Progetto Ingegneria dei Sistemi Distribuiti

Gianluca Giardina - 1000015165
Gianmarco Basile - 1000001626

2024/2025

Indice

1. Descrizione generale del progetto

Il presente progetto ha l'obiettivo di sviluppare un sistema scalabile basato su microservizi per l'estrazione, l'arricchimento e l'indicizzazione di dati provenienti da documenti PDF e immagini. L'infrastruttura è progettata per consentire agli utenti di caricare documenti o immagini ed ottenere informazioni arricchite e strutturate, migliorando l'accessibilità e la fruizione dei contenuti.

Il sistema fornisce servizi di elaborazione avanzata per due tipologie di file:

- **Documenti PDF**: Estrazione del contenuto testuale con possibilità di applicare servizi opzionali di *Summarization* e *NLP*.
- **Immagini**: Estrazione del contesto visivo attraverso un servizio di *Context Extraction*.

Quando un utente carica un documento PDF, il sistema esegue automaticamente l'estrazione del testo. Oltre alla semplice estrazione, è possibile selezionare due servizi opzionali: il **Summarization**, che genera un riassunto automatico del contenuto, e il **NLP (Natural Language Processing)**, che arricchisce il testo con l'estrazione di entità nominate (Named Entity Recognition) e altre informazioni rilevanti. L'elaborazione avviene attraverso algoritmi avanzati di NLP, che permettono di strutturare e valorizzare i dati estratti.

Per quanto riguarda le immagini, il sistema consente di attivare il servizio di **Context Extraction**, che sfrutta algoritmi di Image-to-Text per generare una descrizione semantica del contenuto visivo. Questo processo trasforma l'informazione non strutturata in dati testuali facilmente interpre-

tabili, migliorando così la comprensione e l'accessibilità del contenuto visivo caricato.

1.1 Architettura del progetto

L'architettura del progetto è basata su un modello a **microservizi**, garantendo modularità, scalabilità ed efficienza nella gestione delle richieste. Il sistema è progettato per elaborare documenti PDF e immagini, distribuendo i carichi di lavoro tra diversi servizi specializzati che comunicano tra loro tramite il message broker **RabbitMQ**. I dati vengono memorizzati in un database **MongoDB**, che offre una struttura flessibile per la gestione delle informazioni estratte e arricchite.

- **Client**: L'interfaccia utente permette la registrazione e l'autenticazione degli utenti tramite JWT (JSON Web Token). Una volta loggato, l'utente può caricare documenti PDF o immagini, selezionare i servizi desiderati e visualizzare i risultati elaborati dal sistema.
- **Server Centrale**: Implementato in **Java** con **Spring Boot**, il server centrale è il punto di ingresso delle richieste utente. Si occupa di gestire il flusso di elaborazione, inoltrando i file ai microservizi appropriati:
  - **NLP Service** per l'analisi del testo tramite tecniche di Natural Language Processing.
  - **LLM Summarizer** per la generazione automatica di riassunti.
  - **Image to Text Service** per l'estrazione del contenuto semantico dalle immagini.

  Il server centrale comunica con i microservizi in modo asincrono attraverso RabbitMQ e registra lo stato delle richieste nel database MongoDB.
- **Pdf Text Extraction Service**: Servizio scritto in **Python** con **Flask** che riceve un file PDF, ne estrae il testo utilizzando la libreria **pdfminer.six** e lo restituisce al server centrale per ulteriori elaborazioni.
- **NLP Service**: Microservizio in **Python** che utilizza **spaCy** e altre librerie per eseguire operazioni di analisi testuale avanzata, come:

  - **Named Entity Recognition (NER)** per identificare nomi propri, luoghi e organizzazioni.

- **LLM Summarizer**: Servizio di sintesi testuale basato su modelli di **Large Language Models (LLM)**, come *GPT*, per generare riassunti pertinenti dei documenti analizzati.

- **Image to Text Service**: Servizio che utilizza modelli **Vision Transformer (ViT)**. er analizzare il contenuto visivo delle immagini, trasformandolo in un formato testuale interpretabile dal sistema.

