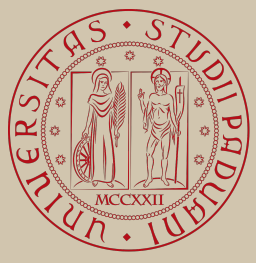
Specifica Tecnica

2025-02-27

V0.0.1

sweetenteam@gmail.com

https://sweetenteam.github.io



Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin AzzurroDigitale
Redattori	Orlando Ferazzani
Verificatori	Mouad Mahdi

Registro delle modifiche

Versione	Data	Autori	Verificatori	Dettaglio
0.0.1	2025-02-27	Orlando Ferazzani	Mouad Mahdi	Prima stesura documento

Indice

Lista della immagini

1) Introduzione

1.1) Scopo del documento

Il presente documento ha lo scopo di fungere da risorsa esaustiva per la spiegazione e conseguente comprensione degli aspetti tecnici del progetto azzurrodigitale:

Figura 1: Logo BuddyBot

La sua finalità primaria è quella di fornire una panoramica dettagliata e approfondita delle scelte progettuali, architetturali e tecnologiche del sistema sviluppato. In particolare, si intende fornire un'analisi profonda estesa al livello di progettazione più basso, includendo spiegazione, definizione e motivazione delle scelte effettuate, e dei *design pattern*$_G$ adottati.

Il documento ha quindi scopi molteplici:

- Motivare le scelte progettuali e di sviluppo adottate;
- Fungere da guida per il processo di sviluppo e manutenzione del sistema;
- Fornire una vista panoramica e monitorare la *Code Coverage*$_G$ dei requisiti del progetto identificati nel documento `Analisi dei Requisiti` (visionabile qui);

L'adeguatezza e la completezza del documento (e del progetto) sono in costante evoluzione e miglioramento in base ai *feedback*$_G$ ricevuti e sulla base dell'aggiornamento dei requisiti.

1.2) Scopo del prodotto

L'obiettivo del progetto è la realizzazione di un *chatbot*$_G$ sotto forma di *Web App*$_G$ atto a fornire un supporto al team di azzurrodigitale: nella gestione delle attività di un progetto in corso di sviluppo. Nella fattispecie, il chatbot utilizza delle *API*$_G$ e un modello di *LLM*$_G$ per, rispettivamente, reperire informazioni da sistemi esterni utilizzati dall'azienda (più specificatamente, Jira, GitHub e Confluence) e elaborare una risposta. Questa risposta può contenere del semplice testo, un link o un *code block*$_G$. Il chatbot ha una singola sessione per ogni utente, e può essere utilizzato da più utenti contemporaneamente.

Il team è confidente che questo genere di prodotto migliorerà il workflow del team di azzurrodigitale:, riducendo i tempi di risposta e migliorando la qualità del lavoro svolto.

1.3) Miglioramenti e maturità

Questo documento è redatto con approccio incrementale e modificato nel tempo per riflettere l'andamento del progetto e le decisioni prese. In particolare, il documento è soggetto a modifiche in base ai feedback ricevuti e all'evoluzione dei requisiti del progetto. Per questo motivo, il documento non è considerabile definitivo, esaustivo e completo fino al raggiungimento di una versione stabile dello stesso (`1.0.0` o superiore).

1.4) Glossario

Per evitare ambiguità e incomprensione riguardanti la terminologia tecnica utilizzata nel documento, viene redatto e adottato un Glossario contenente le definizioni dei termini tecnici utilizzati. Il Glossario è consultabile qui e i termini presenti nel documento sono evidenziati con *questo stile*$_G$.

