Advanced Programming Languages 2023-2024

Weather Event Notifier

Alessandro Genovese 1000002043

Francesco Pennisi 1000055702

Indice

1) Abstract

Lo scopo di questo documento è illustrare le scelte compiute nella progettazione e nello sviluppo di un sistema distribuito che possa essere eseguito su piattaforme architecture-independent.

Il sistema in oggetto è realizzato seguendo un pattern architetturale a microservizi, i quali vengono impacchettati in appositi container facendo uso della tecnologia di containerizzazione Docker.

Il sistema in esame ha l'obiettivo di permettere agli utenti registrati di indicare, per ogni località di interesse, dei parametri meteorologici, ad esempio la massima temperatura o l'eventuale presenza di pioggia. Tali parametri saranno monitorati dal sistema stesso e, in caso di violazione delle condizioni meteo specificate, gli utenti verranno notificati tramite e-mail.

A tale scopo, ad intervalli regolari, anch'essi a discrezione dell'utente, il sistema recupera le informazioni meteorologiche, appoggiandosi al servizio terzo *OpenWeather* (https://openweathermap.org/api) mediante richieste REST API. Tali informazioni vengono filtrate e opportunamente elaborate sulla base delle condizioni sottomesse dagli utenti.

Per permettere all'utente di interagire con il sistema in modo user-friendly è stata realizzata un'applicazione Client dotata di Graphical User Interface (GUI).

2) Architettura del sistema

L'applicazione è costituita da due parti: il lato client e quello server.

- Il client, sviluppato in C# e dotato di GUI, permette all'utente di autenticarsi presso il sistema e di inserire, modificare ed eliminare i parametri meteorologici relativi a una o più località a sua scelta.

L'applicazione server-side, invece, è concepita a microservizi.

- I microservizi User Manager Service (UM) e Weather Management Service (WMS), entrambi sviluppati in Go, sono quelli verso cui vengono instradate le richieste REST effettuate dal client e permettono, rispettivamente, l'autenticazione dell'utente e l'inserimento, la modifica e l'eliminazione delle località e dei parametri di interesse. Entrambi i microservizi sono affiancati a dei database MySQL utilizzati per il salvataggio, rispettivamente, dei dati di autenticazione e delle preferenze degli utenti.

- Invece, i microservizi Worker Service e Notifier Service, sono entrambi sviluppati in Python. Il primo si occupa del controllo periodico dei parametri meteorologici indicati dagli utenti (appoggiandosi al servizio terzo OpenWeather) e il secondo dell'eventuale invio delle e-mail agli utenti stessi in caso di violazione dei valori target da essi specificati. Anch'essi sono affiancati a dei database MySQL, introdotti al solo scopo di salvare il lavoro corrente dei relativi microservizi, in modo da rendere l'applicazione fault-tolerant.

Il client si interfaccia con l'applicazione server-side effettuando tutte le sue richieste ad un API Gateway implementato per mezzo di Nginx, il quale effettua il routing verso i microservizi UM e WMS sulla base dell'URI della richiesta REST sottoposta.

Invece, la comunicazione interna tra i microservizi che compongono l'applicazione server-side sfrutta le tecnologie Apache Kafka e gRPC.

La tecnologia di messaging broker-based Kafka è impiegata per consentire la comunicazione indiretta tra WMS e Worker (per l'indicazione dei parametri da controllare) e tra Worker e Notifier (per indicare gli eventuali parametri violati nelle varie località e comandare l'invio della notifica per gli utenti interessati).

gRPC, invece, è usata per instaurare la comunicazione diretta tra WMS e UM e quella tra Notifier e UM. La prima delle due serve ad autenticare l'utente che ha effettuato una richiesta al WMS, mentre la seconda serve al Notifier per ottenere l'indirizzo e-mail dell'utente da notificare.

