FoxyByte

Specifica Architetturale

FoxyByte - Guida Michelin @ social

foxybyte.swe@gmail.com

Informazioni sul documento

Versione	1.1.0
Redazione	Uderzo Marco
	Fincato Alessandro
	Biasotto Luca
Verifica	Denisa Hida
	Ferrari Gianluca
Responsabile	Bosinceanu Ecaterina
Uso	Esterno
Distribuzione	Vardanega Tullio
	Cardin Riccardo
	Zero12
	FoxyByte

Descrizione

Il presente documento espone l'architettura della *product baseline* sviluppata dal gruppo *FoxyByte*

Registro delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
v1.1.0	2022-09-25	Ferrari Gianluca	*Verificatore*	Verifica generale degli aggiornamenti del documento
v1.0.3	2022-09-24	Marco Uderzo Hida Denisa	*Progettista* *Verificatore*	Ampliamento sezione §2
v1.0.2	2022-09-21	Fincato Alessandro Hida Denisa	*Progettista* *Verificatore*	Aggiornamento sezione §4
v1.0.1	2022-09-12	Fincato Alessandro	*Progettista*	Modifica sezione §4
v1.0.0	2022-09-07	Bosinceanu Ecaterina	*Responsabile*	Accettazione prima versione del documento
v0.2.0	2022-09-07	Ferrari Gianluca	*Verificatore*	Verifica del documento
v0.1.5	2022-09-07	Biasotto Luca	*Progettista*	Aggiunta sezione §2.4
v0.1.4	2022-09-07	Fincato Alessandro	*Progettista*	Aggiunti grafici sezione §4 e aggiornamento sezione §2.1
v0.1.3	2022-09-06	Marco Uderzo	*Progettista*	Aggiornamento Diagrammi UML
v0.1.2	2022-09-04	Fincato Alessandro	*Progettista*	Aggiornamento sezione §4
v0.1.1	2022-09-04	Marco Uderzo	*Progettista*	Ampliamento Sezione §2
v0.1.0	2022-09-02	Fincato Alessandro	*Verificatore*	Verifica documento
v0.0.6	2022-08-31	Marco Uderzo	*Progettista*	Ampliamento Sezione §2
v0.0.5	2022-08-29	Biasotto Luca	*Progettista*	Aggiunta sezione §2.3
v0.0.4	2022-08-09	Bosinceanu Ecaterina	*Responsabile*	Stesura Sezione §3
v0.0.3	2022-08-09	Uderzo Marco	*Progettista*	Stesura Sezione §2
v0.0.2	2022-08-08	Lauriola Pietro	*Amministratore*	Stesura Sezione §1

Versione	Data	Autore	Ruolo	Descrizione
v0.0.1	2022-08-08	Fincato Alessandro	*Responsabile*	Creazione del documento

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Lo scopo della *Specifica Architetturale* è quello di fornire una trattazione completa ed esaustiva dell'architettura del prodotto, le tecnologie che vengono utilizzate e i requisiti richiesti per l'utilizzo dell'applicazione. La struttura del prodotto viene presentata sotto forma di diagrammi delle classi, mentre alcune funzionalità vengono spiegate tramite diagrammi di sequenza. Vengono inoltre spiegati e motivati i *design pattern*$_G$ utilizzati.

1.2 Scopo del prodotto

Il capitolato proposto dall'azienda *Zero12* ha come obbiettivo la creazione di una *guida Michelin*$_G$ che sfrutti le informazioni raccolte dal social network *Instagram*. Lo scopo del prodotto è quindi fornire un'applicativo responsive che permetta agli utenti di ottenere velocemente informazioni, recensioni e opinioni rilasciate degli utenti del social network, sui ristoranti e locali a cui sono interessati.

1.3 Glossario

Al fine di evitare possibili ambiguità nei termini utilizzati nel documento, viene fornito in allegato anche il *Glossario*, dove vengono definiti tutti i termini con un significato specifico. I termini presenti in questo *Glossario* sono identificati dal carattere corsivo e dalla lettera 'G' aggiunta a pedice.

1.4 Maturità del documento

Il presente documento è redatto con un approccio incrementale al fine di poter trattare eventuali nuove questioni in modo rapido ed effiente, sulla base di decisioni concordate tra tutti i membri del gruppo. Può comunque essere considerato definitivo nel senso e nelle scelte che espone alla sua attuale versione, in quanto tratta decisioni il cui cambiamento necessiterebbe una notevole analisi ed eventuali problematiche a monte. Le successive versioni tenderanno quindi ad integrare quanto già definito descrivendolo, eventualmente, in maniera più profonda e precisa.

1.5 Riferimenti

1.5.1 Riferimenti Normativi

- *Norme di Progetto*;
- Capitolato d'appalto C4 - Zero12:
  https://www.math.unipd.it/~tullio/IS-1/2021/Progetto/C4.pdf

1.5.2 Riferimenti Informativi

- Software Engineering - Ian Sommerville (10th Edition);
- Slide dell'insegnamento di Ingegneria del Software:
  https://www.math.unipd.it/~tullio/IS-1/2021/
  - Slide T09 del corso di Ingegneria del Software - Progettazione:
    https://www.math.unipd.it/~tullio/IS-1/2021/Dispense/T09.pdf
  - Slide P02 del corso di Ingegneria del Software - Diagrammi dei casi d'uso:
    https://www.math.unipd.it/~rcardin/swea/2021/Diagrammi%20delle%20Classi_4x4.pdf
  - Slide P03 del corso di Ingegneria del Software - Diagrammi dei casi d'uso:
    https://www.math.unipd.it/~rcardin/swea/2022/Dependency%20Management%20in%20Object-Oriented%20Programming.pdf
  - Slide P05 del corso di Ingegneria del Software - Diagrammi dei casi d'uso:
    https://www.math.unipd.it/~rcardin/swea/2022/Diagrammi%20di%20Sequenza.pdf

