

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

**PROGETTAZIONE E SVILUPPO DI
UN'API REST PER UN AMBIENTE
DI LETTURA E CONFRONTO
TRA ARTICOLI SCIENTIFICI**

Relatore:
Prof.
ANGELO DI IORIO

Presentata da:
STEFANO NOTARI

Correlatore:
Prof.
FRANCESCO POGGI

Sessione I
Anno Accademico 2020/2021

Introduzione

In questa tesi si discuterà della progettazione e dello sviluppo di un'applicazione API REST per il progetto DocuDipity. DocuDipity è stato sviluppato dal DASPLab (Digital and Semantic Publishing Lab), un gruppo di ricerca dell'Università di Bologna, con l'obiettivo di sviluppare un web tool per l'analisi e il confronto di articoli scientifici [30]. Nella versione precedente, DocuDipity permetteva ad un utente di analizzare un certo articolo preso da un insieme precaricato. L'analisi consisteva nel consultare l'articolo nelle due visualizzazioni supportate, la visualizzazione sequenziale classica (Hypertext) e una gerarchica (SunBurst [37]). Il vantaggio di utilizzare tecniche di visualizzazioni differenti consiste nella possibilità di poter scegliere quali caratteristiche di un articolo mettere in evidenza, per esempio la visualizzazione ipertestuale mostra le informazioni in dettaglio mentre la visualizzazione SunBurst mette in risalto la struttura dell'articolo. L'implementazione precedente di DocuDipity, tuttavia, a causa di una non ottimale progettazione presentava alcune criticità. Come ad esempio, l'utilizzo di un modello dati che non formalizza i concetti di articolo e di visualizzazione, determina l'impossibilità di censire nuovi articoli o visualizzazioni. Infatti, non era possibile effettuare ricerche attraverso i metadati di articoli e modificare le visualizzazioni supportate. Per le ragioni sopra indicate, lo sviluppo di tale progetto è stato diviso in due parti:

1. Applicazione back-end: responsabile dell'implementazione delle API per la gestione delle risorse necessarie per il corretto funzionamento dell'applicazione
2. Applicazione front-end: responsabile dell'implementazione dell'interfaccia grafica e della presentazione dei dati ottenuti dall'interrogazione delle API messe a disposizione dal back-end, presentata nella tesi [26]

In questo elaborato verrà discussa la realizzazione dell'applicazione back-end, con l'obiettivo di superare le limitazioni dell'implementazione precedente. La realizzazione dell'applicazione si è suddivisa in più fasi, una di progettazione, una di sviluppo e una di test. Nella fase di progettazione ci si è concentrati nella realizzazione di un modello dati con il quale rappresentare le risorse che l'applicazione dovrà gestire. Successivamente, su tale modello sono state progettate le API REST e documentate con Swagger [36]. Nella fase di sviluppo sono state scelte quali tecnologie utilizzare e come strutturare l'architettura dell'applicazione per facilitare lo sviluppo e la manutenzione. La fase di

test può essere divisa in due parti: una relativa ai test funzionali e una relativa ai test di performance. La parte relativa ai test funzionali è stata eseguita a stretto contatto con la fase di sviluppo, in quanto ultimato lo sviluppo di una nuova API, questa veniva testata per verificare il corretto funzionamento nei casi d'uso previsti. La parte relativa ai test di performance, invece, è stata eseguita una volta ultimato lo sviluppo dell'applicazione in quanto necessitava di utilizzare differenti funzionalità per produrre risultati significativi. L'organizzazione dell'elaborato è strutturato nel seguente modo:

- nel primo capitolo si introdurrà il progetto su cui si basa la tesi, concentrandosi soprattutto sugli obiettivi di tale progetto e sulle limitazioni riscontrate
- nel secondo capitolo si affronterà la fase di progettazione, soffermandosi sulla definizione del modello dati e delle relative API
- nel terzo capitolo si discuterà della fase di implementazione, in particolare, sull'esposizione delle tecnologie e sull'architettura utilizzate
- nel quarto capitolo verranno presentati i test effettuati sull'applicativo, utilizzati per valutare la qualità del software prodotto e delle relative prestazioni
- nel quinto capitolo, infine, si discuterà delle conclusioni e dei possibili sviluppi futuri

Indice

Introduzione	i
1 Il progetto DocuDipity	1
1.1 Infoview per l'analisi di collezioni e di articoli scientifici	1
1.2 DocuDipity: introduzione e concetti principali	3
1.2.1 Sunburst view	5
1.2.2 Hypertext view	6
1.2.3 Coordinated views	6
1.3 DocuDipity "sul campo": un esempio d'uso	7
1.3.1 Posizione delle citazioni	7
1.3.2 Stili di scrittura	9
1.4 Limiti e criticità	11
2 La progettazione delle API di DocuDipity	12
2.1 API e REST	12
2.1.1 I principi REST	13
2.1.2 Approfondimento dei principi REST (Uniform Interface)	13
2.1.3 I metodi HTTP	14
2.1.4 Livelli di maturità di un'applicazione REST	16
2.2 Il modello dati e risorse	17
2.2.1 FRBR	17
2.2.2 Modello Entità-Relazione	18
2.3 Documentazione delle API di DocuDipity	20
2.3.1 Swagger	20
2.3.2 Organizzazione delle API e struttura base	20
3 Implementazione delle API e servizi server-side	25
3.1 Le tecnologie utilizzate	25
3.1.1 Node.JS e Express	26
3.1.2 Persistenza e Database	27
3.1.3 Autorizzazione mediante JWT (JSON Web Tokens)	28

