

I producer hanno lo scopo di ascoltare gli webhook provenienti dai progetti di Redmine e GitLab. Alla ricezione di un webhook viene chiamato un opportuno metodo *webhook()* nel FlaskServer che a sua volta chiamerà l'opportuno metodo *produce()* attraverso una chiamata polimorfa, risultando estremamente fedele al *open/closed principle*$_{G}$. Si hanno quindi i seguenti flussi di esecuzione, a seconda del servizio di provenienza del webhook.

4.1.4.1 Redmine

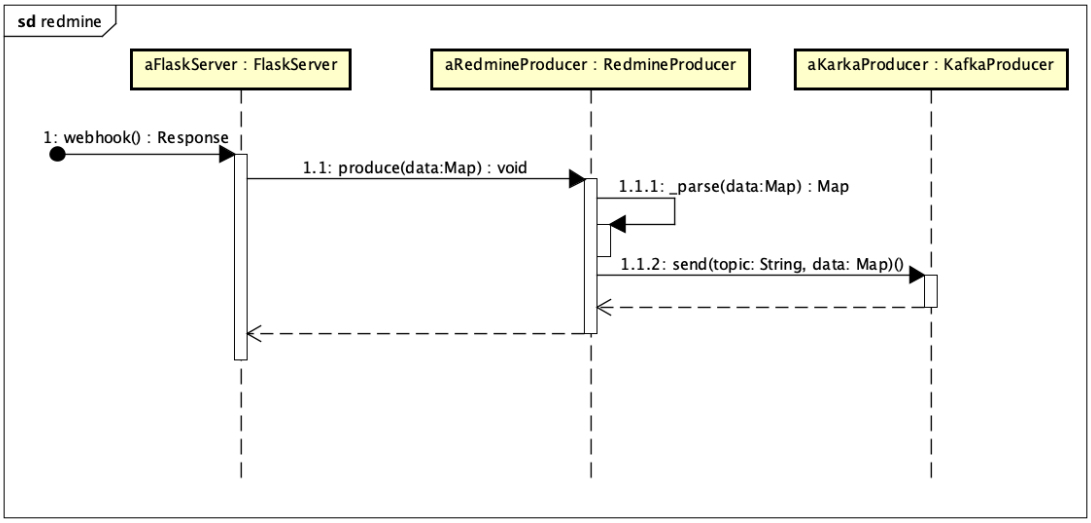

Figura 2: Diagramma di sequenza per la ricezione di un messaggio su Redmine e invio del messaggio su Kafka

4.1.4.2 GitLab

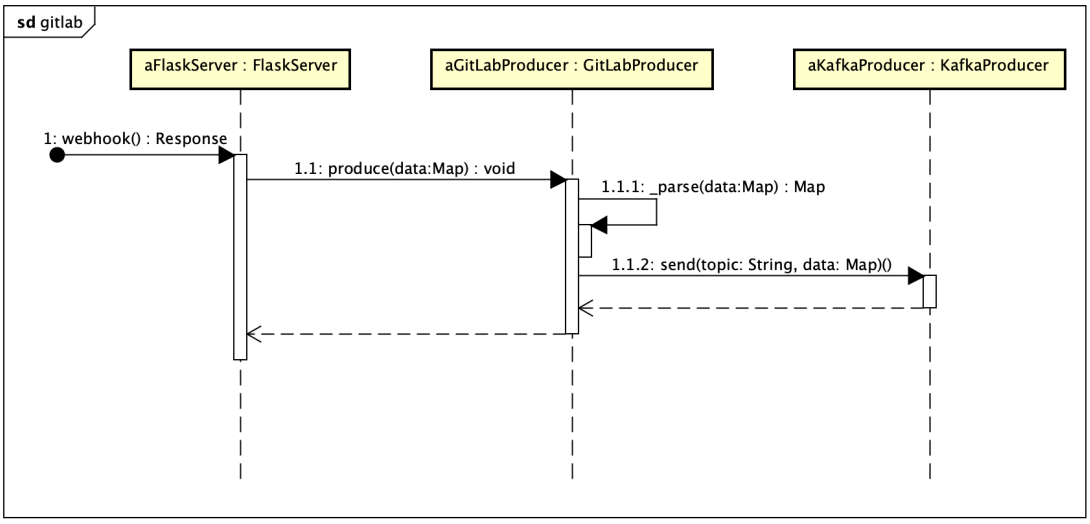

Figura 3: Diagramma di sequenza per la ricezione di un messaggio su Gitlab e invio del messaggio su Kafka

4.2 Gestore personale

Il gestore personale ha lo scopo di ricevere i messaggi inviati dai *producer* su Kafka, interagire con il database *mongodb* per inserire le opportune informazioni sui destinatari interessati e reinstradarli nuovamente in Kafka, delegando ai *consumer* il compito di inviare questi messaggi. La comunicazione avviene interamente utilizzando il formato JSON. Nello specifico, il package *manager* contiene un producer, *DispatcherProducer*, e un consumer, *DispatcherConsumer*, che rispettivamente servono a ricevere e inviare messaggi da Kafka e reinserirli nei $topic_G$ dove rimarranno in ascolto altri consumer; ci sarà inoltre un *Processor* che aggiunge opportune informazioni sui destinatari, ottenute interrogando il database. L'algoritmo con cui il gestore personale ricodifica il messaggio è attualmente influenzato dalle priorità dei progetti, dalla piattaforma di ritorno del messaggio sul quale ricevere la notifica per ogni utente e, se presenti, dai giorni d'indisponibilità.