2 Architettura del prodotto

2.1 Architettura generale

2.1.1 Schema

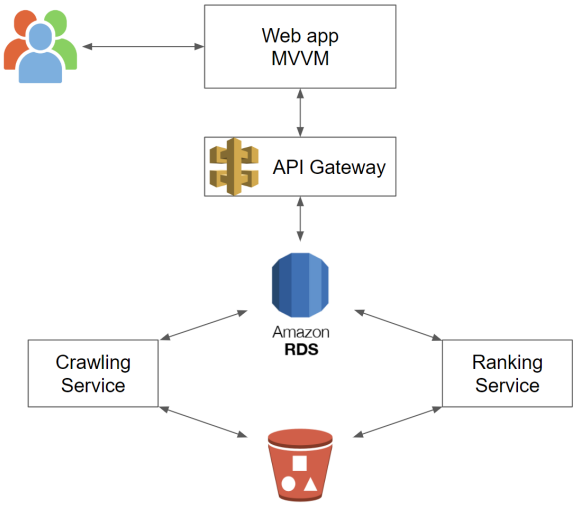


Figura 1: Architettura generale del prodotto

2.1.2 Descrizione

Data la scelta aziendale di Zero12 di lavorare a *microservizi*$_G$, si è deciso di seguire tale approccio anche per il nostro prodotto. Si elencano di seguito i *microservizi*$_G$ principali del *Back-End*$_G$:

- **Crawling Service**: si occupa di trovare, tramite il *crawling*$_G$ di profili instagram, i locali di interesse, sulla base della loro categoria. La pagina della location viene quindi monitorata dal *Crawling Service*$_G$, che recupera le informazioni di ogni post recente della stessa, e le archivia nel *Database*$_G$. Il *crawling*$_G$ dei dati avviene tramite un'*API*$_G$ non ufficiale per Instagram, chiamata *Instagrapi*$_G$. Il *Crawling Service*$_G$ inoltre ricerca il profilo Instagram del locale, al fine di estrapolarne l'immagine di profilo, che principalmente rappresenterà il locale nel *Front-End*$_G$. Si fa notare che, essendo l'intero prodotto basato sull'estrapolazione dei dati dai social media, è considerata come prerequisito per rientrare nei locali analizzati la presenza della location come tag geografico e come profilo ufficiale Instagram che la rappresenti.
- **Ranking Service**: si occupa dell'analisi delle informazioni che il *Crawling Service*$_G$ ha estratto. Ciò avviene tramite i servizi Amazon *Rekognition*$_G$ e *Comprehend*$_G$, oltre ad un algoritmo appositamente creato per valutare i risultati ottenuti da essi, fornendo un punteggio normalizzato che rappresenta la valutazione finale del locale. I dati elaborati dal *Ranking Service*$_G$, insieme a quelli già estrapolati dal *Crawling Service*$_G$, vengono quindi salvati nel database che risiede in Amazon *S3*$_G$.

Per il *Front-End*$_G$ è stato deciso di utilizzare il pattern architetturale *Model-View-ViewModel* (MVVM) per la struttura generale. Questo comunicherà con tutta la parte di Back-End solamente tramite *API Gateway*$_G$. Le motivazioni per cui si è scelto di utilizzare il pattern architetturale MVVM sono:

- La separazione delle responsabilità per un migliore scalabilità, manutenibilità e risoluzione degli errori
- La facilitazione dello sviluppo in parallelo della UI e la logica dietro di esso.
- La libreria *ReactJS*$_G$ che si adatta meglio a tale architettura.

La comunicazione tra i diversi *microservizi*$_G$ avviene attraverso *AWS Lambda*$_G$ (per maggiori informazioni consultare la sezione §2.4), nello specifico:

- **Front end - Back end (RDS)**: il *front end*$_G$, per acquisire le informazioni dal database, utilizza il servizio *API Gateway*$_G$ (per maggiori informazioni consultare la sezione §2.4), che si occupa delle chiamate alle funzioni *lambda*$_G$ corrette; queste hanno il compito di interfacciarsi direttamente con il database, in questo caso *RDS*$_G$, ed eseguire le operazioni necessarie, eventualmente ritornando dei dati al *front end*$_G$, sempre tramite delle *API*$_G$;
- **Crawling Service - Ranking Service**: la comunicazione tra questi due componenti avviene tramite dei trigger, talvolta temporali, e permette il passaggio di informazioni. Inoltre ciascuno di questi microservizi, sempre tramite delle funzioni *Lambda*$_G$ può interfacciarsi indirettamente con il database per eseguire le necessarie operazioni;

2.2 Architettura del Crawling Service

Il *Crawling Service*$_G$ ha tre principali funzionalità:

- **Ricerca degli Utenti**: Al primo avvio, dato un primo profilo Instagram in ingresso, il *crawler*$_G$ cerca nel profilo stesso nuovi profili da seguire, interrogando *Instagrapi*$_G$ circa i profili consigliati da Instagram sulla base dell'utente. Più in generale, per quanto riguarda la funzionalità di estensione del bacino utenti, si può deciderne tramite il file di configurazione del CrawlingService la *policy*$_G$, ovvero se ricercare tramite i profili consigliati sulla base dell'utente, i profili *taggati*$_G$ nei suoi post, o quelli che hanno *taggato*$_G$ l'utente. Si può impostare, inoltre, il numero limite di utenti da aggiungere al bacino.
- **Ricerca dei Locali**: Una volta che il CrawlingService ha sufficienti profili Instagram su cui basarsi, viene eseguita una ricerca di *locations*$_G$ di interesse nei post di ciascun utente, controllando la *location*$_G$ eventualmente *taggata*$_G$. Se viene trovata, e la sua categoria è tra quelle tracciate, la *location*$_G$ viene inserita nel database. Qualora la prima analisi del profilo dell'utente non generasse alcun nuovo inserimento nel database, l'utente stesso verrà rimosso dal database, considerandolo non di interesse. Ciò consente al bacino utenti di non crescere in modo incontrollato o di essere saturo di utenti non utili.
- **Crawling delle Location Pages**: usufruendo dei metodi esposti da *Instagrapi*$_G$, viene eseguito il *crawling*$_G$ dei dati utili degli N post più recenti del locale, dove N è un parametro che può essere impostato tramite il file di configurazione dedicato. Le informazioni estrapolate dai post vengono ulteriormente elaborate, salvandole in oggetti di tipi non primitivi opportunamente definiti, per poi scriverle in un database dedicato.