- **Database - MongoDB**: Il sistema utilizza **MongoDB** come database NoSQL per la memorizzazione delle richieste, degli stati di elaborazione e dei risultati finali. MongoDB è scelto per la sua flessibilità nel gestire dati semi-strutturati e per la sua capacità di scalare facilmente in un'architettura distribuita.

- **Message Broker - RabbitMQ**: RabbitMQ è utilizzato per orchestrare la comunicazione tra i microservizi in modo asincrono. I messaggi vengono inseriti nelle seguenti code:

  - **nlp_queue**: per l'analisi testuale avanzata.
  - **summarize_queue**: per la generazione automatica di riassunti.
  - **context_queue**: per l'estrazione di informazioni contestuali dalle immagini.

- **Scalabilità e Distribuzione**: Grazie alla sua architettura a microservizi, il sistema può scalare orizzontalmente, aumentando il numero di istanze per i servizi più utilizzati. L'uso di **Docker** consente il deployment dinamico dei componenti, ottimizzando le risorse e migliorando la resilienza del sistema.

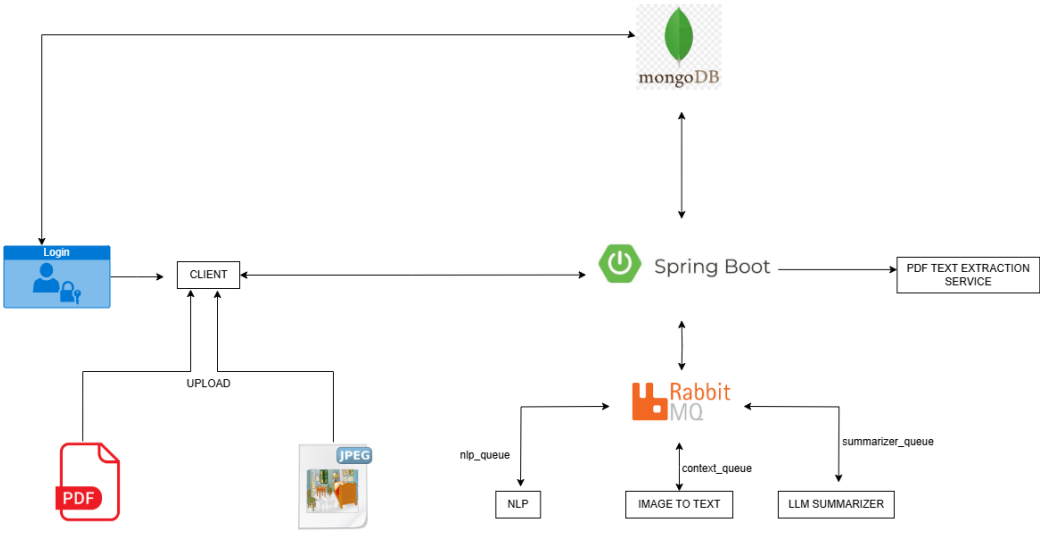

Figura 1.1: Schema dell'infrastruttura del progetto

1.2 Server Centrale SpringBoot

Il **Server Centrale** è il componente principale dell'architettura del sistema, sviluppato in **Spring Boot**. Il suo ruolo è quello di ricevere le richieste provenienti dal client, gestire il flusso di elaborazione e distribuire i task ai microservizi responsabili dell'analisi e trasformazione dei documenti PDF e delle immagini. Il server centrale si interfaccia con **MongoDB** per la gestione dello stato delle richieste e dei risultati elaborati, mentre la comunicazione con i microservizi avviene in maniera asincrona tramite **RabbitMQ** utilizzando il **connettore ufficiale di Spring Boot**.

1.2.1 Interfacce Esposte

Il Server Centrale espone una serie di API REST per interagire con il sistema:

- **Upload di Documenti e Immagini**
  - `POST /api/process_pdf`: Caricamento di un documento PDF e richiesta dei servizi opzionali di NLP e Summarization.
  - `POST /api/process_image`: Caricamento di un'immagine e richiesta del servizio di Context Extraction.
- **Recupero dello Stato delle Richieste**
  - `GET /api/getByUserId?userId={userId}`: Restituisce lo stato di tutte le richieste effettuate da un determinato utente.
  - `GET /api/results/get?requestId={requestId}`: Recupera i risultati dell'elaborazione associati a una richiesta specifica.
- **Gestione dello Stato dei Servizi**
  - `GET /api/status/getByRequestId?requestId={requestId}`: Restituisce lo stato di avanzamento di ogni microservizio coinvolto nell'elaborazione di una richiesta.

