ONION

SOFTWARE

softwareonion@gmail.com

Allegato tecnico

Informazioni sul documento

Versione	1.0.0
Data approvazione	2019-06-04
Responsabili	Linpeng Zhang
Redattori	Federico Brian
	Matteo Lotto
	Nicola Pastore
	Nicola Zorzo
Verificatori	Federico Omodei
	Alessio Lazzaron
Stato	Approvato
Lista distribuzione	*prof. Riccardo Cardin*
	Onion Software

Scopo del documento

Il presente documento contiene le scelte architetturali di Onion Software, adeguatamente motivate, per la realizzazione del progetto Butterfly. Comprende i design pattern e i diagrammi di attività, sequenza, classi e package.

Indice

1 Introduzione **1**
1.1 Scopo del documento . . . 1
1.2 Scopo del prodotto . . . 1
1.3 Glossario . . . 1
1.4 Riferimenti informativi . . . 1
1.5 Riferimenti normativi . . . 1

2 Architettura del prodotto **2**
2.1 Descrizione generale . . . 2
2.2 Diagramma dei package generale . . . 3
2.2.1 Architettura delle classi interfaccia . . . 3
2.2.1.1 Diagramma delle classi . . . 4
2.2.1.2 Design pattern utilizzati . . . 4
2.2.1.2.1 Model View Controller . . . 4
2.3 Architettura di `manager` . . . 4
2.3.1 Diagramma delle classi . . . 5
2.3.2 Design pattern utilizzati . . . 5
2.3.2.1 Strategy . . . 5
2.3.2.2 Template Method . . . 6
2.3.2.3 Object Adapter . . . 6
2.3.3 Diagramma di sequenza . . . 6
2.4 Architettura dei servizi `producer` e `consumer` . . . 6
2.4.1 Producer . . . 6
2.4.1.1 Diagramma delle classi . . . 7
2.4.1.2 Design pattern utilizzati . . . 7
2.4.1.2.1 Factory Method . . . 7
2.4.1.2.2 Strategy . . . 7
2.4.1.2.3 Object adapter . . . 8
2.4.1.3 Diagrammi di sequenza . . . 8
2.4.1.3.1 Redmine . . . 8
2.4.1.3.2 GitLab . . . 9
2.4.1.3.3 SonarQube . . . 9
2.4.2 Consumer . . . 9
2.4.2.1 Diagramma delle classi . . . 10
2.4.2.2 Design pattern utilizzati . . . 10
2.4.2.2.1 Template Method . . . 10
2.4.2.2.2 Object adapter . . . 10
2.4.2.2.3 Factory Method . . . 10
2.4.2.3 Diagramma di sequenza . . . 11
2.4.2.3.1 E-mail . . . 11
2.4.2.3.2 Telegram . . . 11
2.4.2.3.3 Slack . . . 12

Elenco delle figure

1 Diagramma dei package dell'architettura generale . . . 3
2 Diagramma delle classi del componenti interfaccia . . . 4
3 Diagramma delle classi del componente *manager* . . . 5
4 Diagramma di sequenza del componente *manager* . . . 6
5 Diagramma delle classi dei componenti *producer* . . . 7
6 Diagramma di sequenza del componente *producer* Redmine . . . 8
7 Diagramma di sequenza del componente *producer* GitLab . . . 9

8 Diagramma di sequenza del componente *producer* SonarQube 9
9 Diagramma delle classi dei componenti *consumer* 10
10 Diagramma di sequenza del componente *consumer* e-mail 11
11 Diagramma di sequenza del componente *consumer* Telegram 11
12 Diagramma di sequenza del componente *consumer* Slack 12

1 Introduzione

1.1 Scopo del documento

Il documento ha lo scopo di descrivere in maniera dettagliata, coesa e coerente le peculiarità del prodotto software *Butterfly*, sviluppato dal team *Onion Software*. Si descriveranno i *design pattern*$_G$ utilizzati e si illustreranno i *diagrammi di attività*$_G$, *sequenza*$_G$ e *package*$_G$. Infine, verrà effettuato un confronto tra lo stato d'avanzamento dello sviluppo del prodotto operato in sede di *Technology Baseline*$_G$ e l'attuale *Product Baseline*$_G$, ponendo particolare attenzione su casi d'uso e requisiti soddisfatti.

1.2 Scopo del prodotto

Una realtà *enterprise* che opera nel campo dell'*information technology* implementa processi di *Continuous Integration* e *Continuous Delivery*[1] per farlo con efficacia, utilizza degli strumenti che aiutino l'automatizzazione di certe operazioni. La maggior parte di questi strumenti fornisce "out-of-the-box" dei meccanismi di segnalazione che permettono la notifica di eventuali problematiche riscontrate nelle varie fasi. Ognuno di questi strumenti ha, però, un proprio meccanismo specifico di esposizione dei messaggi/segnalazioni, spesso con limitate capacità di configurazione.
I limiti sopra indicati sfociano spesso nella necessità per l'utente di interfacciarsi con molteplici dashboard specifiche, ognuna delle quali con propria struttura. Spesso, poi, queste interfacce sono anche di difficile accessibilità, sia a causa della complessità applicativa, sia a causa di limitazioni di visibilità in rete: è una considerazione troppo ottimistica pensare che tutti siano in grado di utilizzare e gestire con efficacia tutti i meccanismi di tutti gli strumenti utilizzati per costruire un'architettura software. Cosa succede se la persona che si occupa di una determinata attività è in ferie/sta male ed è quindi impossibilitata a rispondere a quell'attività? Il risultato è che l'attività rimane in standby o si perde tra le notifiche.
È qui che nasce l'esigenza di un **monitor per i processi CI/CD**, che quindi raggruppi tutte le notifiche e ne permetta una gestione personalizzata. Questa è la mansione principale che *Butterfly* porta a compimento.