2.2.1 Controlli Notevoli

- **Temporizzazione**: Il CrawlingService viene avviato tramite un trigger temporizzato utilizzando *AWS Lambda*$_G$, che fa iniziare l'esecuzione della ricerca di nuovi utenti, se abilitata dal file di configurazione, e il *crawling*$_G$ delle *locations*$_G$.
- **Ridondanza di Utenti e Locations**: A monte, prima di decidere di procedere al *tracking*$_G$ di un utente o di una *location*$_G$, viene controllato che il profilo dell'utente non sia già tracciato, o che il locale trovato non sia già presente tra quelli da analizzare, in modo da minimizzare il tempo di esecuzione delle chiamate *Lambda*$_G$, fattore da considerare per il costo economico complessivo di utilizzo del servizio.
- **Ridondanza del Crawling di Locations**: Durante una nuova sessione di crawling, il sistema esegue il crawling esclusivamente dei post per esso nuovi, controllando il codice univoco del post più recente analizzato nella sessione precedente, in modo da minimizzare il tempo di esecuzione delle chiamate *Lambda*$_G$, fattore da considerare per il costo economico complessivo di utilizzo del servizio.

2.2.2 Diagrammi delle Classi

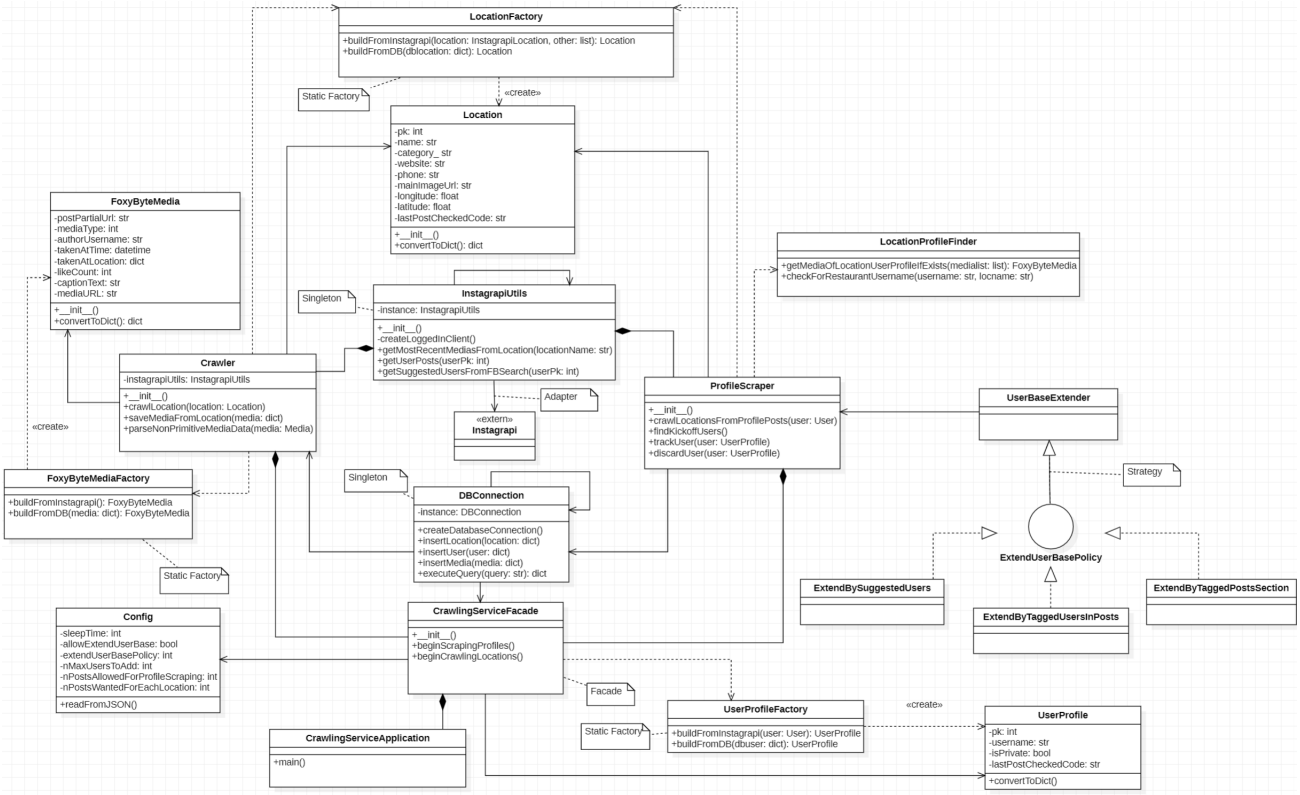


Figura 2: Crawling Service - diagramma delle classi

2.2.3 Descrizione del Diagramma delle Classi

Il *Crawling Service*$_G$ è composto da diversi componenti:

- **CrawlingServiceApplication**: è il punto di ingresso del *CrawlingService*$_G$. Inizializza la **CrawlingServiceFacade** e invoca i suoi principali metodi di avvio del processo di *crawling*$_G$, secondo i parametri di configurazione.
- **Config**: è un *Singleton*$_G$ il cui scopo è caricare i parametri di configurazione da JSON e mantenerli accessibili ai componenti che ne faranno uso. In particolare, i parametri di configurazione sono:
  - **allowExtendUserBase: bool**: abilitare l'estensione del bacino utenti.
  - **extendUserBasePolicyNumber: int**: scelta della policy di estensione (1: *ExtendBySuggestedUsers*, 2:*ExtendByTaggedUsers*, 3:*ExtendByTaggedPostsSection*).
  - **nMaxUsersToAdd: int**: numero massimo di utenti da aggiungere al bacino utenti per sessione.
  - **nPostsAllowedForProfileScraping: int**: numero di post analizzati per profilo durante una sessione.
  - **nPostsWantedForEachLocation: int**: numero di post analizzati per *Location Page* durante una sessione.
  - **instagramUsername: string**: username dell'account di servizio.
  - **instagramPassword: string**: password dell'account di servizio.
  - **locationTags: List[string]**: lista delle categorie di *Locations* di interesse.
- **CrawlingServiceFacade**: implementa i metodi principali appena accennati, ovvero *beginScrapingProfiles()* e *beginCrawlingLocations()* esponendo esclusivamente le funzionalità base, perciò nascondendo la complessità del sistema sottostante.
- **ProfileScraper**: si occupa di ricercare utenti Instagram di interesse e trovare locations di interesse da poter analizzare, per poi salvare entrambi nel database. Controlla inoltre che esista un profilo Instagram ufficiale della *location*, tramite la classe *LocationProfileFinder*, che, dato il nome della *Location* e gli username dei profili Instagram che l'hanno taggata, computa la distanza di *Levenshtein* per determinare se tra tali profili esiste quello della *Location* stessa.
- **LocationProfileFinder**: calcola la distanza di Levenshtein tra due username, come appena spiegato.
- **Crawler**: si occupa di ricercare e aggregare dati di tipo *FoxyByteMedia* dalle *Location Pages* delle locations precedentemente trovate dal *ProfileScraper*.
- **DBConnection**: è un *Singleton*$_G$, implementato tramite *metaclass*, il cui scopo è quello di connettersi al database *RDS*$_G$ e gestire le operazioni di input/output tra esso e il sistema.
- **InstagrapiUtils**: è un *Singleton*$_G$, implementato tramite *metaclass*, e fa da *Adapter* con l'*API*$_G$ esterna Instagrapi esponendo solo i metodi effettivamente utilizzati e quelli che sfruttano una loro combinazione per aggregare dati incompleti.
- **UserBaseExtender**: è uno *Strategy Pattern*$_G$ atto alla decisione della policy di estensione del bacino utenti.
- **UserProfileFactory**: *Static Factory*$_G$ utilizzata per la creazione di oggetti di tipo *UserProfileFactory*, sia da Instagrapi che da Database.
- **LocationFactory**: *Static Factory*$_G$ utilizzata per la creazione di oggetti di tipo *LocationFactory*, sia da Instagrapi che da Database.
- **FoxyByteMediaFactory**: *Static Factory*$_G$ utilizzata per la creazione di oggetti di tipo *FoxyByteMediaFactory*, sia da Instagrapi che da Database.