1.2.2 Flusso di Elaborazione

Il flusso di elaborazione varia a seconda del tipo di file caricato dall'utente:

- **PDF:**
  1. Il client invia un documento PDF al server centrale tramite una chiamata API.
  2. Il server centrale inoltra il PDF al **PDF Text Extraction Service** attraverso una richiesta HTTP REST, recuperando il testo contenuto nel documento.
  3. Il testo estratto viene memorizzato in **MongoDB** e, se richiesti, vengono attivati i servizi opzionali **Summarization** e **NLP**.
  4. Per l'esecuzione di questi servizi, il server centrale utilizza **CompletableFuture** per eseguire le elaborazioni in maniera asincrona, inoltrando i dati ai microservizi tramite RabbitMQ.
  5. Ogni microservizio elabora il testo e restituisce il risultato al server centrale, che aggiorna lo stato dell'elaborazione nel database.
- **Immagini:**
  1. Il client invia un'immagine al server centrale tramite una chiamata API.
  2. Il server centrale inoltra utilizzando **CompletableFuture** per eseguire un invocazione asincrona, l'immagine al microservizio **Image to Text Service** attraverso RabbitMQ per l'estrazione del contenuto testuale o semantico.
  3. Una volta completata l'elaborazione, il risultato viene restituito e memorizzato in **MongoDB**.

1.2.3 Integrazione con RabbitMQ

La comunicazione tra il server centrale e i microservizi avviene attraverso **RabbitMQ**, implementando un sistema di messaggistica asincrono basato sul meccanismo **Direct Reply-To** di RabbitMQ[1]. Questo approccio consente di ricevere una risposta diretta dai microservizi senza la necessità di creare

[1] https://www.rabbitmq.com/docs/direct-reply-to

e gestire code di risposta, riducendo la latenza nella comunicazione.
Ogni richiesta viene instradata in una delle seguenti code:

- **nlp_queue**: Coda dedicata al microservizio NLP per l'estrazione di entità e l'analisi del testo.
- **summarize_queue**: Coda dedicata alla generazione di riassunti tramite LLM.
- **context_queue**: Coda utilizzata per l'estrazione di contesto dalle immagini.

Ogni microservizio sottoscrive una coda specifica, elabora il messaggio e invia la risposta diretta al server centrale, che aggiorna lo stato della richiesta in MongoDB e memorizza i risultati ottenuti.

1.3 Gestione dello Stato delle Richieste

Lo stato delle richieste viene tracciato in **MongoDB** attraverso la collezione `processing_status`, che permette di monitorare l'avanzamento di ogni servizio. Il modello dei dati prevede tre entità principali:

- **RequestState**: Rappresenta la richiesta utente e mantiene informazioni sul tipo di file e sui servizi richiesti.
- **ProcessingStatus**: Tiene traccia dell'avanzamento dell'elaborazione per ogni servizio.
- **RequestResults**: Contiene i risultati finali dell'elaborazione di PDF e immagini.

1.3.1 Flusso di Aggiornamento dello Stato

Ad ogni avanzamento nell'elaborazione, il server centrale aggiorna lo stato della richiesta:

1. Quando un utente carica un file, viene creato un nuovo record in `RequestState` con stato iniziale **"pending"**.
2. Quando il file viene inoltrato a un microservizio, lo stato del servizio specifico viene impostato su **"in_progress"**.

3. Al termine dell'elaborazione, il microservizio restituisce il risultato e il server centrale aggiorna lo stato su **"completed"** o **"failed"** in caso di errore.