1.3 Glossario

Al fine di evitare possibili ambiguità relative al linguaggio utilizzato nei documenti formali, viene fornito il Glossario v3.0.0. In questo documento vengono definiti e descritti tutti i termini con un significato specifico. Per facilitare la comprensione, i termini saranno contrassegnati da una 'G' a pedice.

1.4 Riferimenti informativi

- ***Analisi dei requisiti v3.0.0***;
- **Slide L03 del corso di Ingegneria del Software - Software Architecture Patterns**
  https://www.math.unipd.it/~rcardin/sweb/2019/L03.pdf.

1.5 Riferimenti normativi

- ***Norme di progetto v3.0.0***;
- **Capitolato d'appalto C1 Butterfly: un monitor per i processi CI/CD**
  https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/C1.pdf.

[1] CI/CD

2 Architettura del prodotto

2.1 Descrizione generale

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.

Le principali tipologie di componenti del sistema sono:

- ***producers***: rappresentano i componenti che rimangono in ascolto del *webhook*$_{G}$ proveniente dal servizio loro assegnato (*Redmine, GitLab o SonarQube*), 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.

2.2 Diagramma dei package generale

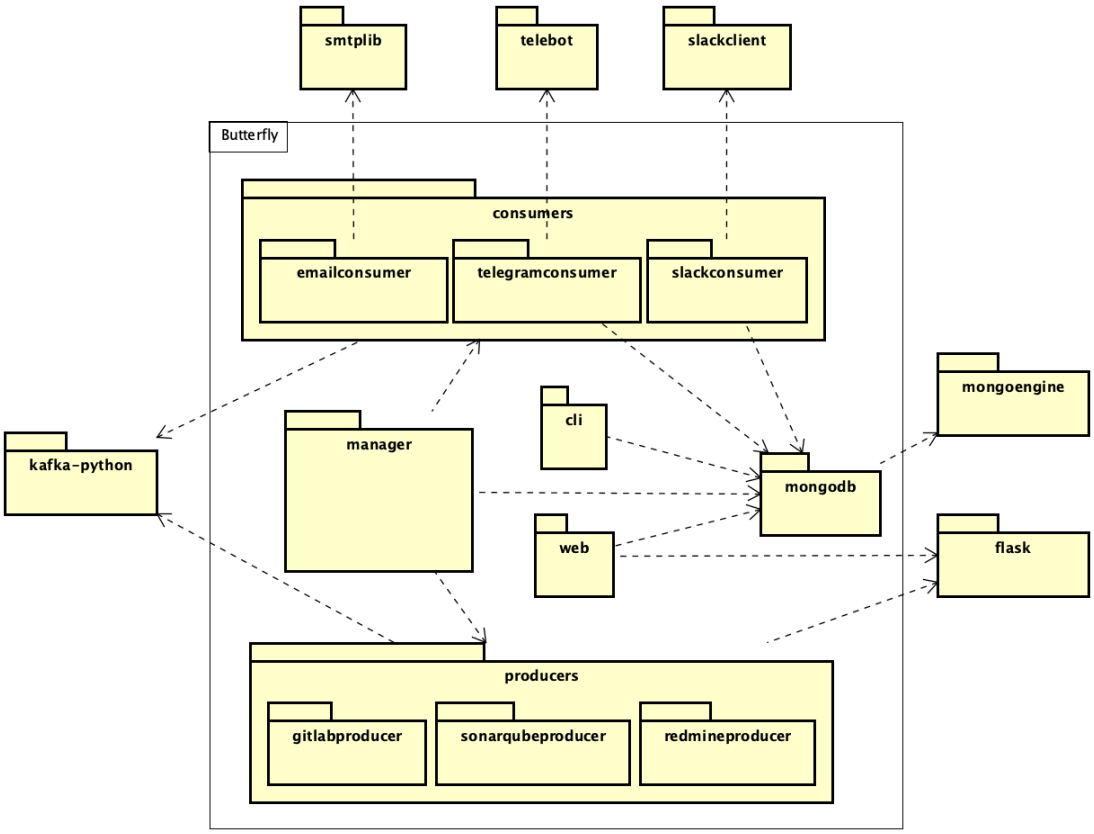

Figura 1: Diagramma dei package dell'architettura generale

Il diagramma di sequenza del funzionamento generale di *Butterfly* è volutamente omesso poiché ritenuto poco efficace: le varie componenti del sistema verranno illustrate separatamente e nel dettaglio nelle prossime sezioni.

2.3 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, Gitlab e Sonarqube), generano messaggi personalizzati secondo i campi d'interesse prestabiliti. I producer implementati nel package *producers* sono tre:

- RedmineProducer;
- GitLabProducer;
- SonarQubeProducer.