2.2.4 Diagrammi di Sequenza

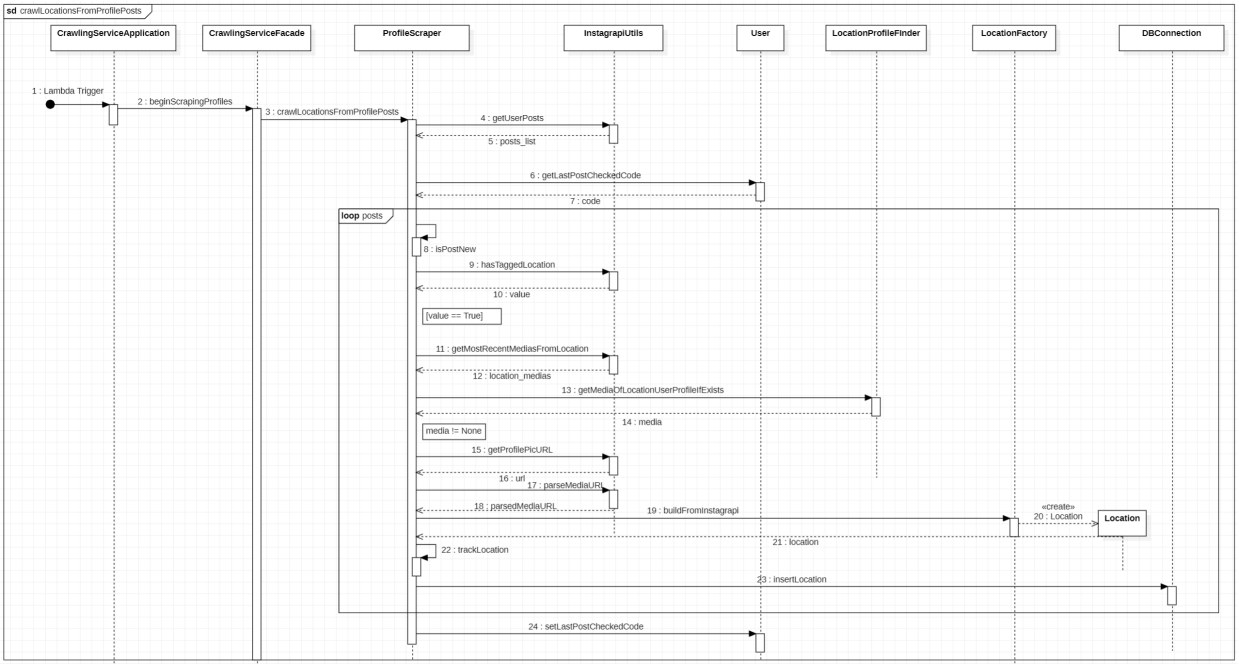


Figura 3: Crawling Service - diagramma di sequenza per il metodo *crawlLocationsFromProfilePosts()* della classe *ProfileScraper*

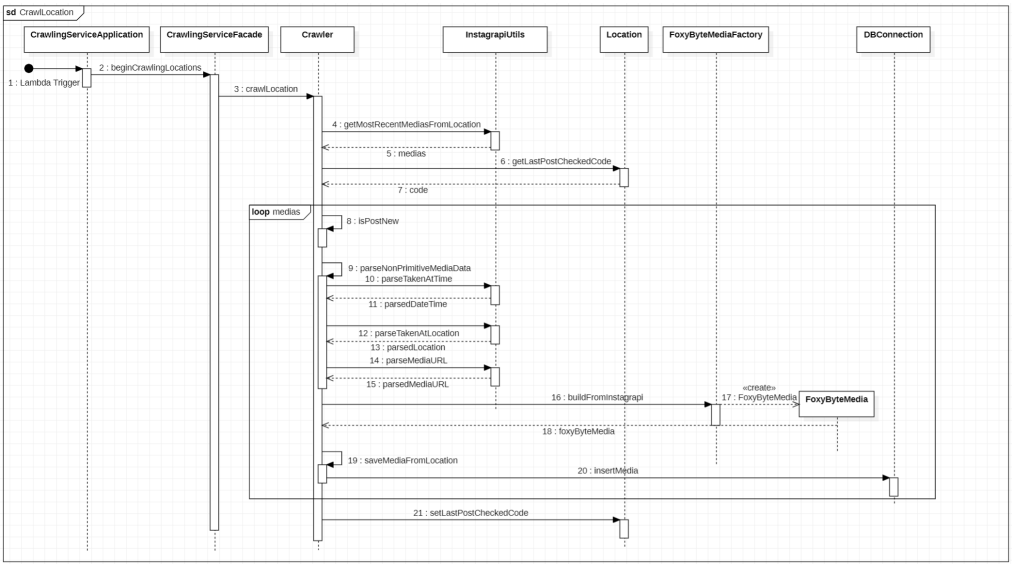


Figura 4: Crawling Service - diagramma di sequenza per il metodo *crawlLocation()* della classe *crawler*