4.2.1 Diagramma delle classi

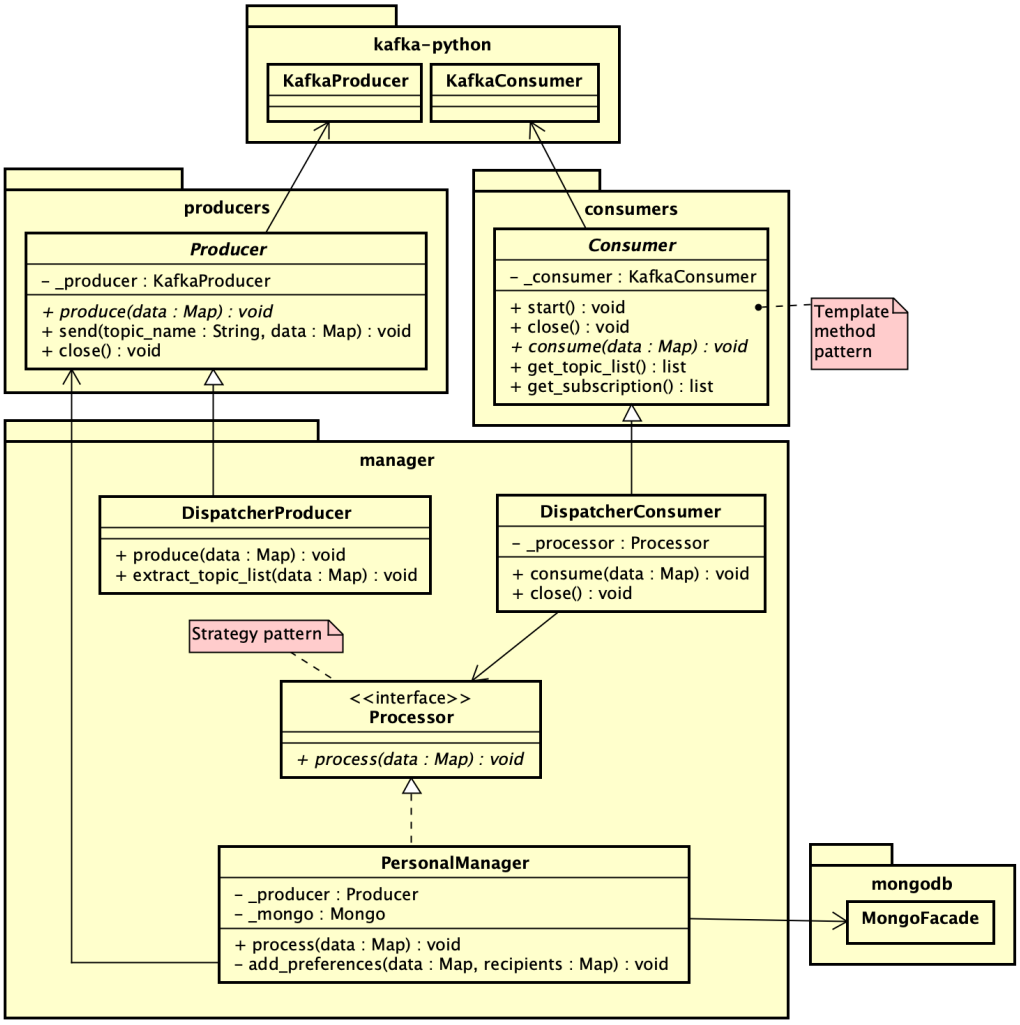

Figura 4: Diagramma delle classi del gestore personale

4.2.2 Design pattern utilizzati

4.2.2.1 Template Method

Il Template Method Pattern, utilizzato dalla classe astratta `Consumer`, è illustrato nel §4.3.2.1.

4.2.3 Diagramma di sequenza

Il DispatcherConsumer rimane in ascolto dei topic su cui scrivono i componenti Producer. All'arrivo di un messaggio viene chiamato il metodo consume() e successivamente si sfrutta il pattern Strategy per processare il messaggio: viene chiamato quindi l'attuale algoritmo di selezione dei destinatari interessati, che potrebbe cambiare in futuro. Infine si aggiungono i destinatari al messaggio e lo si invia nuovamente su un topic di Kafka: per fare questo dinamicamente verrà selezionato il Producer appropriato, in questo caso DispatcherProducer.