1.5) Riferimenti

1.5.1) Riferimenti normativi

- Presentazione pdf del capitolato C9: C9p.pdf (*versione disponibile al 2025-03-20*)
- Norme di Progetto: Norme_di_Progetto_v1.0.0.pdf
- Piano di Qualifica: Piano_di_Qualifica_v1.0.0.pdf

1.5.2) Riferimenti informativi

- Analisi dei Requisiti: Analisi_dei_Requisiti_v1.1.0.pdf
- Glossario: Glossario
- I diagrammi dei casi d'uso: Use case
- Progettazione: I pattern architetturali Software Architecture Patterns
- Verifica e validazione: analisi statica (T10): analisi statica
- Verifica e validazione: analisi dinamica aka testing (T11): analisi dinamica
- Programmazione: SOLID programming principles

1.5.3) Riferimenti Tecnici

- Documentazione ufficiale Typescript: Typescript
- Documentazione ufficiale Langchain: Langchain
- Documentazione ufficiale NodeJs: Node.js
- Documentazione ufficiale NestJs: Nest.js
- Documentazione ufficiale Groq: GroqCloud
- Documentazione ufficiale Qdrant: Qdrant
- Documentazione ufficiale NomicAi: NomicAi
- Documentazione ufficiale PostgreSQL: PostgresSQL
- Documentazione ufficiale Oktokit: Octokit
- Documentazione JiraJs: JiraJs
- Documentazione Confluence Js: ConfluenceJs
- Documentazione ufficiale Docker: Docker
- Documentazione ufficiale ReactJs: React
- Documentazione ufficiale ReactQuery (TanStack) ReactQuery
- Documentazione ufficiale TailwindCSS: Tailwind CSS
- Documentazione ufficiale NextJs Next.js

2) Tecnologie

In questo capitolo sono elencate tutte le tecnologie della ***tech stack***$_G$ che il team utilizza per lo sviluppo del progetto di azzurrodigitale;, come linguaggi di programmazione, ***framework***$_G$, ***librerie***$_G$ e ***ambienti di sviluppo***$_G$.

2.1) Typescript

Typescript è un linguaggio di programmazione open-source. È un super-set di JavaScript, che aggiunge forte tipizzazione statica. Il team ha scelto di utilizzare Typescript per la sua tipizzazione statica, che permette di ridurre gli errori di programmazione e di rendere il codice più leggibile e manutenibile.

Figura 2: Logo Typescript

2.2) Langchain

Langchain è un framework open-source per la creazione di applicazioni basate sull'utilizzo *LLM*$_G$. Il team ha scelto di utilizzare Langchain per la sua facilità d'uso e per la sua integrazione con altri servizi come Qdrant e Groq, oltre che ad avere una libreria in Typescript, rendendolo compatibile con il nostro linguaggio.

Figura 3: Logo di Langchain

2.3) Node.js

Node.js è un ambiente di runtime open-source per l'esecuzione di codice JavaScript lato server. Il team ha scelto di utilizzare Node.js per la sua scalabilità e per la sua facilità di utilizzo.

Figura 4: Logo di Node.js

2.4) Nest.js

Nest.js è un framework per la creazione di applicazioni server-side in Node.js. Il team ha scelto di utilizzare Nest.js per la sua struttura modulare e per la sua scalabilità e per la facilità con cui è possibile creare i design pattern più opportuni.

Figura 5: Logo di Nest.js

2.5) GroqCloud

È una piattaforma AI basata su hardware specializzato (LPU) per inferenza ad alte prestazioni, supporta modelli LLM e integrazione con strumenti AI per elaborazione in tempo reale.

Figura 6: Logo di GroqCloud

2.6) Qdrant

Qdrant è un motore di ricerca e analisi di dati non strutturati, supporta l'indicizzazione e la ricerca di dati in tempo reale, oltre che la ricerca di dati basata su vettori.

Figura 7: Logo di Qdrant

2.7) NomicAi

NomicAi è un servizio di elaborazione del linguaggio naturale (NLP) basato su modelli LLM che permette l'embedding di testo. Il team ha scelto di utilizzare NomicAi per la sua facilità d'uso e per la sua integrazione con altri servizi come Langchain e Groq.

Figura 8: Logo di NomicAi

2.8) PostgreSQL

PostgreSQL è un sistema di gestione di database relazionale open-source. Il team ha scelto di utilizzare PostgreSQL per la sua affidabilità e per la sua estensiva documentazione.