2.2.5 Design Pattern Notevoli

Per lo sviluppo del *Crawling Service*$_G$, sono stati utilizzati i seguenti *Design Pattern*$_G$:

- **Strategy**: Utilizzato nella classe *UserBaseExtender* nella decisione della policy di estensione del bacino di profili Instagram tracciati da *ProfileScraper*.
- **Singleton**: Utilizzato per la creazione di InstagrapiUtils, fornisce al *Crawling Service*$_G$ una singola istanza di client per Instagrapi, che viene condivisa dalle classi *Crawler* e *ProfileScraper*. Lo stesso vale per la classe *DBConnection*.
- **Facade**: Utilizzato per creare *CrawlingServiceFacade*, un'interfaccia più semplice per *Crawler* e *ProfileScraper*, che espone esclusivamente le funzionalità base, quali il caricamento dei parametri di configurazione da JSON e i metodi di avvio dei processi di *crawling*$_G$.
- **Adapter**: Utilizzato per la classe *InstagrapiUtils*, al fine di disaccoppiare *Instagrapi* dal nostro sistema, esponendo solo i metodi effettivamente utilizzati, o metodi che ne utilizzano una combinazione per ottenere dati completi a partire da informazioni parziali.
- **Static Factory**: Per rendere scalabile il *CrawlingService*, si è deciso di passare dall'uso di dizionari seriabilizzabili in file *JSON* alla creazione di oggetti di tipi specifici, garantendo allo stesso tempo una buona separazione dai metodi e dai tipi di *Instagrapi*. Le classi *FoxyByteLocationFactory*, *FoxyByteMediaFactory* e *UserFactory* sono delle *Static Factory* che creano oggetti, rispettivamente, di tipo *FoxyByteLocation*, *FoxyByteMedia* e *User*.

2.3 Architettura del Ranking Service

Il *Ranking Service*$_G$ analizza i dati che il *Crawling Service*$_G$ ha raccolto e caricato su *RDS*$_G$. Da questa analisi verrà creato un database contenente tutte le informazioni che dovranno in seguito essere ricevute dal *Front-End*$_G$ dell'applicazione (vedi diagramma delle classi in seguito). Quindi, lo scopo del servizio è quello di integrare i dati raccolti dal *Crawling Service*$_G$ con delle informazioni aggiuntive, in modo da assegnare un rank a ciascun luogo di interesse.
Queste informazioni aggiuntive, da ricavate tramite quelle già presenti nei dati raccolti dal *Crawling Service*$_G$ sono:

- **Indirizzo del luogo di interesse**: i dati raccolti dal *crawler*$_G$ forniscono esclusivamente la latitudine e la longitudine del luogo di interesse, informazioni precise ma poco leggibili dal punto di vista degli utenti finali. Per ricavarne la posizione si fa affidamento ad un modulo di python, *geopy*$_G$, che consente di trasformare tali coordinate in un indirizzo preciso;
- **Punteggio del luogo di interesse**: questo deve essere basato sui giudizi che i clienti hanno espresso sui social media e rispecchiare la loro esperienza. Per fare ciò si utilizzano due servizi di *AWS*$_G$:
  - *Comprehend*$_G$: prende come input un testo scritto e, utilizzando un modello di Machine Learning, deduce il linguaggio in cui è stato scritto e cerca di comprenderne il sentimento espresso;
  - *Rekognition*$_G$: prende come input un'immagine o un video e, utilizzando un modello di Machine Learning, è in grado di riconoscere il contenuto delle immagini date in input, sotto forma di oggetti, luoghi oppure azioni.

In pratica, a partire dai dati letti, il programma crea delle istanze della classe *Restaurant*, che rappresenta uno dei luoghi di interesse e che vengono raggruppate in una lista. Ogni oggetto di tipo *Restaurant* contiene una lista di oggetti di tipo *Media*, ciascuno dei quali racchiude i dati di un singolo post relativo al *Restaurant* di appartenenza.