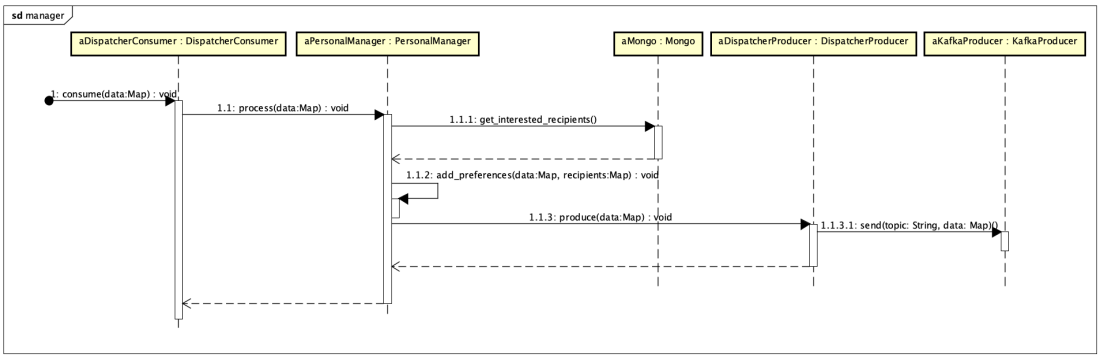

Figura 5: Diagramma di sequenza per l'elaborazione di un messaggio da parte del gestore personale

4.3 Consumer

I *consumer* sono i componenti dell'architettura che hanno lo scopo di restare in ascolto su un topic specifico di Kafka per ogni applicativo di risposta implementato. All'arrivo di un messaggio precedentemente elaborato dal gestore personale si occupano di inviarlo ai destinatari finali, utilizzando una classe opportuna di supporto. Queste classi, nel caso di TelegramBot e SlackBot, dovranno mantenere l'integrità con il database in modo da correlare username di Telegram ed email di Slack con gli id specifici necessari per l'invio dei messaggi. I consumer implementati sono tre:

- TelegramConsumer;
- EmailConsumer;
- SlackConsumer.

4.3.1 Diagramma delle classi

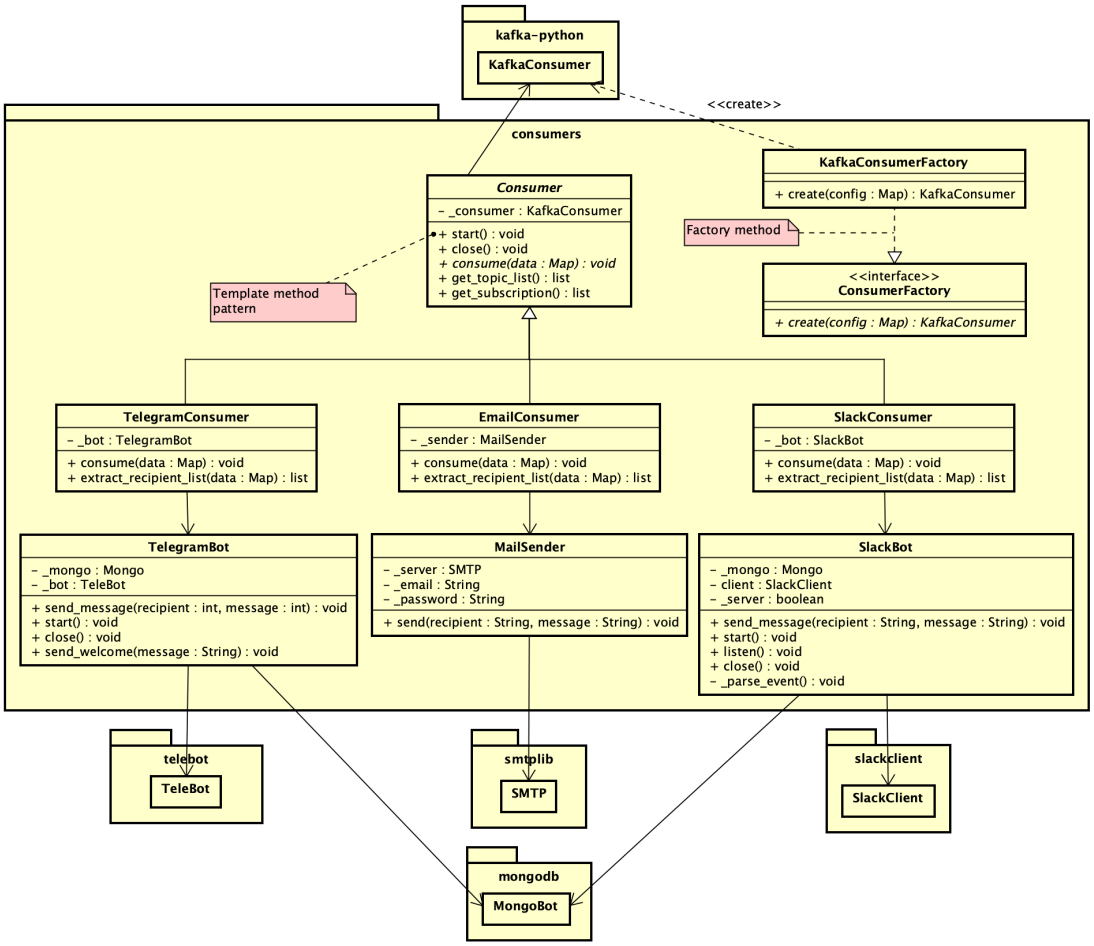

Figura 6: Diagramma delle classi dei componenti *consumer*

4.3.2 Design pattern utilizzati

4.3.2.1 Template Method

Il metodo `start()`, che implementa il pattern *Template Method*, ha comportamento identico per tutte e tre le classi consumer. È stato quindi scelto di incapsulare tale metodo in una classe