3.2	Docker e Docker-Compose	29
3.2.1	Persistenza dei dati in Docker	31
3.3	Architettura dell'applicazione	33
3.3.1	Model-View-Controller	33
3.3.2	Esempio di implementazione Endpoint	34
3.3.3	Discussione dell'architettura del progetto	38
4	Valutazione	42
4.1	I requisiti funzionali	42
4.1.1	Gli strumenti utilizzati	43
4.1.2	La struttura dei test	43
4.1.3	I risultati	47
4.2	I requisiti non funzionali	48
4.2.1	La struttura del test	49
4.2.2	I risultati	51
5	Conclusioni	54

Capitolo 1

Il progetto DocuDipity

1.1 Infoview per l'analisi di collezioni e di articoli scientifici

Esistono diverse tecniche per la visualizzazione di un articolo che si affiancano alla tradizionale lettura sequenziale, infatti a seconda del tipo di lettore, dell'obiettivo e del tempo che quest'ultimo ha intenzione di impiegare per leggere il documento, ci sono metodologie, seppur ancora non del tutto esplorate, che permettono di scoprire ed ottenere nuove informazioni senza dover necessariamente leggere tutto il contenuto dell'articolo. Attualmente molti sistemi adottano ancora visualizzazioni statiche, come quella lineare di tipo ipertestuale, con il conseguente problema di non riuscire a rappresentare in maniera ottimale la struttura logica di documenti, che a causa dell'aumento dei volumi dei dati da gestire è diventata sempre più nidificata e complessa da rappresentare su un display. Questo problema si traduce in una difficoltà per i lettori che si interfacciano all'articolo, ognuno con esigenze e tempi differenti da dedicare alla lettura; a volte può essere sufficiente un colpo d'occhio per ottenere le informazioni di cui si ha bisogno, in altre invece il lettore ha la necessità di esplorare il documento per intero, analizzando ogni singola parola, valutando le citazioni presenti nell'articolo, ecc.

A tal appunto, è doveroso citare l'articolo di K.Hornbæk "Reading Pattern and usability in visualizations of electronic documents"[19], in cui è stata valutata l'efficacia di vari pattern di lettura, chiedendo a 20 soggetti scrittori di articoli scientifici di rispondere ad una serie di domande, utilizzando tre interfacce di lettura differenti: una lineare, una fisheye¹ ed una di "overview + detail". L'esperimento ha evidenziato come differenti interfacce di lettura influenzino il modo in cui il lettore legge e trae informazioni dal documento stesso, in particolare l'interfaccia di lettura lineare ha portato a risultati

¹Nell'interfaccia fisheye certe parti del documento sono considerate più importanti rispetto ad altre. Le parti più importanti del documento sono sempre leggibili mentre quelle meno importanti risultano essere inizialmente non leggibili, ma possono essere rese comprensibili cliccando su di esse con il mouse

complessivamente inferiori rispetto agli altri, quella fisheye è risultata essere la più veloce ma i soggetti che l'hanno utilizzata hanno ottenuto il minor punteggio di “apprendimento accidentale” di informazioni, ed infine quella di overview + detail, che è risultata essere la preferita dai lettori e quella con cui sono stati ottenuti risultati migliori. Le tecniche alternative alla visualizzazione lineare ipertestuale, sono rappresentate da strutture che permettono di mostrare visivamente grandi volumi di dati in schemi sempre più piccoli, organizzando le informazioni in maniera gerarchica ed ottimizzando gli spazi ridotti.

Per garantire un efficiente utilizzo degli spazi a disposizione per rappresentare dati, è preferibile utilizzare visualizzazioni gerarchiche implicite rispetto a visualizzazione esplicite (tecniche che mostrano in modo esplicito le relazioni tra nodi con archi e linee di collegamento)[27].

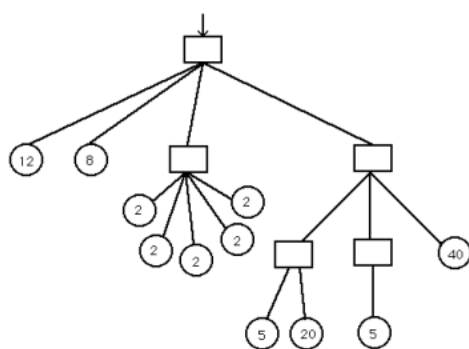
Una tra le tecniche alternative di visualizzazione più popolari è il Treemap[33], una tecnica “slice-and-dice” che riempie lo spazio a disposizione sulla base di una pianta rettangolare. Ogni nodo dell'albero è rappresentato da un rettangolo che contiene ulteriori rettangoli annidati tra loro, rappresentando così un'estensione della struttura ed usando al meglio lo spazio di visualizzazione disponibile

Numerose sono state le proposte di variazioni e miglioramenti delle tecniche di treemap iniziali (3D Treemap[20], Triangular Aggregated Treemap[4], Quantum Treemap[1], Cascaded Treemap[25], ecc.).