Figura 9: Logo di PostgreSQL

2.9) Octokit

Octokit è un toolkit per l'interazione con le API di GitHub. Il team ha scelto di utilizzare Octokit per la sua estesa documentazione e per utilizzare un prodott ufficiale per interagire on GitHub stesso.

Figura 10: Logo di Octokit

2.10) JiraJs

JiraJs è un toolkit per l'interazione con le API di Jira. Il team ha scelto di utilizzare JiraJs per la sua documentazione affidabile e per la sua facilità d'uso.

Figura 11: Logo di JiraJs

2.11) ConfluenceJs

ConfluenceJs è un toolkit per l'interazione con le API di Confluence. Il team ha scelto di utilizzare ConfluenceJs per la sua documentazione affidabile e per la sua facilità d'uso.

Figura 12: Logo di ConfluenceJs

2.12) Docker

Docker è una piattaforma open-source per lo sviluppo, il deploy e l'esecuzione di applicazioni in container. Il team ha scelto di utilizzare Docker per la sua facilità di deploy e per la sua scalabilità.

Figura 13: Logo di Docker

2.13) React.js

ReactJs è una libreria open-source per la creazione di interfacce utente. Il team ha scelto di utilizzare ReactJs per la sua immediatezza nell'uso, per la sua scalabilità e per la sua estesa documentazione.

Figura 14: Logo di ReactJs

2.14) ReactQuery

ReactQuery è una libreria open-source per la gestione dello stato in React. Il team ha scelto di utilizzare ReactQuery per la sua integrazione con React.

Figura 15: Logo di ReactQuery

2.15) TailwindCSS

TailwindCSS è un framework CSS utilizzato per la creazione di interfacce utente. Il team ha scelto di utilizzare TailwindCSS per la sua facilità d'uso e per la sua documentazione dettagliata oltre che per utilizzare una tecnologia più compatibile con il resto.

Figura 16: Logo di TailwindCSS

2.16) Next.js

Next.js è un framework per la creazione di applicazioni web in React. Il team ha scelto di utilizzare Next.js per i metodi nativi a disposizione per le richieste alle API e per utilizzare una tecnologia più nuova rispetto al resto.