base astratta, `Consumer` (supertipo astratto di `TelegramConsumer`, `EmailConsumer` e `SlackConsumer`). In particolare si vuole che tutti i consumer restino in attesa di un arrivo di un messaggio per consumarlo. Poiché tale comportamento differisce solamente per il modo di consumare il messaggio, è stato deciso di adottare questo pattern, implementando quindi il metodo `start()` nella classe base astratta, e lasciando il metodo `consume()` astratto. Tale scelta comporta i seguenti benefici:

- un maggior **riuso del codice**;
- **raggruppano i comportamenti in comune** tra diverse classi, rendendosi indispensabili in contesti come, ad esempio, in classi di librerie.

4.3.2.2 Factory Method

L'interfaccia `ConsumerFactory` mette a disposizione un metodo astratto *`create(config: map)`* implementato nella classe concreta `KafkaConsumerFactory`. Questa classe ha il compito di istanziare un KafkaConsumer a partire da una mappa contenente le informazioni per la configurazione. L'utilizzo di questo pattern consente agli utilizzatori di avere un riferimento all'interfaccia, in modo che il codice sia più estendibile qualora vi fosse necessità di istanziare particolari tipi di KafkaConsumer, o qualora dovesse cambiare il formato dei file di configurazione.

4.3.3 Diagramma di sequenza

I consumer hanno lo scopo di rimanere in ascolto di un opportuno topic su Kafka. All'arrivo di un messaggio viene chiamato il metodo *consume()*, che si occupa di estrapolare informazioni sul destinatario per poi delegare l'invio del messaggio ad un appropriato servizio. Si hanno quindi i seguenti flussi di esecuzione, a seconda del servizio di invio del messaggio.