2.3.1 Diagramma delle classi

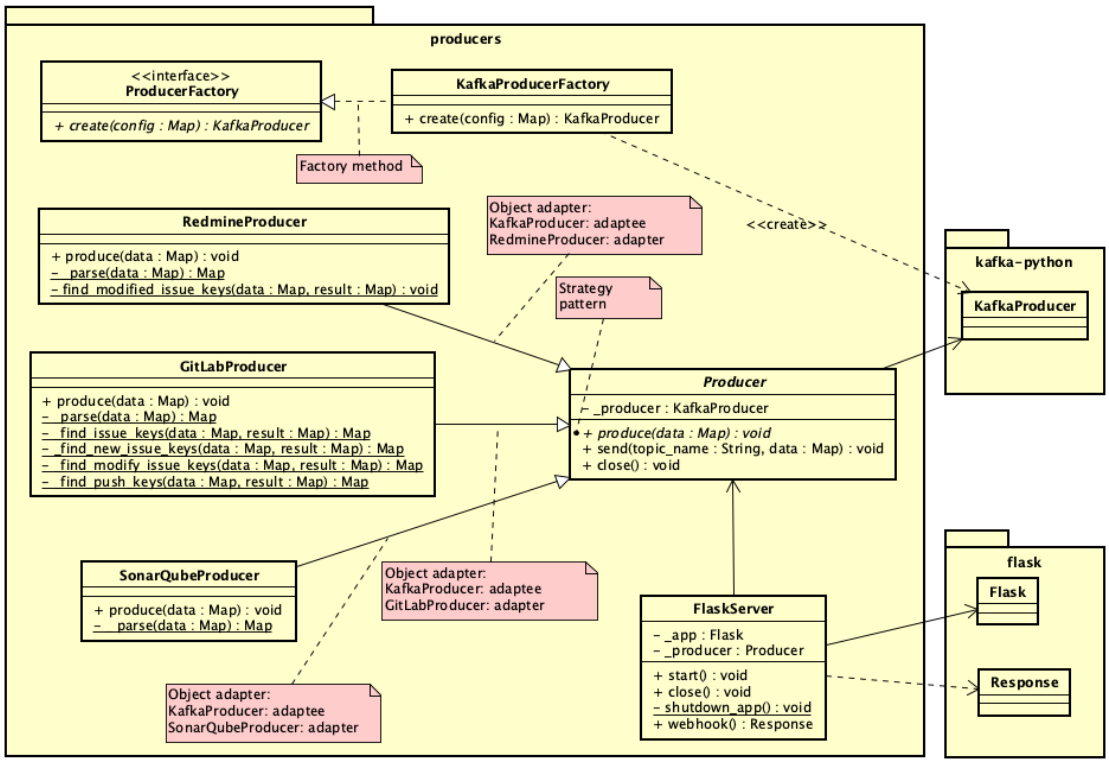

Figura 2: Diagramma delle classi dei componenti *producer*

2.3.2 Design pattern utilizzati

2.3.2.1 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.

2.3.2.2 Strategy

La classe `FlaskServer` ha bisogno di un algoritmo per la generazione e l'invio dei messaggi in linguaggio JSON che è potenzialmente diverso per ogni componente producer implementato. È

stato quindi scelto di incapsulare i differenti algoritmi *produce*, utilizzati per estrapolare le informazioni d'interesse per *Butterfly*, in un'unica classe astratta *Producer*, piuttosto che implementare diversi metodi diversi per ogni algoritmo di generazione e invio dei messaggi. Questa scelta offre i seguenti benefici:

- **organizza tipologie di algoritmi simili in un'unica classe**, agevolando il riuso del codice in contesti analoghi;
- **una valida alternativa al *subclassing*$_G$**, che altrimenti legherebbe significativamente il contesto d'utilizzo con i diversi algoritmi, rendendo il codice poco riusabile, difficile da comprendere e da mantenere, senza contare che non sarebbe possibile scegliere dinamicamente[2] l'algoritmo giusto da usare;
- **esclude gli statement condizionali**: sarà infatti il *dynamic-binding*$_G$ dell'oggetto d'invocazione a stabilire il giusto algoritmo da utilizzare, evitando codice che appesantirebbe ulteriormente la classe.

2.3.2.3 Object adapter

La classe `KafkaProducer`, propria del broker, accetta messaggi codificati in uno specifico formato. È compito della classe `RedmineProducer` assicurarsi che venga generato un messaggio comprensibile per il broker. Si rende necessario, quindi, l'utilizzo di un Object Adapter fra le due classi sopracitate.

N.B.: tale scelta architetturale è stata effettuata anche per `GitLabProducer` e `SonarQubeProducer`, per motivi del tutto analoghi.

2.3.3 Diagrammi di sequenza

I producer hanno lo scopo di ascoltare gli webhook provenienti dai progetti di Redmine, GitLab e SonarQube. Alla ricezione di un webhook viene chiamato un opportuno metodo *webhook()* nel FlaskServer che a sua volta chiamerà l'opportuno metodo *produce()* grazie al design pattern Strategy. Si hanno quindi i seguenti flussi di esecuzione, a seconda del servizio di provenienza del webhook.