2.3.1 Diagramma delle Classi

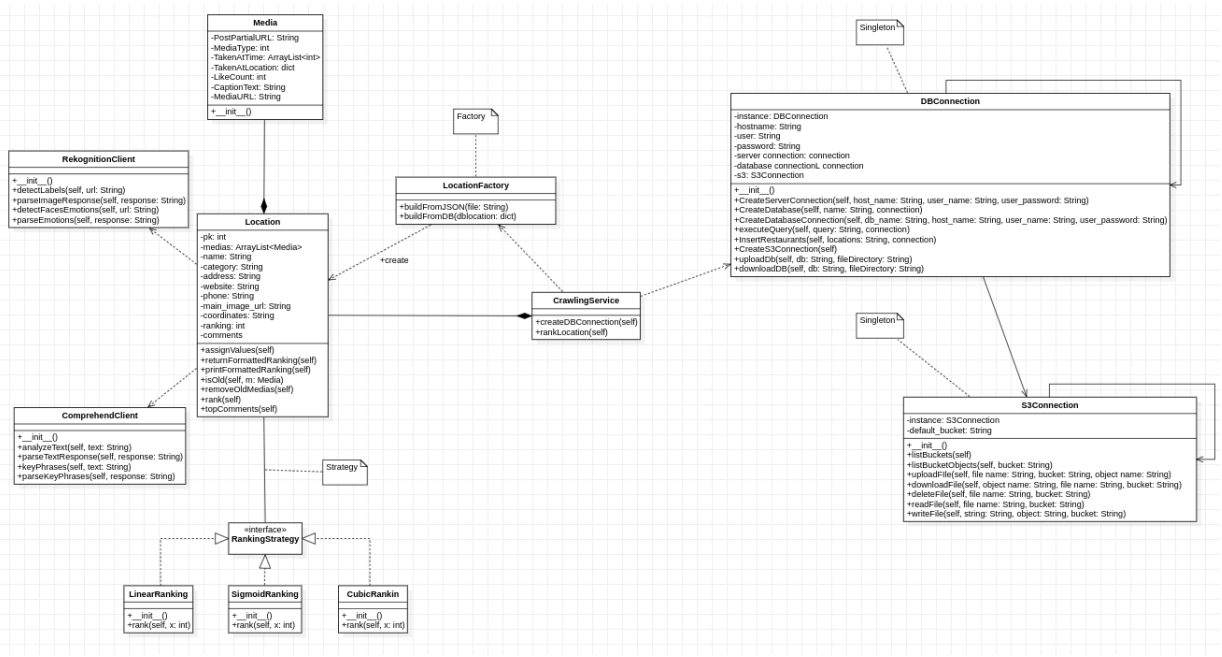


Figura 5: Ranking Service - diagramma delle classi

2.3.2 Diagramma di Sequenza

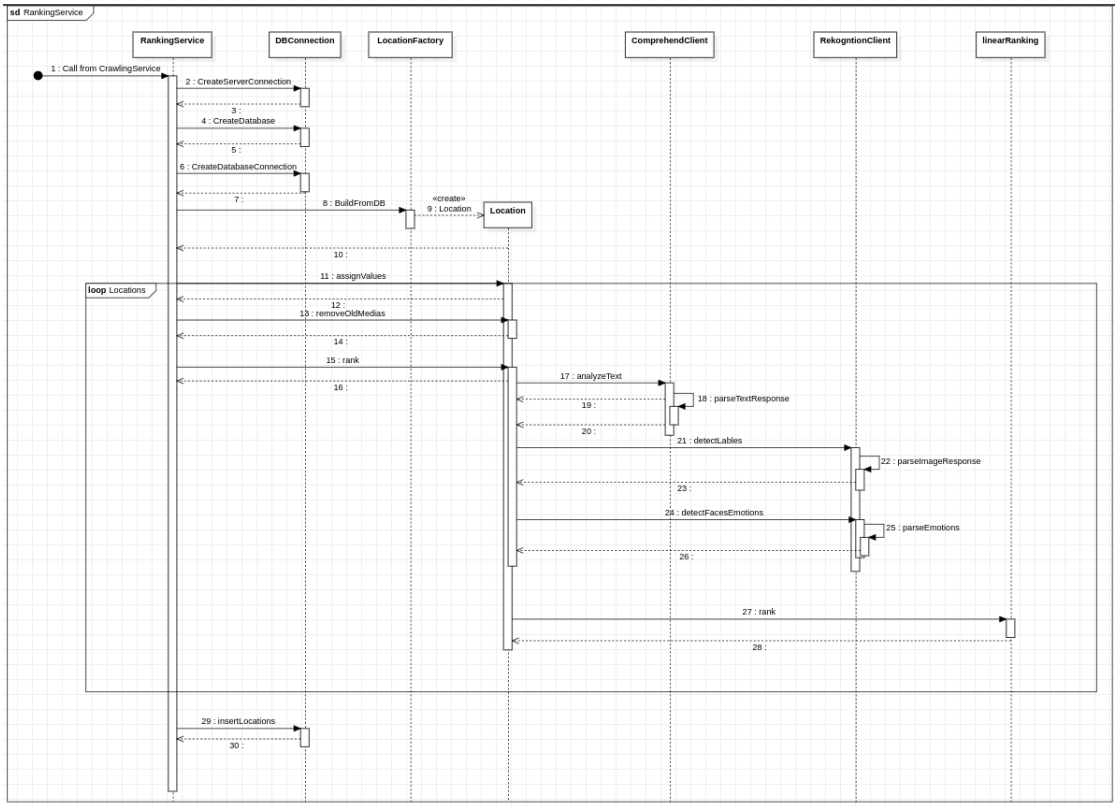


Figura 6: Ranking Service - diagramma di sequenza x

2.3.3 Design Pattern Notevoli

Per lo sviluppo del *Ranking Service*$_G$, sono stati utilizzati i seguenti *Design Pattern*$_G$:

- **Strategy**: Utilizzato nella classe *Location* nella decisione della policy di trasformazione dal ranking nel range da un range di $[-1, 1]$ a un range di $[0, 10]$.
- **Singleton**: Utilizzato per la creazione di oggetti che rappresentano una connesione con il database e con il servizion Amazon *S3*.
- **Factory**: per creare le classi definite da utenti che contengono i dati suoi luoghi di interesse, in maniera compatibile con quanto fatto nel *CrawlingService*. Le classi *FoxyByteLocationFactory*, *FoxyByteMediaFactory* e *UserFactory* sono delle *Static Factory* che creano oggetti, rispettivamente, di tipo *FoxyByteLocation*, *FoxyByteMedia* e *User*.

2.4 Architettura Serverless

Su richiesta del proponente, il prodotto finale è stato sviluppato utilizzando un'architettura *serverless*$_G$, ossia basata sul concetto di non avere un server di proprietà costantemente attivo. Si fa invece affidamento su un sistema di servizi in cloud forniti da una parte terza, in questo caso *Amazon Web Services*$_G$, da qui in poi *AWS*$_G$

In particolare i servizi di *AWS*$_G$ utilizzati sono:

- **AWS Simple Storage Service (S3)** Si tratta di uno spazio in cloud per il salvataggio e la gestione di file. Questi possono essere organizzati in contenitori detti *bucket*, i quali contengono

oggetti (file o cartelle). Viene utilizzato per salvare alcuni file di configurazione utilizzati dal programma e per caricare dei file temporanei utilizzati da *AWS Rekognition*$_G$;

- **AWS Relational Database Service (RDS)** È un database relazione gestito da Amazon, per il quale si può scegliere l'engine (*MariaDB*$_G$ nel nostro caso) e il livello di accesso (pubblico o privato). Viene usato come database sia dal *Front-End*$_G$ che dal *Back-End*$_G$, nei modi sopra descritti;

- **AWS Lambda** Le funzioni *Lambda*$_G$ di *AWS*$_G$ consentono di eseguire codice utilizzando le capacità di cloud computing fornite da Amazon. È possibile impostare la quantità di memoria massima richiesta da ciascuna funzione così come un tempo massimo di esecuzione. Per salvare su una memoria persistente le informazioni necessarie tra un'esecuzione e quella successiva si fa affidamento ai sopracitati *S3* e *RDS*. Le diverse funzioni possono chiamarsi a vicenda, o essere eseguite in seguito a uno specifico trigger che è possibile impostare. I vantaggi di questa tecnologia rispetto all'utilizzo di un server di proprietà sono la delega della gestione infrastrutturale a una società terza e una maggiore efficienza dal punto di vista dei costi, in quanto non si paga per tenere un server costantemente attivo bensì soltanto il tempo in cui il codice entra concretamente in esecuzione;

- **AWS API Gateway** Fornisce un'interfaccia di collegamento tra il *Front-End*$_G$ della *Webapp*$_G$ e i dati raccolti ed analizzati nel *Back-end*$_G$. Quello che fa il servizio nello specifico è collegare le richieste della *WebApp*$_G$ con le funzioni che si connettono direttamente con il database, *RDS*$_G$, e che eseguiranno le azioni richieste.