4.3.4 Telegram

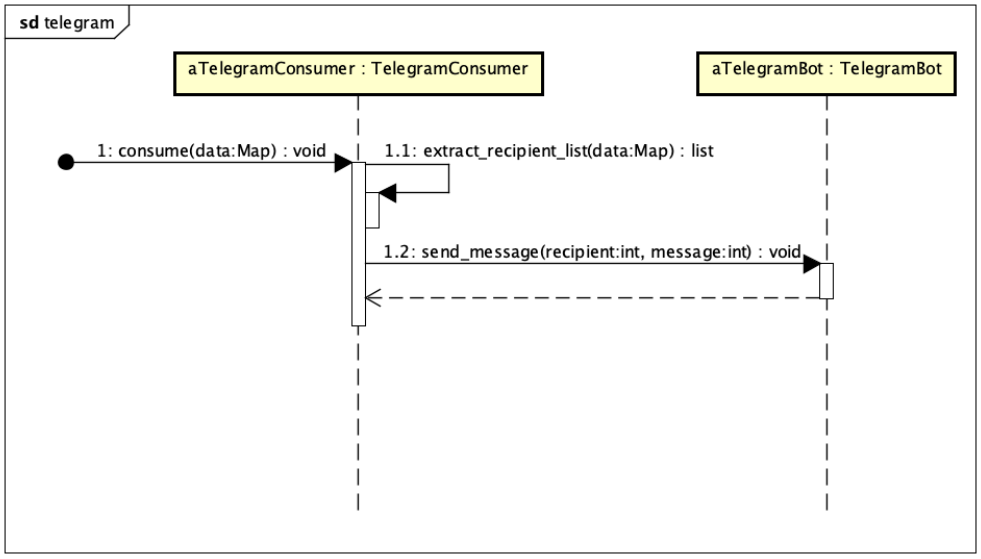

Figura 7: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via telegram

4.3.5 Email

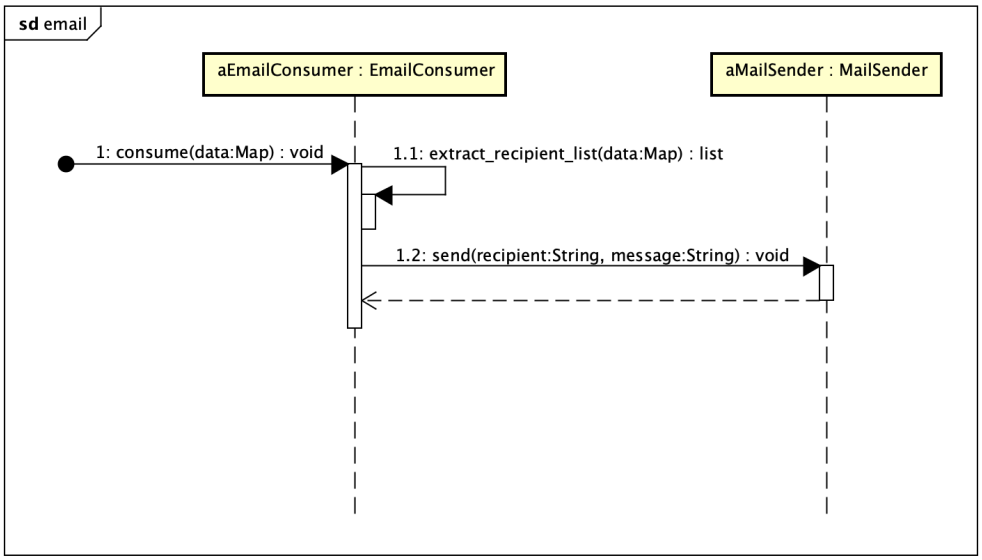

Figura 8: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via email

4.3.6 Slack

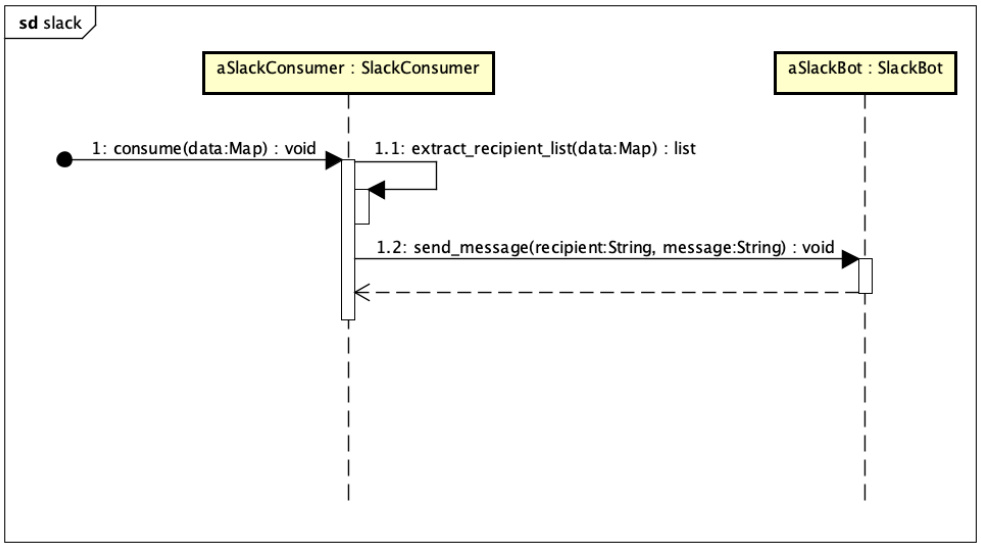

Figura 9: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via slack

4.3.7 Database

Per gestire il database che contiene contatti, progetti ed iscrizioni sono state create classi opportune per suddividere le responsabilità di ogni classe, in accordo con il *Single responsibility principle*$_G$. Inoltre è stata utilizzata una classe *MongoBot* che contiene i metodi per mantenere la correlazione tra username di Telegram, email di Slack con gli id specifici di Telegram, Slack. Naturalmente queste classi hanno dipendenze con il package esterno mongoengine, che fornisce un'astrazione per l'interazione con Mongo.