[2] a run-time

2.3.3.1 Redmine

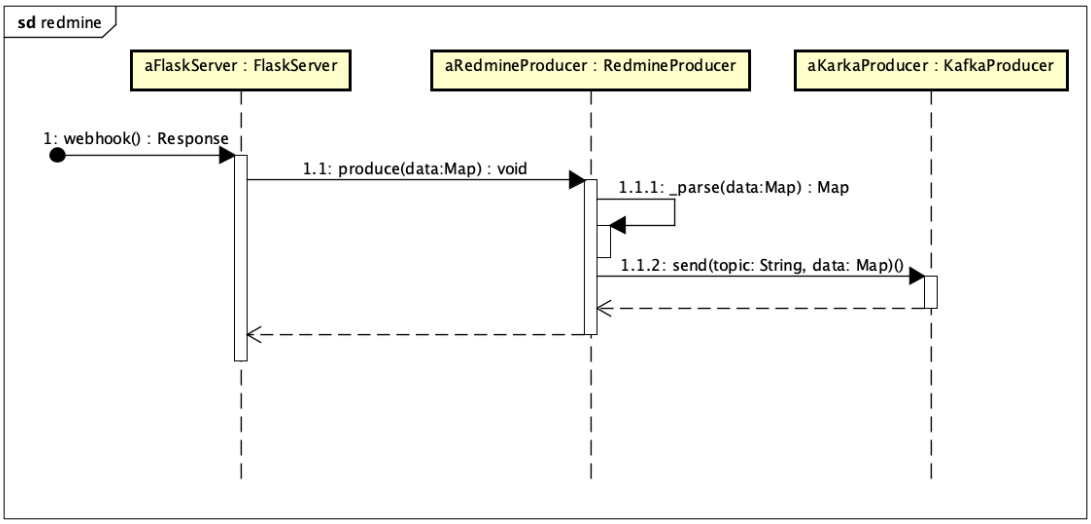

Figura 3: Diagramma di sequenza per la ricezione di un messaggio su Redmine e invio del messaggio su Kafka

2.3.3.2 GitLab

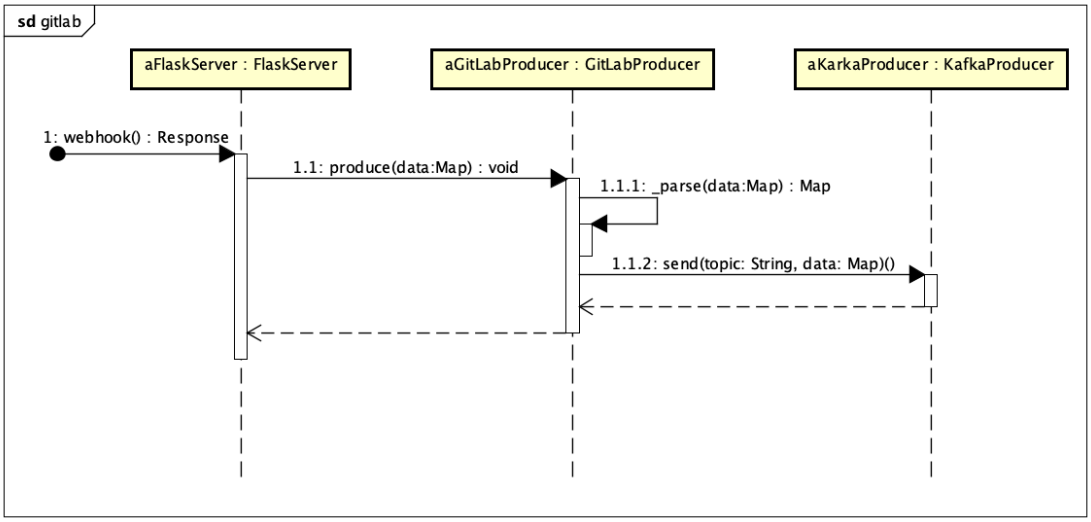

Figura 4: Diagramma di sequenza per la ricezione di un messaggio su Gitlab e invio del messaggio su Kafka

2.3.3.3 SonarQube

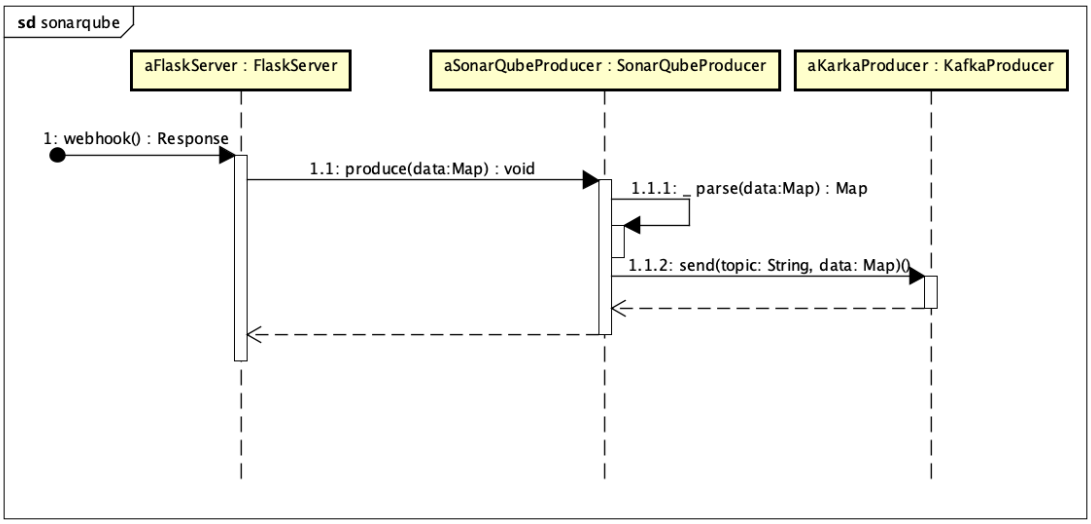

Figura 5: Diagramma di sequenza per la ricezione di un messaggio su Sonarqube e invio del messaggio su Kafka