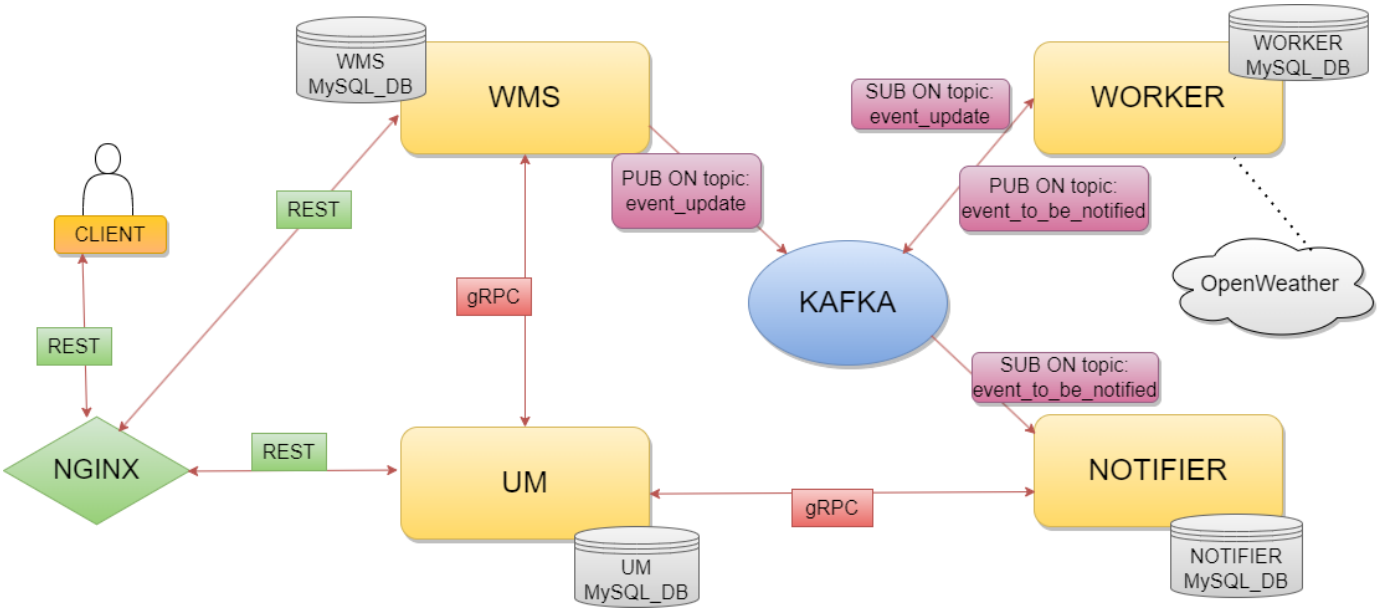

Figura 1 Architettura del sistema

Per maggior chiarezza forniamo di seguito una descrizione più dettagliata dei microservizi server-side:

- **User Manager Service (UM):** consente la registrazione e la successiva autenticazione degli utenti nel sistema. L'autenticazione sfrutta la tecnologia JWT token; l'utente, dunque, attraverso l'utilizzo di un client, effettua il login tramite una richiesta REST API e ottiene un token JWT, il quale viene memorizzato e inserito nell'header *HTTP Authorization* in tutte le successive richieste effettuate dall'utente stesso al sistema.
  Nel momento in cui l'utente volesse inserire/modificare dei parametri meteorologici per una determinata località, egli contatterà il microservizio WMS, inserendo il token di autorizzazione nell'intestazione della richiesta. Quest'ultimo viene immediatamente passato allo User Manager per l'autenticazione dell'utente e, dopo aver validato il token, lo User Manager stesso restituirà al WMS le informazioni necessarie affinché la richiesta dell'utente possa essere soddisfatta con successo.
  L'UM è collegato a un suo database MySQL che include la tabella *Users*. Tale tabella contiene le informazioni relative all'utente: indirizzo e-mail, password e codice identificativo univoco dell'utente.

- **Weather Management Service (WMS):** permette all'utente l'inserimento, la modifica e l'eliminazione dei parametri meteorologici relativi a una o più località a scelta dell'utente stesso.
  Il WMS è collegato a un suo database MySQL che include la tabella *Locations*, in cui vengono memorizzate le informazioni relative alle località di interesse degli utenti, e la tabella *UserConstraints*.
  Quest'ultima contiene le regole specificate dagli utenti, un *trigger_period,* che indica ogni quanti minuti l'utente desidera che il sistema monitori le regole sottoscritte, e un timestamp, che viene aggiornato ogni volta che il WMS prende in considerazione la specifica entry.
  Il WMS, ciclicamente, attraverso l'utilizzo di un timer, controlla la tabella *UserConstraints* e, di volta in volta, prende in considerazione solo le entries per cui è trascorso il *trigger_period* dall'ultimo controllo.
  Il compito del WMS, in sostanza, è quello di costruire dei messaggi Kafka da pubblicare sul topic *event_update*, del quale il microservizio Worker è sottoscrittore.
  Tali messaggi includono le regole degli utenti di cui il Worker dovrà controllare le eventuali violazioni.