1.3.2 Design Pattern Utilizzati

L'architettura del server si basa su diversi **Design Pattern** per garantire modularità, scalabilità e facilità di manutenzione:

- **MVC (Model-View-Controller)**: Struttura il codice separando la logica di business dai controller REST, garantendo un'organizzazione chiara del codice.
- **Facade Pattern**: La classe `RequestProcessingFacade` centralizza l'accesso ai servizi di elaborazione, semplificando le chiamate e riducendo l'accoppiamento tra i componenti.
- **Strategy Pattern**: Il sistema di elaborazione è progettato con una gerarchia di classi che implementano l'interfaccia `ProcessingStrategy`, permettendo di selezionare dinamicamente la strategia di elaborazione più adatta (*Summarization, NLP, Context Extraction*).
- **Repository Pattern**: Per l'accesso ai dati in MongoDB, vengono utilizzate interfacce di repository che astraggono le operazioni CRUD, migliorando il codice.
- **Message Queue Pattern**: La comunicazione tra il server e i microservizi avviene tramite RabbitMQ, implementando un sistema di code per la gestione asincrona delle richieste.

1.4 PDF Text Extraction Service

Il **Consumer PDF Text Extraction** è un microservizio sviluppato in **Python** che si occupa di estrarre e pulire il contenuto testuale da documenti PDF ricevuti in formato **Base64**. Il servizio espone un'API REST sulla route `/extract/text` e accetta richieste in formato JSON, restituendo il testo estratto in formato strutturato.
Alla ricezione di un file PDF codificato in Base64, il servizio decodifica il contenuto e lo converte in un flusso di dati utilizzando la libreria **io.BytesIO**. Il testo viene quindi estratto attraverso la libreria **pdfminer.six**, per l'estrazione del testo presente nei documento PDF.
Dopo l'estrazione, il testo viene filtrato per rimuovere caratteri non necessari e migliorare la leggibilità. Il processo di pulizia comprende la sostituzione dei ritorni a capo e delle tabulazioni con spazi, oltre alla rimozione degli spazi multipli attraverso un'espressione regolare. Questo garantisce che il testo sia correttamente formattato prima di essere restituito al **Server Centrale**.

1.5 Consumer Summarization

Il **Consumer Summarization** è un microservizio sviluppato in **Python** che esegue la generazione automatica di riassunti utilizzando un modello di **Large Language Models (LLM)**, in particolare `Flan-T5-Large`. Il servizio è progettato per ascoltare sulla coda `summarize_queue` di **RabbitMQ** e rispondere direttamente al **Server Centrale** tramite il meccanismo **Direct Reply-To**.
Il consumer attende l'arrivo di un messaggio contenente un testo da sintetizzare. Alla ricezione, la funzione di *callback* utilizza il modello di summarization, preceduto da un prompt specifico in stile T5 per guidare il modello nell'elaborazione del riassunto. Il processo di generazione del testo avviene tramite un meccanismo di beam search con penalità sulla lunghezza e restrizioni sugli *n-grammi ripetuti*, al fine di ottenere un output più coeso e informativo.
Se la richiesta proviene da una chiamata diretta RPC, il risultato viene inviato immediatamente al Server Centrale utilizzando **Direct Reply-To**, evitando così la creazione di code dedicate. Dopo l'elaborazione, il messaggio viene confermato a RabbitMQ per garantire che non venga reinviato.
L'architettura del Consumer Summarization. Grazie all'integrazione con RabbitMQ, il microservizio può essere scalato orizzontalmente per gestire un elevato numero di richieste concorrenti, migliorando la reattività complessiva del sistema.

1.6 Consumer NLP

Il **Consumer NLP** è un microservizio sviluppato in **Python** che estrae entità nominate (*Named Entity Recognition, NER*) dal testo utilizzando la libreria **spaCy**. Il servizio ascolta sulla coda `nlp_queue` di **RabbitMQ** e risponde direttamente al **Server Centrale** tramite il meccanismo **Direct Reply-To**.
Una volta ricevuto un messaggio, la funzione di *callback* esegue le seguenti operazioni: decodifica il testo, applica il modello NLP di spaCy per l'analisi ed estrae le entità rilevanti. Il risultato viene inviato al Server Centrale pubblicando un messaggio con la stessa `correlation_id` della richiesta originale, permettendo di associare correttamente la risposta.
L'uso di **Direct Reply-To** semplifica la gestione delle code di risposta, eli-

minando la necessità di code dedicate per ogni richiesta. Dopo l'elaborazione, il messaggio viene confermato a RabbitMQ.
Il consumer è progettato per rimanere in attesa di nuovi messaggi e può essere scalato orizzontalmente avviando più istanze. Questa configurazione consente di gestire in modo efficiente un elevato numero di richieste concorrenti, migliorando la scalabilità e la reattività del sistema.