2.4 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 nelle code relative ai consumer finali; ci sarà inoltre un *Processor* che aggiunge opportune informazioni sui destinatari, ottenute interrogando mongoDB. L'algoritmo per 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.

2.4.1 Diagramma delle classi

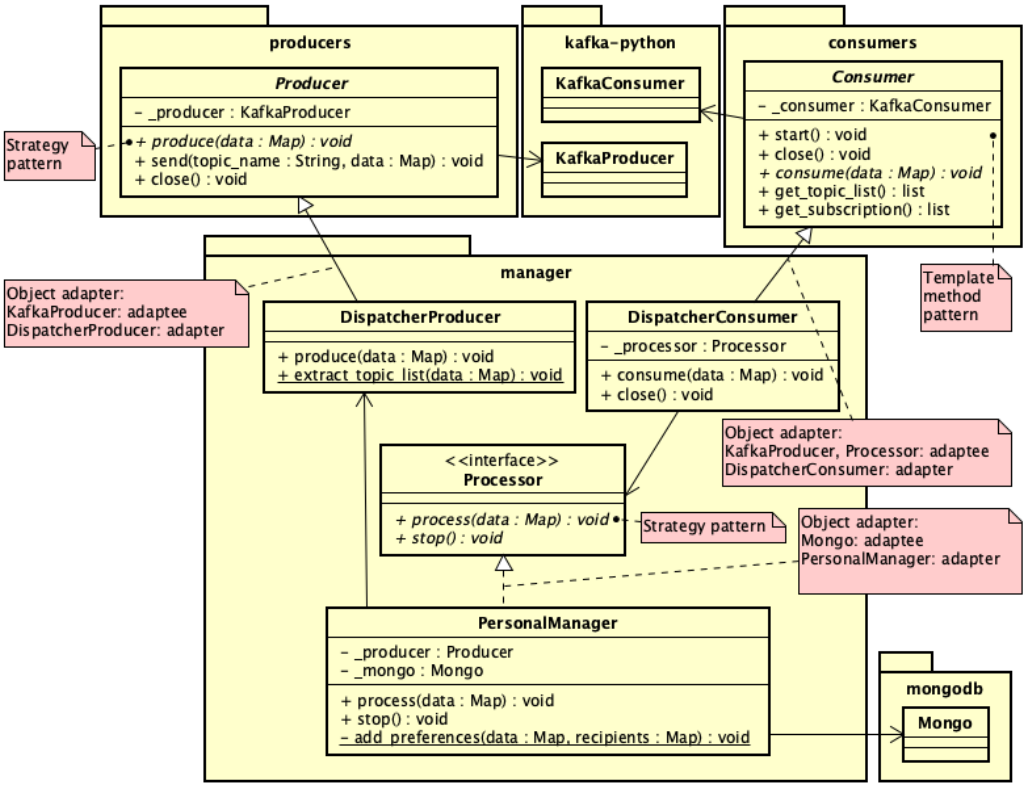

Figura 6: Diagramma delle classi del gestore personale

2.4.2 Design pattern utilizzati

2.4.2.1 Template Method

Il Template Method Pattern, utilizzato dalla classe astratta `Consumer`, è illustrato nel §2.4.2.2.1.

2.4.2.2 Object Adapter

- **fra** `DispatcherProducer` **e** `KafkaProducer`:
  `dispatcherProducer` riceve da `PersonalManager`[3], un messaggio contenente varie

[3] il gestore personale

informazioni, come ad esempio il titolo ed il contenuto della segnalazione. È necessario rendere tale messaggio comprensibile per il broker Kafka: si rende quindi indispensabile l'utilizzo del pattern Object Adapter, in modo da codificare il messaggio ricevuto dai producer per far sì che risulti utilizzabile dal `KafkaProducer`;

- **fra `KafkaConsumer` a `DispatcherConsumer`**:
  il messaggio, ricevuto precedentemente da `KafkaProducer`, si trova attualmente all'interno di Kafka. Il broker provvederà ad arricchire il messaggio con altre informazioni di varia natura. È necessario rendere tale messaggio comprensibile per i futuri consumer: si rende quindi indispensabile l'utilizzo del pattern Object Adapter, in modo da far risultare utilizzabile il messaggio da parte dei consumer. Di questo se ne occuperà la classe `DispatcherConsumer`;

- **fra `PersonalManager` e `Mongo`**:
  `PersonalManager` dovrà utilizzare le funzionalità di Mongo per poter inviare con efficacia ed efficenza il messaggio al destinatario giusto, secondo i canali designati. Per fare questo ha bisogno di comunicare con il database per arricchire la segnalazione: si rende quindi necessario il template Object Adapter.