- **Worker Service:** ha la responsabilità di contattare il servizio *OpenWeather* per ottenere i dati metereologici attuali e verificare l'eventuale violazione dei parametri specificati dagli utenti.
  Ciò avviene attraverso il seguente ordine procedurale: il Worker preleva dal topic Kafka *event_update* i messaggi pubblicati dal WMS, estrae le regole da controllare, contatta il servizio *OpenWeather* e confronta i risultati ottenuti con quelli ricevuti dal WMS.
  Nel caso in cui si presentino delle violazioni, il Worker le pubblicherà in un messaggio sul topic Kafka *event_to_be_notified*. All'interno del messaggio, ogni parametro violato è associato all'utente, che ne ha richiesto il monitoraggio, tramite il suo codice identificativo univoco.

- **Notifier Service:** il suo compito è quello di prelevare eventuali messaggi Kafka pubblicati sul topic *event_to_be_notified* dal Worker.
  Una volta estratto il messaggio, si occupa di avvisare gli utenti delle avvenute violazioni mediante l'invio di una mail.
  Il Notifier è in grado di recuperare l'e-mail dell'utente contattando lo User Manager tramite il meccanismo di comunicazione gRPC; è possibile identificare univocamente l'utente grazie al suo codice identificativo contenuto nel messaggio Kafka.

- **Kafka:** è un sistema di messaggistica distribuita di tipo publish-subscribe, progettato per essere veloce, scalabile e consentire la persistenza dei messaggi. Tutti gli attori che interagiscono con Kafka possono essere considerati client, publisher e/o subscriber.
  I publisher pubblicano dei messaggi su un topic, mentre i subscriber reperiscono i messaggi pubblicati sui topic a cui si sono sottoscritti.
  La comunicazione è pensata per essere asincrona e indiretta, in modo che non ci sia il bisogno che i processi comunicanti conoscano i rispettivi indirizzi IP. Ogni client comunica in maniera diretta esclusivamente con il broker Kafka.
  Un messaggio è costituito da dati di qualsiasi natura. Spetta ai client definire il formato del messaggio stesso.

- **Nginx:** è un web-server open-source ad alte prestazioni, utilizzato anche come reverse proxy. Garantisce un basso consumo di memoria e un'elevata concorrenza. Anziché creare nuovi processi per ogni richiesta web, Nginx utilizza un approccio asincrono, basato sugli eventi, in cui le richieste vengono gestite in un singolo thread. Nel progetto in esame è

stato utilizzato con la funzione di API gateway, in modo da assicurare la *Location transparency* e la *Partition transparency*.
Nginx riceve le richieste dell'utente tramite richieste REST API e le inoltra all'appropriato microservizio tra User Manager e WMS.

3) Scelte Progettuali

Nel seguente capitolo sono illustrate le scelte progettuali coinvolte nello sviluppo del sistema, suddivise in base ai linguaggi utilizzati.

3.1) C#: Client

Per lo sviluppo del client è stato scelto il linguaggio di programmazione C# perché offre la possibilità di realizzare una interfaccia grafica in modo semplice e inoltre, grazie all'integrazione con il framework .NET, offre un'ampia gamma di librerie e strumenti per lo sviluppo di applicazioni desktop, web e mobile, semplificando il processo di sviluppo.

3.1.1) .NET MAUI

Per il Client scritto in C# si è utilizzato il framework multipiattaforma .NET MAUI con la versione .NET 8. .NET Multi-Platform App UI (.NET MAUI) è un framework multipiattaforma per la creazione di app native per dispositivi mobili e desktop che sfrutta i linguaggi C# e XAML. Usando .NET MAUI, è possibile sviluppare app che possono essere eseguite in Android, iOS, macOS e Windows da una singola codebase condivisa. Il framework permette di tradurre file in formato .xaml in oggetti delle classi MAUI, i quali verranno sostituiti con gli oggetti nativi di ogni piattaforma.

I file XAML sono utilizzati per definire l'interfaccia utente delle applicazioni .NET MAUI. Questi file contengono la descrizione della struttura e dell'aspetto visivo degli elementi dell'interfaccia utente, come layout, controlli, stili e comportamenti.