1.7 Consumer Image-to-Text

Il **Consumer Image-to-Text** è un microservizio sviluppato in **Python** che si occupa di generare una descrizione testuale delle immagini ricevute utilizzando un modello di **image captioning** basato su **ViT-GPT2**. Il servizio ascolta sulla coda `context_queue` di **RabbitMQ** e restituisce il risultato direttamente al **Server Centrale** tramite il meccanismo **Direct Reply-To**..
Il consumer si connette a RabbitMQ e attende l'arrivo di un messaggio contenente un'immagine codificata in **Base64**. Alla ricezione, il messaggio viene decodificato e convertito in un oggetto immagine utilizzando la libreria **Pillow (PIL)**. Successivamente, il modello **ViT-GPT2** genera una descrizione testuale dell'immagine.
Se la richiesta proviene da una chiamata diretta RPC, il consumer risponde immediatamente utilizzando **Direct Reply-To**, evitando la creazione di code di risposta dedicate.
Dopo l'elaborazione, il messaggio viene confermato a RabbitMQ per garantire che non venga reinviato.
L'architettura è progettata per supportare la scalabilità orizzontale, consentendo l'avvio di più istanze del consumer per gestire un alto volume di richieste in parallelo.

1.8 Server di Autenticazione

Il sistema di autenticazione del progetto è basato su un microservizio sviluppato in **Flask**, progettato per gestire la registrazione, l'autenticazione e la protezione delle API tramite **JWT (JSON Web Token)**. Il server è containerizzato con **Docker** e utilizza **MongoDB** per la memorizzazione degli utenti.

1.8.1 Tecnologie Utilizzate

- **Flask**: Framework web per la gestione delle API REST.
- **Flask-JWT-Extended**: Libreria per la gestione dei token JWT.
- **MongoDB (pymongo)**: Database NoSQL per la memorizzazione degli utenti.
- **bcrypt**: Per la gestione sicura delle password.
- **Flask-CORS**: Per abilitare richieste da domini diversi.
- **dotenv**: Per la gestione delle variabili d'ambiente.
- **Waitress**: Server di produzione per l'esecuzione dell'applicazione.

1.8.2 Flusso di Autenticazione

1. **Registrazione (POST /register)**:
   - Riceve username e password in formato JSON.
   - Verifica che i campi siano presenti.
   - Genera un hash della password con **bcrypt**.
   - Salva le credenziali in **MongoDB**.
2. **Login (POST /login)**:
   - Riceve username e password.
   - Controlla se le credenziali esistono in MongoDB.
   - Se valide, genera un **token JWT** con scadenza predefinita.
   - Il token viene inviato al client e usato per le richieste successive.
3. **Protezione delle API**:
   - Le route protette usano il decoratore `@jwt_required()`.
   - Gli utenti devono includere il token JWT nell'header `Authorization` per accedere alle risorse protette.

1.8.3 Configurazione del Servizio

Le configurazioni principali sono definite in un file `.env`:

```
SECRET_KEY=supersecretkey
JWT_SECRET_KEY=jwtsecretkey
MONGO_URI=mongodb://mongo:27017/auth_db
JWT_TOKEN_LOCATION=cookies
```

Il servizio è containerizzato con **Docker**, e il **Dockerfile** definisce l'ambiente di esecuzione con le dipendenze necessarie.

Il server di autenticazione garantisce un accesso sicuro agli utenti, fornendo un'architettura scalabile e modulare. Grazie all'uso di **JWT**, MongoDB e Flask, il sistema pu'o essere facilmente integrato con gli altri microservizi della piattaforma, mantenendo un alto livello di sicurezza e protezione dei dati.

1.9 Client

Il client del sistema è una web application sviluppata con **React** e **Vite**, progettata per consentire agli utenti di interagire con il backend, caricare documenti e visualizzare i risultati dell'elaborazione.