2.4.2.3 Strategy

- `Producer`:
  Lo Strategy Pattern utilizzato dalla classe astratta `Producer` è illustrato nel §2.4.1.2.2;

- `Processor`:
  La classe astratta `Processor` può definire diversi algoritmi per gestire i messaggi: questa scelta è stata attuata per dare ulteriore modularità al codice. In particolare, il metodo astratto *`process(data: map)`* non fornisce alcun tipo di informazione circa il modo di processare il messaggio proveniente da `DispatcherConsumer`, come ad esempio la scelta del consumer da utilizzare. È stato scelto di implementare quindi lo Strategy Pattern per far sì che `DispatcherConsumer` potesse decidere a run-time come processare l'informazione adeguatamente, in modo che questo sia estendibile anche in caso, ad esempio, di cambiamenti nei criteri di selezione dei destinatari, di formato dei messaggi, e molto altro ancora.

  Questa scelta offre i seguenti benefici:

  - **organizza tipologie di algoritmi simili in un'unica classe**, agevolando il riuso del codice in contesti analoghi;
  - **una valida alternativa al *subclassing*** $_G$, che altrimenti legherebbe significativamente il contesto d'utilizzo con i diversi algoritmi, rendendo il codice poco riusabile, difficile da comprendere e da mantenere, senza contare che non sarebbe possibile scegliere dinamicamente[4] l'algoritmo giusto da usare;
  - **esclude gli statement condizionali**: sarà infatti il *dynamic-binding* $_G$ dell'oggetto d'invocazione a stabilire il giusto algoritmo da utilizzare, evitando codice che appesantirebbe ulteriormente la classe.

[4] a run-time

2.4.3 Diagramma di sequenza

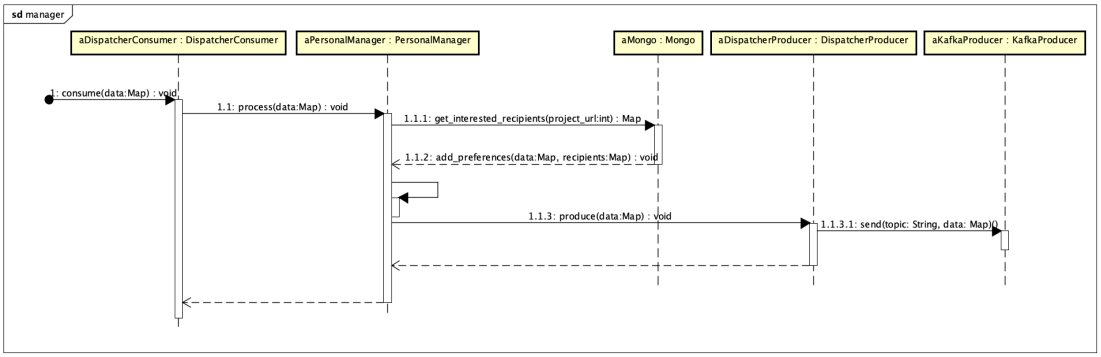

Figura 7: Diagramma di sequenza per l'elaborazione di un messaggio da parte del gestore personale

2.5 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.

2.5.1 Diagramma delle classi

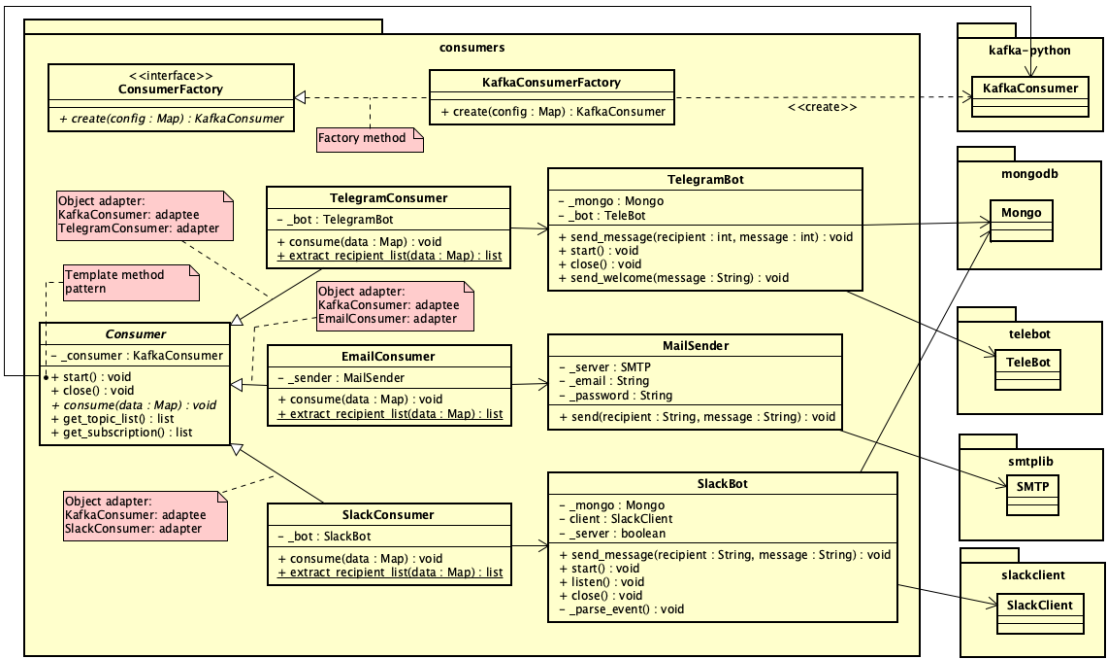

Figura 8: Diagramma delle classi dei componenti *consumer*

2.5.2 Design pattern utilizzati

2.5.2.1 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.