XAML è un linguaggio dichiarativo, il che significa che gli sviluppatori specificano cosa deve essere visualizzato nell'interfaccia utente, piuttosto che descrivere come ottenerlo tramite codice imperativo. Questo approccio favorisce una migliore separazione tra la logica dell'applicazione e la sua presentazione visiva; infatti, il framework traduce i file in formato .xaml in codice C# per la visualizzazione e la definizione del comportamento della GUI.

L'inizio del programma è rappresentato dal metodo CreateMauiApp all'interno di MauiProgram.cs, il quale gestisce la configurazione e l'avvio dell'applicazione MAUI.

3.1.2) Pattern MVVM

Il Client segue il pattern architetturale Model-View-ViewModel (MVVM).

La View è costituita dai file .xaml uniti ai loro relativi file .xaml.cs. Il suo compito è quello di presentare all'utente un'interfaccia grafica con cui poter interagire.

Il Model, invece, contiene i dati e la logica di business dell'applicazione client. Esso include la gestione dello stato dell'applicazione stessa, il recupero dei dati dal backend e il loro aggiornamento.

Il ViewModel, infine, ha lo scopo di disaccoppiare la View dal Model in modo che entrambi possano evolversi in modo indipendente l'una dall'altro. La sua unica responsabilità è fare da ponte per la trasmissione dei dati dal Model alla View e viceversa.

Il pattern MVVM, proprio grazie al ViewModel, facilita la separazione dei compiti tra presentazione e gestione dei dati all'interno di un'applicazione, consentendo una maggiore manutenibilità, testabilità e scalabilità.

Nello specifico, ciascuna View è associata a un oggetto di una classe del ViewModel specifica per quella View, grazie alla proprietà BindingContext impostata sulla View. Inoltre, tutte le classi del ViewModel implementano l'interfaccia ObservableObject, che richiede la presenza della funzione OnPropertyChanged. Quest'ultima viene richiamata quando una proprietà dell'oggetto varia. Una volta richiamata, essa scatena un evento PropertyChanged, il quale notifica gli elementi grafici che hanno effettuato il binding sulla proprietà in questione, garantendo così la sincronizzazione con la View grafica tramite il pattern Observer.

Nell'ambito dell'applicazione in esame, una proprietà importante nell'interazione tra View e ViewModel è la proprietà *AllRules*, appartenente alla classe *RulesViewModel*. Tale proprietà racchiude una collezione di oggetti della classe *RuleViewModel*, legati tramite binding ai rispettivi elementi grafici nella pagina di visualizzazione delle regole dell'utente, corrispondente alla View *AllRulesPage*. Ognuno di questi elementi grafici, infatti, rappresenta una località a cui l'utente è interessato e per cui ha richiesto il monitoraggio di alcuni parametri meteorologici.

Inoltre, la proprietà *AllRules* è un istanza della classe ObservableCollection, la quale implementa l'interfaccia INotifyCollectionChanged. Questa

interfaccia definisce un evento CollectionChanged, che viene generato ogni volta che la collezione associata viene modificata (elementi aggiunti, rimossi). In tal modo, la View *AllRulesPage* si sincronizza alla modifica effettuata.

Il Client sviluppato presenta due classi Model: Rule.cs e User.cs. In ciascuna di esse vengono definite delle JsonProperties in modo che il modello di dati sia compatibile con il backend in merito alla trasmissione delle request REST API e alla ricezione delle response del server. Le operazioni di serializzazione e deserializzazione Json sono state incluse con il pacchetto Nuget *Newtonsoft.Json*. Il dettaglio dell'interazione tra client e server è descritto nel file README.md associato al progetto.

La classe User contiene i dati relativi all'utente e la logica in merito alla gestione dell'account dell'utente. I metodi di questa classe effettuano delle richieste REST API allo User Manager per l'esecuzione delle operazioni di registrazione, login e cancellazione dell'account.

La classe Rule contiene i dati relativi alle regole metereologiche (associate alle località d'interesse per l'utente) e la logica di recupero e modifica delle regole stesse tramite interazione con il backend.