3 Architettura del Front-end

Il *Front-End*$_G$ è stato realizzato utilizzando la libreria *React*$_G$ integrata al pattern architetturale *Model-View-ViewModel*. Tale pattern è stato scelto in quanto React ha un flusso dati monodirezionale, noto come *one-way-data-binding*$_G$, il che limita l'implementazione di un MVC.

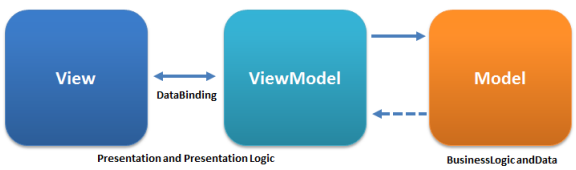
Model-View-ViewModel (MVVM) è un modello di progettazione strutturale che separa gli oggetti in tre gruppi distinti:

- Le classi *Models* contengono i dati dell'applicazione. Corrispondono alle classi User, Place e Favurites.

- Le classi *Views* hanno come responsabilità la definizione della struttura, il layout e l'aspetto.

- Le classi *ViewModel* implementano le funzionalità che manipolano i dati presi dal Model per adattarli alle esigenze della View.

La separazione delle responsabilità è un criterio molto importante per la manutenzione di un prodotto software in futuro. E non solo, anche la risoluzione degli errori, la flessibilità e la scalabilità diventano più semplici quando si adotta una tale architettura.
La *View* chiama i metodi del ViewModel sull'azione dell'utente. Quando un componente si renderizza, si usa il React hook *useEffect()* per chiamare i dati e i metodi del ViewModel necessari per essere mostrati al utente. Il ViewModel chiama il Model per ottenere i dati ed il Model, tramite le chiamate alle *API*$_G$, recupera i dati e li ritorna alla ViewModel. Le funzionalità del ViewModel che chiamano le funzionalità del Model sono tendenzialmente asincrone(async) per non bloccare la View, la quale osserva e si aggiorna quando i dati cambiano. Le componenti della View sono degli 'observer' che osservano tramite i React hooks i cambiamenti che ViewModel notificherà di aver subito.



3.1 Diagramma delle classi

La classe View è la classe che fa il rendering dei principali componenti. Alcuni dei componenti renderano altri componenti neccessari per loro, come sono le componenti *Home* e *UserPage* che utilizzano la stessa componente *PlaceCard* adottandola alle loro esigenze.

La rappresentazione delle classi da un diagramma UML:

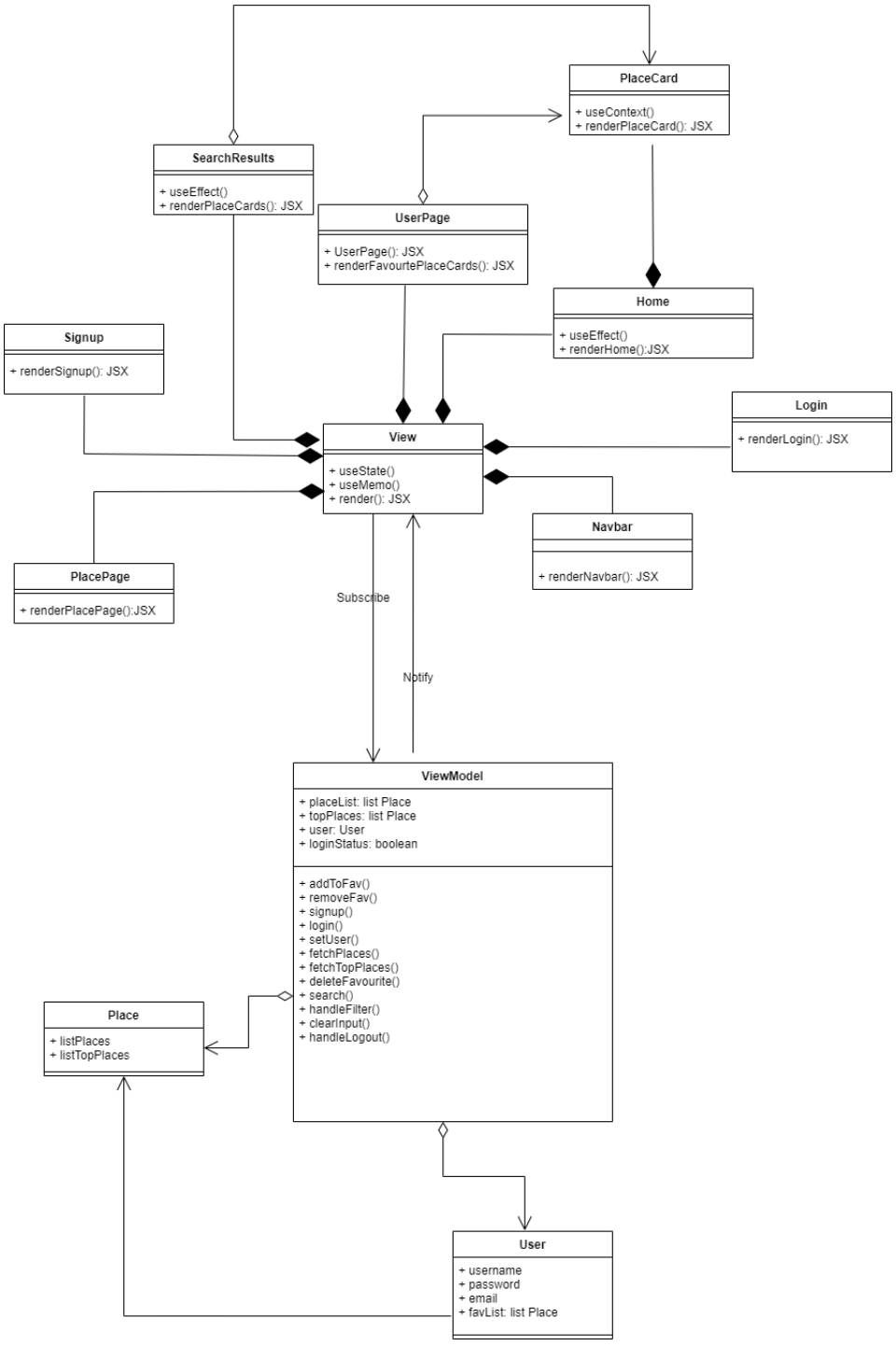


Figura 7: Front-end - diagramma delle classi

4 Requisiti Soddisfatti

4.1 Requisiti funzionali

Tabella 1: Requisiti funzionali

Codice	Classificazione	Avanzamento	Codice	Classificazione	Avanzamento
ROF1	Obbligatorio	Soddisfatto	ROF1.1	Obbligatorio	Soddisfatto
ROF1.3	Obbligatorio	Soddisfatto	ROF2	Obbligatorio	Soddisfatto
ROF3	Obbligatorio	Soddisfatto	ROF4	Obbligatorio	Soddisfatto
ROF4.1	Obbligatorio	Soddisfatto	ROF4.2	Obbligatorio	Soddisfatto
ROF4.3	Obbligatorio	Soddisfatto	ROF5	Obbligatorio	Soddisfatto
ROF5.1	Obbligatorio	Soddisfatto	ROF6	Obbligatorio	Soddisfatto
ROF7	Obbligatorio	Soddisfatto	ROF8	Obbligatorio	Soddisfatto
ROF9	Obbligatorio	Soddisfatto	ROF10	Obbligatorio	Soddisfatto
ROF10.1	Obbligatorio	Non Soddisfatto	ROF10.2	Obbligatorio	Soddisfatto
ROF10.3	Obbligatorio	Soddisfatto	ROF10.4	Obbligatorio	Non Soddisfatto
ROF11	Obbligatorio	Soddisfatto	ROF11.1	Obbligatorio	Soddisfatto
ROF11.3	Obbligatorio	Soddisfatto	ROF11.4	Obbligatorio	Non Soddisfatto
ROF11.5	Obbligatorio	Soddisfatto	ROF12	Obbligatorio	Soddisfatto
RDF12.1	Obbligatorio	Soddisfatto	ROF13	Obbligatorio	Soddisfatto
RDF13.1	Obbligatorio	Soddisfatto	ROF14	Obbligatorio	Soddisfatto
ROF15	Obbligatorio	Soddisfatto	RDF1.2	Desiderabile	Soddisfatto
RDF2	Desiderabile	Soddisfatto	RDF2.1	Desiderabile	Soddisfatto
RDF5.2	Desiderabile	Soddisfatto	RDF11.2	Desiderabile	Non Soddisfatto
RDF11.6	Desiderabile	Non Soddisfatto			

4.2 Requisiti di qualità

Tabella 2: Requisiti di qualità

Codice	Classificazione	Avanzamento
RQ1	Obbligatorio	Soddisfatto
RQ2	Obbligatorio	Soddisfatto
RQ3	Desiderabile	Soddisfatto
RQ4	Desiderabile	Soddisfatto
RQ5	Obbligatorio	Soddisfatto

4.3 Requisiti di vincolo

Tabella 3: Requisiti di vincolo

Codice	Classificazione	Avanzamento
RV1	Obbligatorio	Soddisfatto
RV2	Obbligatorio	Soddisfatto
RV3	Obbligatorio	Soddisfatto
RV4	Obbligatorio	Soddisfatto
RV5	Obbligatorio	Soddisfatto
RV6	Obbligatorio	Soddisfatto
RV7	Obbligatorio	Soddisfatto
RV8	Obbligatorio	Soddisfatto
RV9	Obbligatorio	Soddisfatto
RV10	Obbligatorio	Soddisfatto

4.4 Grafici relativi al soddisfacimento dei requisiti

4.4.1 Requisiti soddisfati

Il seguente grafico a torta mostra la percentuale di requisiti del nostro prodotto che sono stati soddisfatti, ossia 47, su un totale di 52 (i requisiti rimasti da soddisfare sono quindi 6).

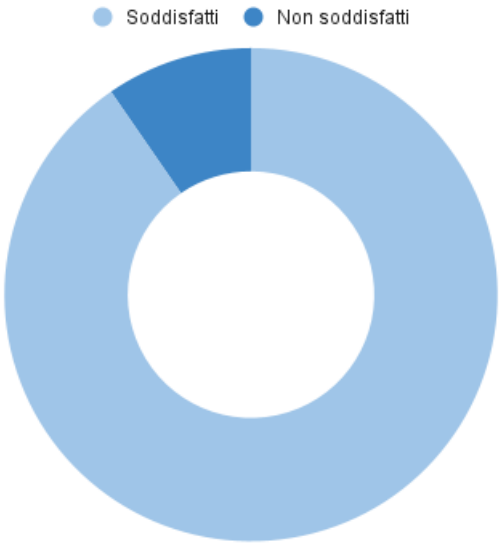


Figura 8: Requisiti soddisfatti

4.4.2 Requisiti obbligatori soddisfatti

Sono stati definiti 44 requisiti obbligatori da soddisfare, nello specifico sono: 31 requisiti funzionali, 3 requisiti di qualità e 10 requisiti di vincolo. Allo stato attuale ne sono stati soddisfatti 41 (28 funzionali, 2 di qualità e 10 di vincolo), ne rimangono ancora 4.

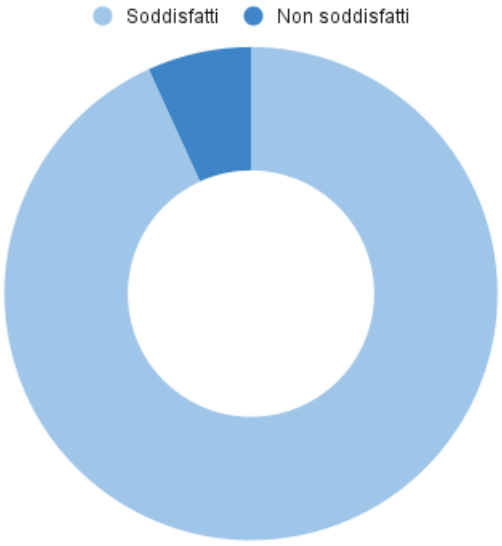


Figura 9: Requisiti obbligatori soddisfatti