Figura 17: Logo di Next.js

3) Architettura di sistema

3.0.1) Github

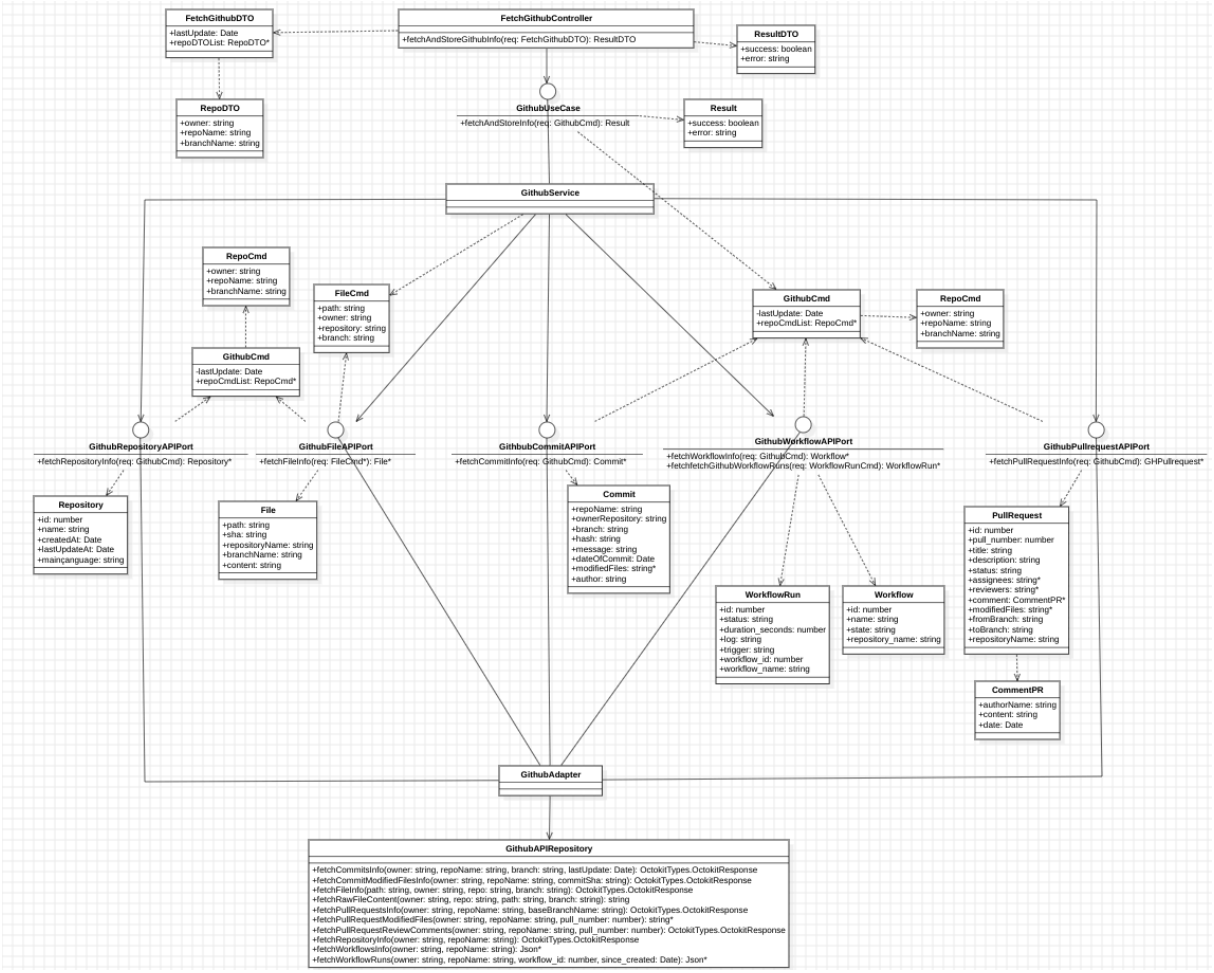


Figura 18: Diagramma UML di dettaglio riguardo alla raccolta delle informazioni di Github