Ogni richiesta del client verso l'applicazione server-side viene autenticata. L'autenticazione dell'utente è gestita tramite un token JWT, il quale viene restituito dallo User Manager in fase di login e memorizzato in locale su un file testuale. Ciò consente il riutilizzo del token per accessi successivi senza la necessità di rieffettuare il login, poiché sarà il client stesso a recuperare dal file locale il token e fornirlo al backend ad ogni nuova richiesta. Il corretto utilizzo del token JWT da parte del client richiede l'utilizzo della libreria *System.IdentityModel.Tokens.Jwt*, ottenuta grazie al gestore di pacchetti NuGet.

3.1.3) Suddivisione sviluppo software

Lo sviluppo del client è stato diviso nel seguente modo:

Francesco Pennisi: Sviluppo del pattern MVVM per la gestione dell'account.

Alessandro Genovese: Sviluppo del pattern MVVM per la gestione delle regole associate all'utente.

3.2) Go: WMS e UM

Per lo sviluppo dei microservizi WMS e UM è stato scelto il linguaggio di programmazione Go, che consente di coniugare efficienza e semplicità. Quest'ultimo consente di gestire facilmente la concorrenza grazie a un meccanismo più snello dei Thread, ovvero le Go routine. Ciò è particolarmente utile perché entrambi i microservizi devono gestire le richieste da parte dei client. Go, difatti, è un linguaggio fortemente utilizzato nello sviluppo di sistemi distribuiti a microservizi. Inoltre, offre un buon supporto per la comunicazione gRPC, la quale è stata sfruttata nel sistema in questione per la comunicazione interna server-side.

3.2.1) Endpoint REST

Ciascuno dei due microservizi espone degli endpoint, i quali sono spiegati dettagliatamente nel file README.md associato al progetto.

La libreria utilizzata per le comunicazioni REST è *github.com/gorilla/mux*. Quest'ultima permette di ottimizzare le prestazioni, garantendo che la gestione delle route HTTP avvenga in modo efficiente e senza aggiungere un carico significativo al server.

Di seguito sono indicati gli endpoint esposti per le richieste REST da parte del client.

Essi sono suddivisi in base al microservizio a cui afferiscono.

User Manager:

- */register* (POST): per la registrazione degli utenti, i quali forniscono e-mail e password.
- */login* (POST): per l'autenticazione degli utenti; il client riceve come risposta il token JWT.
- */delete_account* (POST): per l'eliminazione dell'account dell'utente e delle relative regole.

WMS:

- */update_rules* (POST): per l'inserimento o l'aggiornamento dei parametri meteorologici, per una data località, scelti dall'utente.
- */show_rules* (GET): per visualizzare le regoli correnti dell'utente loggato.
- */update_rules/delete_user_constraints_by_location* (POST): per eliminare tutte le regole dell'utente relative a una data località.

Il dettaglio dei parametri necessari per ogni richiesta è indicato nel file "README.md".

3.2.2) Database Connector

Ciascun microservizio è collegato a un suo database di tipo MySQL.

Per facilitarne la comunicazione con il database è stata utilizzata la libreria *github.com/go-sql-driver/mysql*. Inoltre, si è deciso di inglobare tutte le funzioni per l'utilizzo del database all'interno di un tipo chiamato DatabaseConnector. Questo contribuisce a rendere il codice più comprensibile e manutenibile nel tempo.

3.2.3) KafkaProducer

Il WMS sfrutta la tecnologia Kafka attraverso l'uso della libreria *github.com/confluentinc/confluent-kafka-go/kafka*. Anche in questo caso si è deciso di utilizzare un tipo, chiamato KafkaProducer, che incorpora le funzioni per la comunicazione con il broker Kafka e la pubblicazione sul topic *event_update*. I motivi che hanno portato a tale scelta sono gli stessi presentati per il DatabaseConnector.

3.2.4) SecretInitializer

Per evitare di esporre la password di accesso al database si è scelto di implementare i Secrets.

Un Secret è un qualsiasi dato che contiene informazioni sensibili e che, dunque, non deve essere trasmesso su una rete o riportato nel codice sorgente dell'applicazione. La gestione dei Secrets è un procedimento che assicura che le informazioni sensibili restino confidenziali.

Nel sistema in esame, i Secrets sono pensati per fare in modo che le informazioni sensibili vengano immesse, in fase di running locale delle immagini, ognuna all'interno di un file testuale posizionato nel file system del container. In tal modo, si permette di recuperare l'informazione ma al tempo stesso si evita di cristallizzare il contenuto sensibile nelle immagini dei vari microservizi pubblicate sul Docker Hub.