1.9.1 Tecnologie Utilizzate

- **React**: Framework front-end per la creazione di interfacce utente dinamiche.
- **React Router**: Libreria per la gestione multi route della Single Page Application.
- **TailwindCSS**: Framework CSS per la gestione degli stili dell'applicazione.
- **Vite**: Ambiente di sviluppo e build tool per applicazioni web moderne.
- **Docker**: Il client è containerizzato per una facile distribuzione.

1.9.2 Struttura del Progetto

Il codice è organizzato in una serie di componenti modulari, ciascuno responsabile di una funzionalità specifica:

- **App.jsx**: Punto di ingresso principale dell'applicazione.
- **UploadPDF.jsx**: Interfaccia per il caricamento di documenti PDF.
- **Requests.jsx**: Gestione delle richieste utente e del loro stato.
- **Results.jsx**: Visualizzazione dei risultati elaborati dal backend.

1.9.3 Flusso di Interazione

L'utente può interagire con il sistema seguendo il seguente flusso:

1. Caricamento di un documento PDF tramite l'interfaccia utente.
2. Il documento viene inviato al backend tramite un'API REST.
3. Il backend elabora il file e invia i risultati al client.
4. L'utente visualizza i risultati in una dashboard interattiva.

1.9.4 Configurazione del Servizio

Le configurazioni principali sono definite in un file `.env`:

```
BASE_URL = http://localhost:8080/api
UPLOAD_PDF_URL = http://localhost:8080/api/process_pdf
UPLOAD_IMAGE_URL = http://localhost:8080/api/process_image
STATUS_BY_USER_URL = http://localhost:8080/api/status/getByUserId
STATUS_URL = http://localhost:8080/api/status/get
RESULTS_URL = http://localhost:8080/api/results/get
LOGIN_URL = http://localhost:5002/login
REGISTER_URL = http://localhost:5002/register
VALIDATE_TOKEN_URL = http://localhost:5002/api/validate_token
```

Il servizio è containerizzato con **Docker**, e il **Dockerfile** definisce l'ambiente di esecuzione con le dipendenze necessarie.

Il client web fornisce un'interfaccia semplice e intuitiva per interagire con il backend del sistema. Grazie all'uso di **React** e **Vite**, l'applicazione è veloce, modulare e facilmente estensibile per future funzionalità.

1.10 Conclusioni

Il progetto ha sviluppato un sistema scalabile e modulare per l'elaborazione automatizzata di documenti PDF e immagini, sfruttando un'architettura a microservizi integrata con **RabbitMQ** e **MongoDB**. Grazie all'uso di tecnologie di elaborazione del linguaggio naturale (**spaCy**, **T5**, **GPT**) e di modelli di visione artificiale (**ViT**), il sistema è in grado di estrarre, arricchire e indicizzare informazioni in modo efficiente.

L'adozione di **RabbitMQ** come message broker ha permesso di ottimizzare la comunicazione asincrona tra i microservizi, garantendo un'elaborazione distribuita e reattiva. Inoltre, l'utilizzo del meccanismo **Direct Reply-To** ha ridotto la complessità nella gestione delle risposte, migliorando la latenza delle operazioni. La persistenza dei dati in **MongoDB** ha assicurato un'archiviazione flessibile e adatta a gestire informazioni semi-strutturate.

L'architettura implementata consente un'elevata scalabilità orizzontale, permettendo di distribuire il carico tra più istanze dei microservizi a seconda della domanda. Inoltre, l'uso di tecnologie containerizzate come **Docker** e **Kubernetes** facilita il deployment e la gestione delle risorse.

I risultati ottenuti dimostrano la validità del modello adottato e aprono la strada a possibili sviluppi futuri. Tra le possibili estensioni del progetto vi sono:

- L'integrazione con modelli di intelligenza artificiale più avanzati per migliorare la qualità dell'analisi testuale e della generazione di riassunti.
- L'ottimizzazione delle prestazioni dei microservizi attraverso tecniche di caching e parallelizzazione avanzata.
- L'estensione del sistema per supportare nuovi formati di documenti e linguaggi diversi.