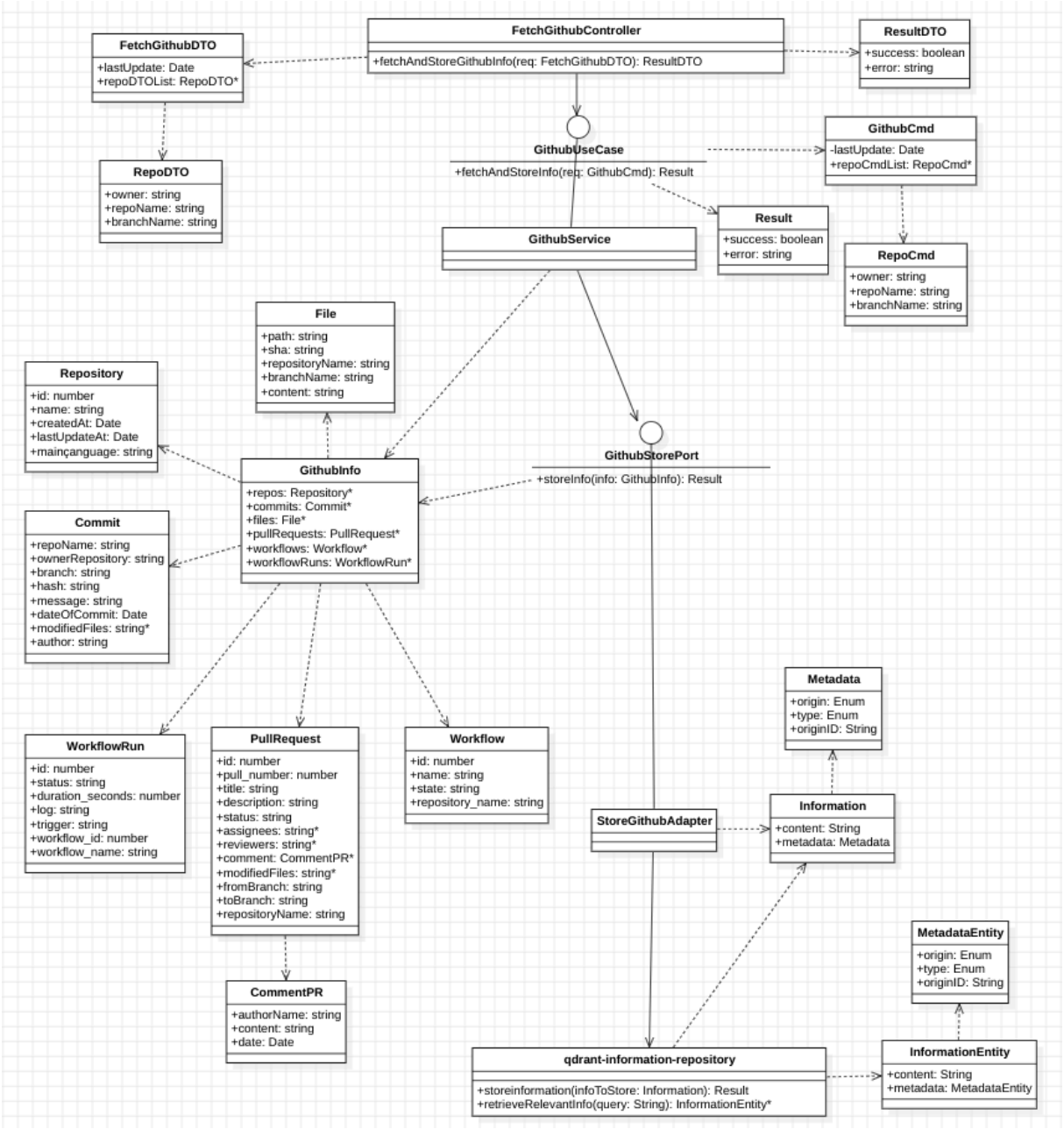


Figura 19: Diagramma UML di dettaglio riguardo al salvataggio delle informazioni di Github

3.0.1.1) FetchGithubDTO

Classe che viene ricevuta in input dall'`InformationController`, contiene una lista di `RepoDTO`, spiegati in seguito, e la data dall'ultima raccolta di informazioni.

```
export class FetchGithubDto {
  constructor (
    public readonly repoDTOList: RepoGithubDTO[],
    public readonly lastUpdate?: Date
  ){}
}
```

3.0.1.2) RepoDTO

Classe che contiene le informazioni necessarie a identificare univocamente la risorsa di cui vogliamo raccogliere le informazioni, ossia:

- a chi appartiene il repository su *Github*$_G$
- il nome del repository
- il branch del repository

```
export class RepoGithubDTO{
  constructor(
    public readonly owner: string,
    public readonly repoName: string,
    public readonly branch_name: string
  ){}
}
```

3.0.1.3) GithubCmd

Questa classe rappresenta il Command che riceve la business logic, contiene una lista di `RepoCmd`, che contiene gli stessi campi di `RepoDTO`, e lo stesso "lastUpdate" ricevuto nel `FetchGithubDto`

```
export class GithubCmd {
  constructor(
    public readonly repoCmdList:RepoCmd[],
    public readonly lastUpdate?:Date
  ){}
}
```

3.0.1.4) RepoCmd

Questa classe è la classe `RepoDTO` adattata alla business logic.

```
export class RepoCmd{
  constructor (
    public readonly owner: string,
    public readonly repoName: string,
    public readonly branch_name: string
  ){}
}
```

3.0.1.5) Commit

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei commit di una determinata repository.

```
export class Commit{
    constructor(
        private repoName: string,
        private ownerRepository: string,
        private branch: string,
        private hash: string,
        private message: string,
        private dateOfCommit: string,
        private modifiedFiles: string[],
        private author: string,
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }

    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.COMMIT, this.hash);
```

```
    }
}
```

3.0.1.6) File

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei files di un determinato branch in una determinata repository.

```
export class File{
    constructor(
        private path: string,
        private sha: string,
        private repositoryName: string,
        private branchName: string,
        private content: string
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }

    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.FILE, this.sha);
    }
}
```