2.5.2.2 Object adapter

La classe `TelegramConsumer` accetta messaggi codificati in uno specifico formato. È suo compito assicurarsi che venga generato un messaggio comprensibile proveniente dalla classe `KafkaConsumer`, propria del broker, per poter inviare efficacemente una segnalazione. Si rende necessario, quindi, l'utilizzo di un Object Adapter fra le due classi sopracitate.

N.B.: tale scelta architetturale è stata effettuata anche per `EmailConsumer` e `SlackConsumer`, per motivi del tutto analoghi.

2.5.2.3 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.

2.5.3 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.

2.5.4 Telegram

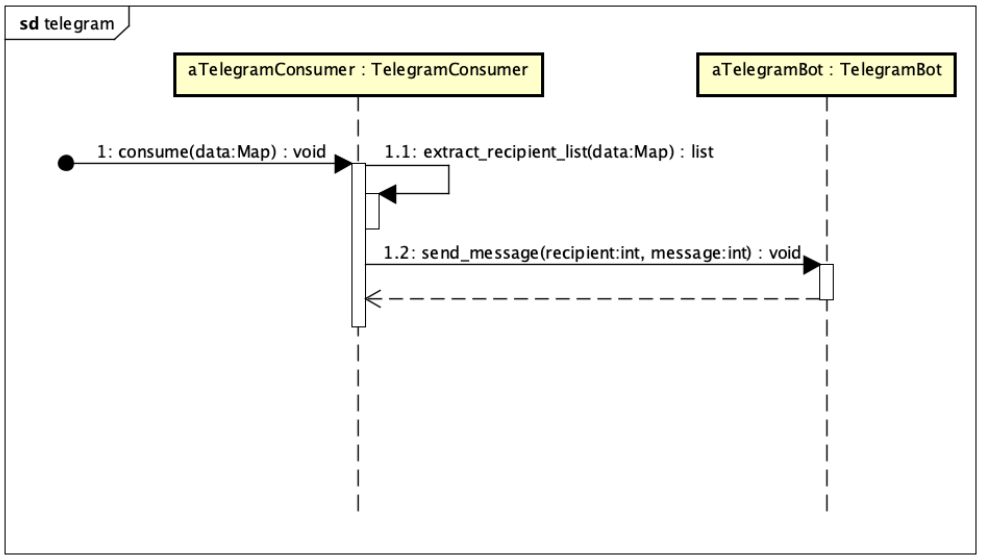

Figura 9: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via telegram

2.5.5 Email

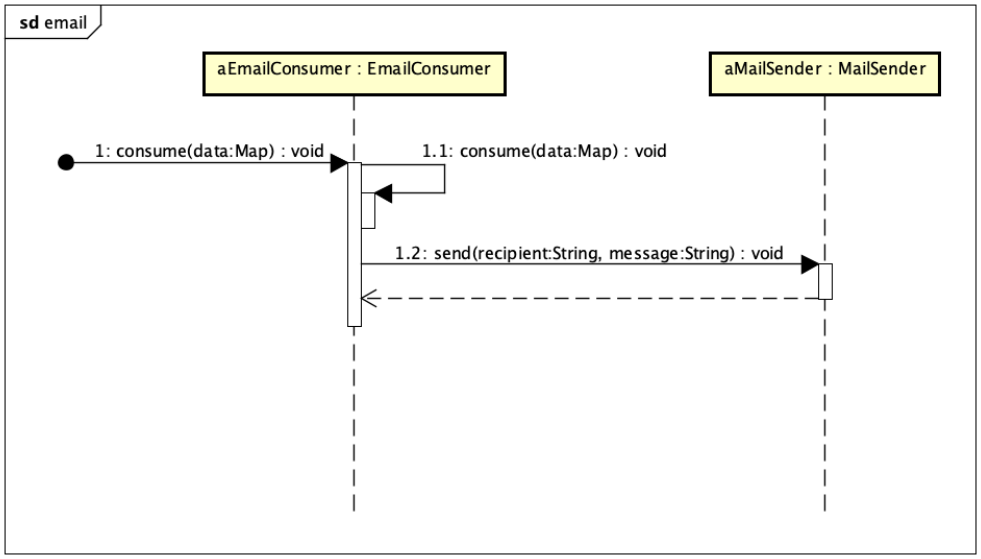

Figura 10: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via email

2.5.6 Slack

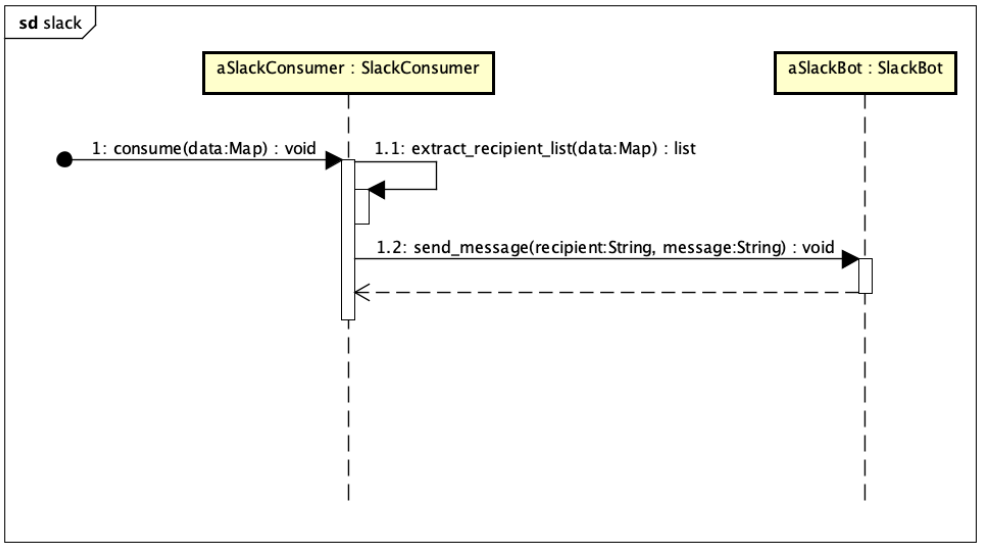