4.3.8 Diagramma delle classi

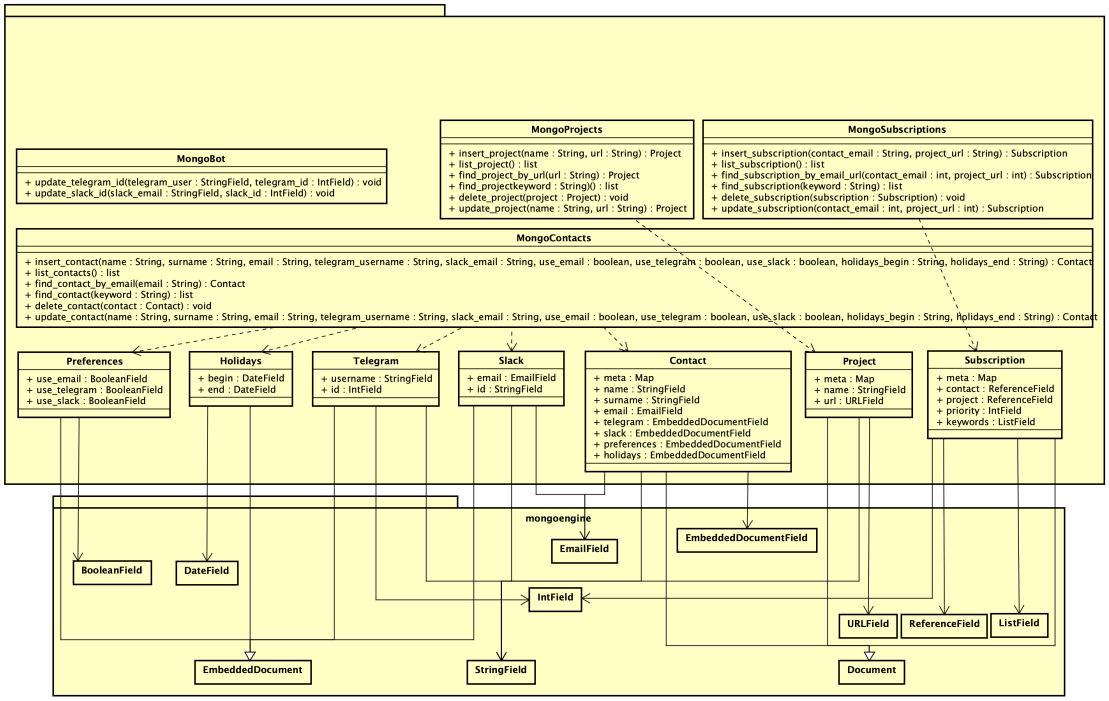

Figura 10: Diagramma delle classi del database

4.3.9 Interfacce ed architettura MVC

Onion Software ha sviluppato una piccola applicazione web, che ha lo scopo di essere user-friendly per l'utilizzatore finale. Il team ha anche sviluppato una *CLI*$_G$, per poter comunicare direttamente con il gestore personale tramite riga di comando.
Sia l'interfaccia web che quella a riga di comando permettono inserimento, modifica, visualizzazione e cancellamento dei dati del modello.

4.3.9.1 Diagramma delle classi

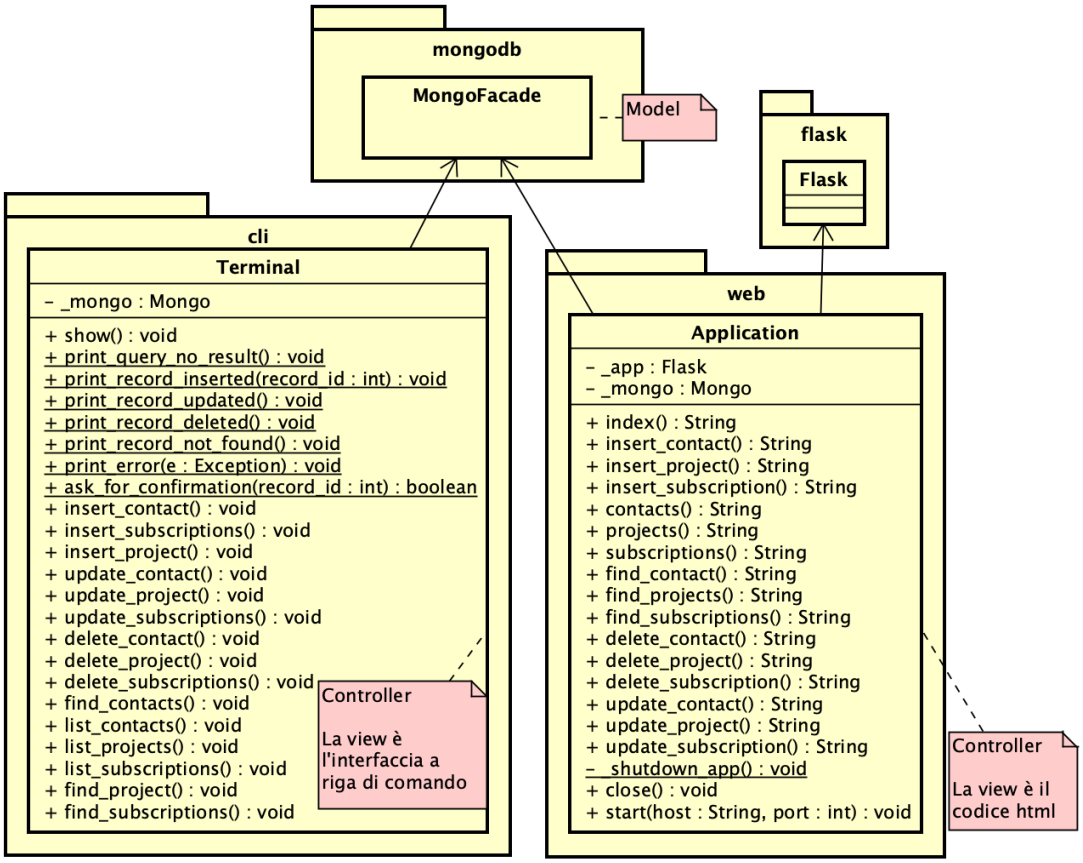

Figura 11: Diagramma delle classi dell'architettura MVC

4.3.9.2 Design pattern utilizzati

4.3.9.2.1 Model View Controller

La scelta di questo pattern è cruciale in quanto permette a *Butterfly* di essere totalmente indipendente dal modo in cui si vuole visualizzare qualsiasi informazione e manipolare dati, ad opera dell'utente finale.
Come suggerisce il nome, le componenti di questo pattern sono tre:

- **model**, che incorpora i dati, nel nostro caso la classe *MongoFacade*.
- **view**, che permette all'utente di interagire con i dati contenuti nel model. Nel nostro caso esso è rappresentata dalle pagine HTML o dal terminale.
- **controller**, che esegue le operazioni decise dall'utente finale comunicando con la view e manipolando i dati contenuti nel model. Questa funzionalità è offerta dalle classi *Application* e *Terminal*.