3.0.1.7) PullRequest

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere delle pull requests in una determinata repository.

```
export class PullRequest{
    constructor(
        private id: number,
        private pull_number: number,
        private title: string,
        private description: string,
        private status: string,
        private assignees: string[],
        private reviewers: string[],
        private comments: CommentPR[],
        private modifiedFiles: string[],
        private fromBranch: string,
        private toBranch: string,
        private repository_name: string,
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }

    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.PULLREQUEST, this.id.toString());
    }
}
```

3.0.1.8) CommentPR

Questa classe è oggetto della business logic, è contenuta all'interno di `PullRequest` in quanto si occupa di contenere al suo interno le informazioni riguardanti un determinato commento di review su una `PullRequest`.

```
export class CommentPR{
    constructor(
        private authorName: string,
        private content: string,
        private date: Date
    ){}

    getAuthorName(): string {
        return this.authorName;
    }

    getContent(): string {
        return this.content;
    }

    getDate(): Date {
        return this.date;
    }
}
```

3.0.1.9) Repository

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere di una determinata repository.

```
export class Repository {
    constructor(
    private id: number,
    private name: string,
    private createdAt: string,
    private lastUpdate: string,
    private mainLanguage: string,
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

  getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.REPOSITORY, this.id.toString());
  }
}
```

3.0.1.10) Workflow

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei workflow in una determinata repository.

```
export class Workflow{
  constructor(
    private id: number,
    private name: string,
    private state: string,
    private repository_name: string,
```

```
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

  getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.WORKFLOW, this.id.toString());
  }
}
```

3.0.1.11) WorkflowRun

Questa classe è oggetto della business logic, è contenuta all'interno di `Workflow` in quanto si occupa di contenere al suo interno le informazioni riguardanti una determinata run di un `Workflow`.

```
export class WorkflowRun {
  constructor(
    private readonly id: number,
    private readonly status: string,
    private readonly duration_seconds: number,
    private log: string,
    private trigger: string,
    private workflow_id: number,
    private workflow_name: string
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

   getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.WORKFLOW_RUN, this.id.toString());
  }
}
```

3.0.1.12) GithubUseCase

Interfaccia che si comporta da porta d'ingresso alla business logic, offre il metodo `fetchAndStoreInfo`, che prende in input il `GithubCmd` ricevuto dal controller.

```
export interface GithubUseCase {
    fetchAndStoreGithubInfo(req: GithubCmd): Promise<boolean>;
}
```

3.0.1.13) GithubService

La classe principale della business logic, che implementa `GithubUseCase` citato precedentemente. Si occupa di recuperare tutte le informazioni descritte nell'*Analisi dei Requisiti*$_G$ e di salvarle nel database vettoriale.

3.0.1.14) GithubCommitAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchCommitInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Commit`.

```
export interface GithubCommitAPIPort {
    fetchGithubCommitsInfo(req: GithubCmd): Promise<Commit[]>
}
```

3.0.1.15) GithubFileAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubFilesInfo` che riceve in input `FileCmd` e ritorna in output una lista di `Commit`.

```
export interface GithubFilesAPIPort {
    fetchGithubFilesInfo(req: FileCmd[]): Promise<File[]>
}
```

3.0.1.16) GithubPullRequestAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubPullRequestsInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `PullRequest`.

```
export interface GithubPullRequestsAPIPort {
    fetchGithubPullRequestsInfo(req: GithubCmd): Promise<PullRequest[]>
}
```

3.0.1.17) GithubRepositoryAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubRepositoryInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Repository`.

```
export interface GithubRepositoryAPIPort {
    fetchGithubRepositoryInfo(req: GithubCmd): Promise<Repository[]>
}
```

3.0.1.18) GithubWorkflowAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre i metodi:

- `fetchGithubWorkflowInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Workflow`.
- `fetchGithubWorkflowRuns` che riceve in input `WorkflowRunCmd` e ritorna in output una lista di `WorkflowRun`.

```
export interface GithubWorkflowsAPIPort {
    fetchGithubWorkflowInfo(req: GithubCmd): Promise<Workflow[]>
    fetchGithubWorkflowRuns(req: WorkflowRunCmd): Promise<WorkflowRun[]>
}
```

3.0.1.19) GithubAPIAdapter

Questa classe implementa:

- `GithubCommitAPIPort`
- `GithubFileAPIPort`
- `GithubPullRequestAPIPort`
- `GithubRepositoryAPIPort`
- `GithubWorkflowAPIPort`

ponendosi come adapter tra la business logic e la classe che si occupa di fare le richieste API, ossia `GithubAPIFacade`. Trasforma infatti gli oggetti JSON "grezzi" ritornati da quest'ultima e li trasforma negli oggetti della business logic.

3.0.1.20) GithubAPIRepository

Questa è la classe che si occupa di interfacciarsi direttamente con le API di Github. Esegue richieste tramite il client offerto da `octo-kit` e ritorna JSON con i dati "grezzi".

3.0.1.21) GithubStoreInfoPort

Questa è l'interfaccia che funge da porta d'uscita (outbound port) al fine di salvare i `GithubInfo` nel database vettoriale, offre il metodo `storeGithubInfo` che riceve in input una lista di `GithubInfo`.

```
export interface GithubStoreInfoPort {
    storeGithubInfo(req: GithubInfo): Promise<boolean>
}
```

3.0.1.22) GithubStoreInfoAdapter

Questa classe implementa `GithubStoreInfoPort`, si occupa di trasformare i `GithubInfo` in `Information` per poter essere usati dal `qdrant-information-repository` ed essere salvati sul database vettoriale.