Figura 11: Diagramma di sequenza per la ricezione di un messaggio su Kafka e invio del messaggio via slack

2.5.7 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.
Per quanto riguarda il modello dei dati è stata implementata una classe *Mongo*. Per facilitare la gestione con il package *PyMongo* è stato utilizzato il package *mongoengine* che ne facilita l'utilizzo astraendone i dettagli di basso livello. Naturalmente sia l'interfaccia web che quella a riga di comando permettono inserimento, modifica, visualizzazione e cancellamento dei dati del modello.

2.5.7.1 Diagramma delle classi

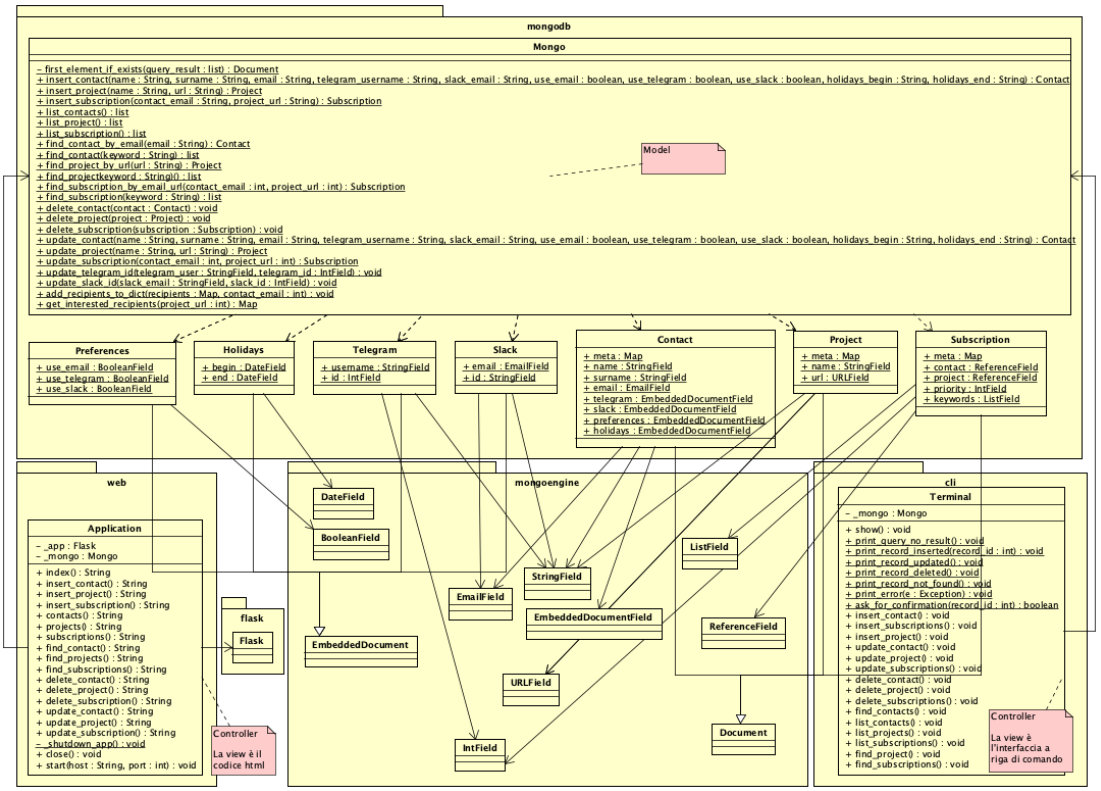

Figura 12: Diagramma delle classi di MongoDB e l'interfaccia

2.5.7.2 Design pattern utilizzati

2.5.7.2.1 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 *Mongo*.
- **view**, che permette all'utente di interagire con i dati contenuti nel model. Nel nostro caso esso è rappresentata dalle pagine *HTML*$_G$ o dalla riga di comando.
- **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*.