Nei microservizi sviluppati in Go, il recupero del segreto viene realizzato da un tipo chiamato SecretInitializer, il quale accede in lettura al file contenente il segreto e imposta quest'ultimo come variabile d'ambiente all'interno del container.

Un aspetto da evidenziare è che per lo sviluppo del SecretInitializer è stato sfruttato il pattern Singleton.

3.2.5) gRPC Communication

Sia nel WMS sia nell'UM, il codice necessario per la comunicazione tramite gRPC è stata isolato all'interno di un file *grpc_communication.go*, contenente tutti i tipi e le funzioni necessarie.

All'interno dei due file è stata importata la libreria *google.golang.org/grpc*.

3.2.6) Suddivisione sviluppo software

Lo sviluppo del WMS e UM è stato diviso nel seguente modo:

Francesco Pennisi: DatabaseConnector (UM e WMS), gRPCCommunication (WMS), ShowRules http handler (WMS), Login http handler (UM), DeleteAccount http handler (UM).

Alessandro Genovese: KafkaProducer (WMS), SecretInitializer (UM e WMS), DeleteRules http handler (WMS), UpdateRules http handler (WMS), Register http handler (UM), gRPCCommunication (UM).

3.3) Python: Worker e Notifier

Per lo sviluppo dei microservizi Worker e Notifier è stato scelto il linguaggio di programmazione Python. Esso è un linguaggio semplice da utilizzare che permette uno sviluppo veloce dell'applicazione; infatti, lo sviluppo del sistema è iniziato proprio dalla scrittura in Python del codice per il Notifier.

Come già chiarito precedentemente, il Worker si occupa di controllare l'eventuale violazione dei parametri meteorologici a cui gli utenti sono interessati e, nel caso avvenissero una o più violazioni per le diverse località di interesse, di pubblicare sul topic Kafka *event_to_be_notified* un messaggio per località su cui ci si sono verificate delle violazioni per qualche utente.

Al contrario, il Notifier estrae dagli eventuali messaggi Kafka le informazioni da notificare ai vari utenti e provvede all'invio delle e-mail.

3.3.1) KafkaProducer e KafkaConsumer

Entrambi i microservizi scritti in Python interagiscono con il broker Kafka. Per questa ragione, è stato necessario importare in essi il package *confluent_kafka*.

A scopi di maggior leggibilità e manutenibilità del codice, sono state sviluppate due classi: la classe KafkaConsumer e la classe KafkaProducer, nelle quali sono implementati tutti i metodi per la comunicazione con Kafka. Il Notifier, che

è solo un consumer Kafka, sfrutterà solo una delle due classi, al contrario del Worker che è sia un producer che un consumer.

3.3.2) DatabaseConnector

Come già illustrato nel paragrafo 3.2.2, anche i microservizi Worker e Notifier sono associati a un database MySQL, nel quale memorizzano il loro lavoro corrente. Anche in questo caso si è scelto di sfruttare una classe DatabaseConnector per l'implementazione delle funzioni di comunicazione con il database. È stata utilizzata la libreria *mysql.connector*.

3.3.3) SecretInitializer

Analogo a quanto discusso nel paragrafo 3.2.4. Anche in questo caso è stato sviluppato secondo il pattern Singleton.

3.3.4) RestQuerier

In aggiunta alle classi precedentemente elencate, nel Worker è stata implementata anche la classe RestQuerier. Tale classe è stata sviluppata al fine di incapsulare tutte le funzioni necessarie per l'esecuzione delle richieste HTTP REST al servizio esterno *OpenWeather* e l'adattamento delle risposte alla logica di business dell'applicazione. Al fine di consentire l'utilizzo del servizio terzo via REST è stata importata la libreria *requests*.

3.3.5) EmailSender

All'interno del Notifier, è stata inclusa anche la classe EmailSender. Essa ha la responsabilità di recuperare, tramite comunicazione gRPC con lo User Manager, l'e-mail dell'utente da avvisare di una avvenuta violazione e successivamente procedere con l'invio dell'e-mail di notifica. Per quest'ultimo scopo è stata utilizzata la libreria *smtplib*.

3.3.6) Suddivisione sviluppo software

Lo sviluppo del Worker e del Notifier è stato diviso nel seguente modo:

Francesco Pennisi: KafkaProducer, EmailSender, DatabaseConnector (Worker e Notifier).

Alessandro Genovese: KafkaConsumer (Worker e Notifier), RestQuerier.