3.0.2) Confluence

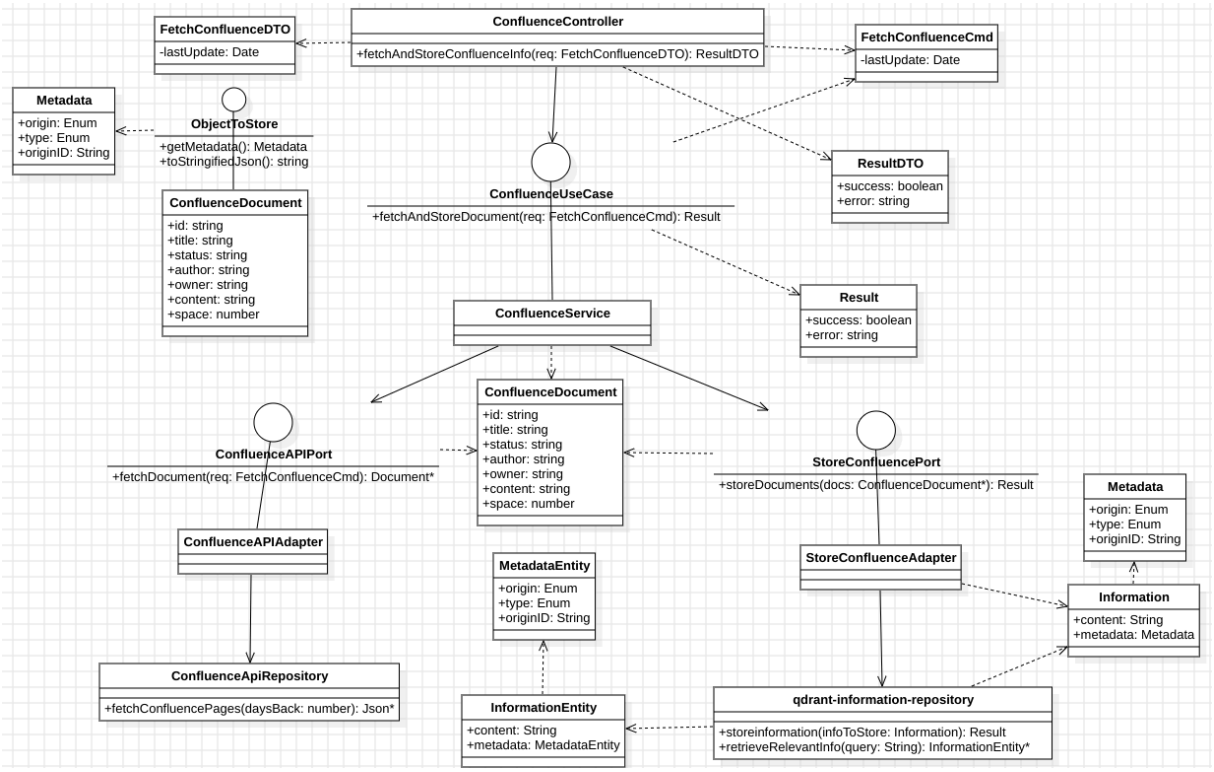


Figura 20: Diagramma UML di dettaglio riguardo a Confluence

3.0.2.1) ConfluenceController

Controller che resta in attesa di messaggi sulla coda `information-queue`, al fine di portare a termine le operazioni di raccolta e salvataggio delle informazioni ottenute da Confluence. Ritorna come output un oggetto `ResultDTO`.

3.0.2.2) ConfluenceUseCase

Interfaccia che si comporta da porta d'ingresso alla business logic, offre il metodo `fetchAndStoreDocument`, che prende in input il `ConfluenceCmd` ricevuto dal controller e ritorna come output un oggetto `Result`.

```
export interface ConfluenceUseCase {
  fetchAndStoreConfluenceInfo(req: ConfluenceCmd): Promise<Result>;
}
```

3.0.2.3) ConfluenceService

La classe principale della business logic, che implementa `ConfluenceUseCase` citato precedentemente. Si occupa di recuperare i documenti creati e modificati entro una certa data, presente all'interno di `ConfluenceCmd`.

```
export class ConfluenceService implements ConfluenceUseCase {
  constructor(
    @Inject(CONFLUENCE_API_PORT) private readonly confluenceAPIAdapter:
ConfluenceAPIPort,
    @Inject(CONFLUENCE_STORE_INFO_PORT) private readonly confluenceStoreAdapter:
ConfluenceStoreInfoPort
  ) {}

  async fetchAndStoreConfluenceInfo(req: ConfluenceCmd): Promise<Result> {
    const documents = await this.confluenceAPIAdapter.fetchDocuments(req);
    return await this.confluenceStoreAdapter.storeDocuments(documents);;
  }
}
```

3.0.2.4) ConfluenceDocument

Classe del domain, definisce le informazioni che vengono raccolte e viene usato come oggetto della business logic.

3.0.2.5) ConfluenceAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchDocuments` che riceve in input `ConfluenceCmd` e ritorna in output una lista di ConfluenceDocument.

3.0.2.6) ConfluenceAPIAdapter

Questa classe implementa `ConfluenceAPIPort`, ponendosi come adapter tra la business logic e la classe che si occupa di fare le richieste API, ossia `ConfluenceAPIFacade`. Trasforma infatti gli oggetti JSON "grezzi" ritornati da quest'ultima e li trasforma negli oggetti della business logic di `ConfluenceDocument`.

3.0.2.7) ConfluenceAPIRepository

Questa è la classe che si occupa di interfacciarsi direttamente con le API di Confluence. Esegue richieste tramite il client offerto da `confluence.js` e ritorna JSON con i dati "grezzi".

3.0.2.8) ConfluenceStorePort

Questa è l'interfaccia che funge da porta d'uscita (outbound port) al fine di salvare i `ConfluenceDocument` nel database vettoriale, offre il metodo `storeDocuments` che riceve in input una lista di `ConfluenceDocument`.

```
export interface ConfluenceStoreInfoPort {
  storeDocuments(req: ConfluenceDocument[]): Promise<Result>;
}
```

3.0.2.9) ConfluenceStoreAdapter

Questa classe implementa `ConfluenceStorePort`, si occupa di trasformare i `ConfluenceDocument` in `Information` per poter essere usati dal `qdrant-information-repository` ed essere salvati sul database vettoriale.