5 Estendere Butterfly

Butterfly è stato progettato seguendo i principi $SOLID_G$, risultando quindi estremamente facile da estendere.

5.0.1 Aggiungere un Producer

Per aggiungere un componente Producer basterà estendere la classe astratta *Producer*, implementando opportunamente il metodo *produce(data: Map)* in modo che si produca un messaggio nel formato desiderato. Ad esempio, il webhook in arrivo può essere filtrato a seconda delle informazioni di interessato.

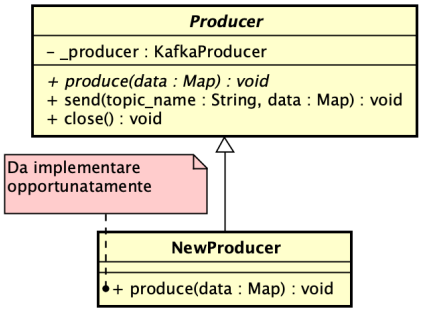

Figura 12: Aggiungere un Producer

5.0.2 Modificare criterio di selezione dei destinatari

Per cambiare il criterio o l'algoritmo di selezione dei destinatari è sufficiente implementare l'interfaccia *Processor*. Questo dovrà processare il messaggio aggiungendo i destinatari. È consigliato l'utilizzo di un *MongoFacade* e di un *Producer* per poter interagire con il database e inviare il messaggio nuovamente su kafka.

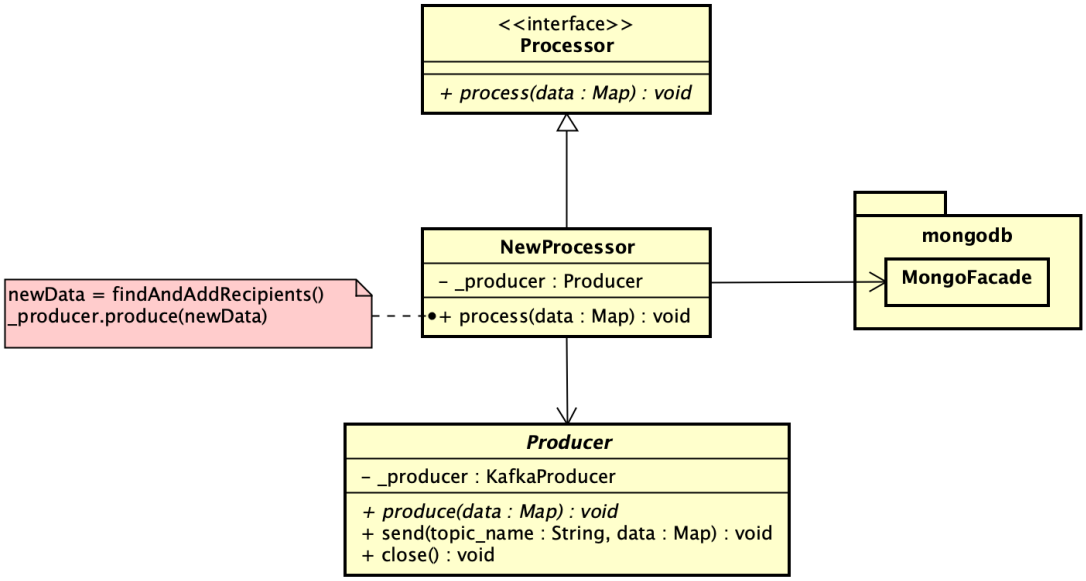

Figura 13: Modificare criterio di selezione dei destinatari

5.0.3 Aggiungere un Consumer

Per aggiungere un componente Consumer basterà estendere la classe astratta *Consumer*, implementando opportunamente il metodo *consume(data: Map)* che viene richiamato nel metodo *start()* (è stato adottato un pattern template method). Tale implementazione del metodo dovrà:

1. estrapolare i destinatari dal messaggio;
2. inviare il messaggio ai destinatari coretti.

Naturalmente è consentito, e consigliato, introdurre una classe di supporto per inviare messaggi attraverso servizi specifici (ad esempio TelegramBot).

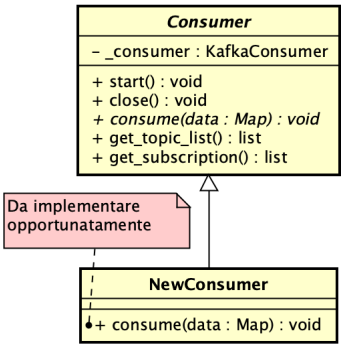

Figura 14: Aggiungere un consumer

6 Test

6.1 Aggiungere nuovi test

Creare nuovi test per *Butterfly* è molto semplice ed intuitivo: è sufficiente scriverli in file che andranno ubicati nella cartella *tests/*.
Per poter eseguire i test è prerequisito necessario avere installato nell'ambiente d'esecuzione la libreria *pytest* e due suoi plugin, in caso contrario sarà sufficiente installarla con il gestore di pacchetti python tramite il comando:

```
pip install pytest pip install pytest-cov pip install pytest-mock
```

È buona prassi nominare i file contenenti test anteponendo o posponendo al loro specifico nome la parola *test*, come mostrato di seguito: *test_nomefile.py* o, alternativamente, *nomefile_test.py*. Questa nomenclatura, oltre a rendere più comprensibile il contenuto dei file ai manutentori, permette un facile riconoscimento automatico dei test da parte della libreria designata alla loro gestione.

Per quanto riguarda le regole di sintassi ed altre funzionalità più avanzate per la creazione di test, si rimanda alla documentazione ufficiale della libreria pytest:

https://docs.pytest.org/en/latest/contents.html#toc

6.2 Eseguire test esistenti

Per eseguire tutta la suite di test disponibile per *Butterfly* è sufficiente:

1. collocarsi nella directory principale (root) del progetto;
2. da terminale, eseguire il seguente comando:

```
python -m pytest
```

Successivamente verranno mostrati i risultati derivanti dall'esecuzione di tutti i test contenuti nella cartella *tests/*.

A Glossario

C

CLI

Una interfaccia a riga di comando (dall'inglese Command Line Interface, acronimo CLI) o anche console, a volte detta semplicemente riga di comando e, impropriamente, prompt dei comandi, è un tipo di interfaccia utente caratterizzata da un'interazione testuale tra utente ed elaboratore. L'utente impartisce comandi testuali in input mediante tastiera alfanumerica e riceve risposte testuali in output dall'elaboratore mediante display o stampante alfanumerici.

Consumer

Il consumer è un componente del pattern *Producer/Consumer* e consiste nel componente che consuma gli elementi che il producer ha inserito nella coda.

Container

Nei sistemi di virtualizzazione che operano a livello di sistema operativo, un container è un'istanza isolata nello spazio utente.

D

Deployment

Consegna o rilascio al cliente, con relativa installazione e messa in funzione o esercizio, di una applicazione o di un sistema software tipicamente all'interno di un sistema informatico aziendale.

E

Event-driven

La programmazione a eventi è un paradigma di programmazione che fa in modo di determinare il flusso del programma dal verificarsi di eventi esterni, a differenza di un programma tradizionale, in cui l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore.

F

Feed

Un'unità di informazioni, formattata secondo specifiche precedentemente stabilite, al fine di rendere interpretabile e interscambiabile il contenuto fra diverse applicazioni o piattaforme.

Framework

Nello sviluppo software, un framework è un'architettura logica di supporto (spesso un'implementazione logica di un particolare design pattern), su cui un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore.

J

Json

Acronimo per *JavaScript Object Notation*, è un formato adatto all'interscambio di dati fra applicazioni client/server.

N

NoSQL

Architettura database caratterizzata dal fatto di non utilizzare il modello relazionale, di solito usato dalle basi di dati tradizionali. L'espressione "NoSQL" fa riferimento al linguaggio SQL, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale.

O

Open/Closed principle

Nella programmazione orientata agli oggetti, il principio aperto/chiuso (open/closed principle, abbreviato con OCP) afferma che le entità (classi, moduli, funzioni, ecc.) software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche; in maniera tale che un'entità possa permettere che il suo comportamento sia modificato senza alterare il suo codice sorgente.

P

Producer

Il producer è un componente del pattern *Producer/Consumer* e consiste nel componente che fornisce gli oggetti alla coda (che verrà successivamente consumata dal consumer)

S

Sandbox

Ambiente adibito alla fase di test e sviluppo del prodotto software.

Single responsibility principle

Nella programmazione orientata agli oggetti, il principio di singola responsabilità (single responsibility principle, abbreviato con SRP) afferma che ogni elemento di un programma (classe, metodo, variabile) deve avere una sola responsabilità, e che tale responsabilità debba essere interamente incapsulata dall'elemento stesso. Tutti i servizi offerti dall'elemento dovrebbero essere strettamente allineati a tale responsabilità.

SOLID Principles

L'acrostico SOLID si riferisce ai "primi cinque principi" dello sviluppo del software orientato agli oggetti. Tali principi vengono detti SOLID principles (letteralmente "principi solidi"). La parola è un acrostico che serve a ricordare tali principi (Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion)

T

Topologia broker

La topologia broker è un pattern architetturale dove il flusso dei messaggi è distribuito dai componenti che processano gli eventi in una modalità a catena, fino ad arrivare al broker dei messaggi, come ad esempio *Apache Kafka*.

W

Webhook

Un Webhook (in italiano letteralmente: "uncino del web") nella programmazione web è un metodo per aumentare o alterare il comportamento di una pagina web, o di un'applicazione web, con chiamate di ritorno personalizzate. Queste chiamate di ritorno possono essere mantenute, modificate e gestite da utenti di terze parti e chi le sviluppa non fa necessariamente parte del sito o applicazione d'origine.