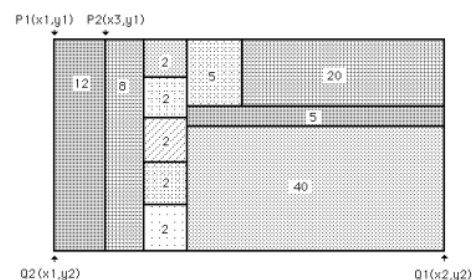
In questo elaborato, verrà data maggiore enfasi ad un'altra tecnica per la visualizzazione gerarchica di articoli, ossia la tecnica Sunburst[38], utilizzata in DocuDipity.

Un sommario di tutte le metodologie e approcci proposti negli ultimi 30 anni è descritto in [32].

In Figura 1.1a e Figura 1.1b vediamo come una struttura ad albero viene rappresentata attraverso Treemap.



(a) Struttura ad albero a 3 livelli



(b) Treemap di Figura 1.1a

1.2 DocuDipity: introduzione e concetti principali

L'intero lavoro è stato realizzato sulla base di un sistema già esistente chiamato DocuDipity[30].

DocuDipity è un web-tool interattivo che consente l'esplorazione e l'analisi di articoli scientifici, fornendo visualizzazioni alternative di quest'ultimi. Il tool affianca alla semplice visualizzazione dell'ipertesto del documento, il cui contenuto può essere visionato e letto in maniera sequenziale, una visualizzazione coordinata a quella ipertestuale, che permette di visualizzare i documenti (in particolare quelli XML), in maniera gerarchica, attraverso la tecnica SunBurst.

Il tool si basa su un principio cardine della disciplina Visual Analytics, ossia "Overview first, zoom and filter, then details-on-demand" [34].

L'idea è quella di avere a disposizione un'interfaccia in grado di dare una panoramica complessiva dei dati a disposizione (overview), ridurre la complessità di rappresentazione rimuovendo le informazioni non di interesse e focalizzandosi invece su quelle di interesse (zoom and filter), ed infine permettere all'utente di selezionare item specifici o gruppi di item di interesse ed ottenere informazioni aggiuntive (details-on-demand). Questo principio è stato applicato in DocuDipity per fare analisi su un set di articoli scientifici caricati su un server, al fine di scoprire caratteristiche e renderle evidenti ai lettori.

Il nome DocuDipity ha origine proprio dal termine che fa riferimento alla scoperta di qualcosa di inaspettato mentre si sta cercando altro, ossia serendipità (dall'inglese 'serendipity').

La scoperta di un nuovo comportamento inaspettato, emerge da un processo di ricerca di informazioni, che ha un significato diverso a seconda del contesto a cui viene applicato, tant'è che P.Kingrey in "Concepts of information seeking and their presence in the practical library literature", definisce la ricerca di informazioni[23] come un concetto più generale: "information seeking involves the search, retrieval, recognition, and application of meaningful content. This search maybe explicit or implicit, the retrieval may be the result of specific strategies or serendipity, the resulting information may be embraced or rejected, ... and there may be a million other potential results."

Il sistema DocuDipity è incentrato su un motore di estrazione di informazioni che si basa sulla teoria dei pattern strutturali[6], il cui algoritmo ha l'obiettivo di individuare un insieme minimale di elementi strutturali in grado di esprimere in maniera omogenea la struttura di un qualunque documento testuale (in formato XML). Il motore di estrazione utilizzato rende DocuDipity schema-agnostico, consentendogli di elaborare qualsiasi documento XML senza aver bisogno di alcuna conoscenza preventiva dell'organizzazione e della semantica del vocabolario XML utilizzato. A tal proposito, DocuDipity è stato testato su circa 1500 documenti liberamente disponibili scritti in 10 differenti vocabolari XML sviluppati per contesti eterogenei (libri, articoli scientifici, documentazioni software, lyrics, ecc.). Per quanto riguarda l'obiettivo di questa tesi, l'applicazione di-

retta della teoria dei pattern non è centrale, in quanto tutto il lavoro è stato svolto su documenti già “pattern-based”.

La precedente interfaccia grafica era formata da quattro parti principali, come mostrato in Figura 1.2. Era presente una parte superiore con una barra di navigazione che consentiva all’utente di selezionare il documento da investigare, visualizzare il documento selezionato nella sezione centrale del tool attraverso le due viste coordinate menzionate poc’anzi (sulla sinistra la view basata sul modello Sunburst che fornisce un sommario della struttura del documento, e sulla destra quella ipertestuale). In aggiunta alle due viste mirate alla lettura ed esplorazione di articoli, in DocuDipity è stato anche integrato un terzo componente, ossia un editor attraverso il quale l’utente può impostare delle regole di colorazione per evidenziare caratteristiche rilevanti e pattern dei documenti che sta analizzando.

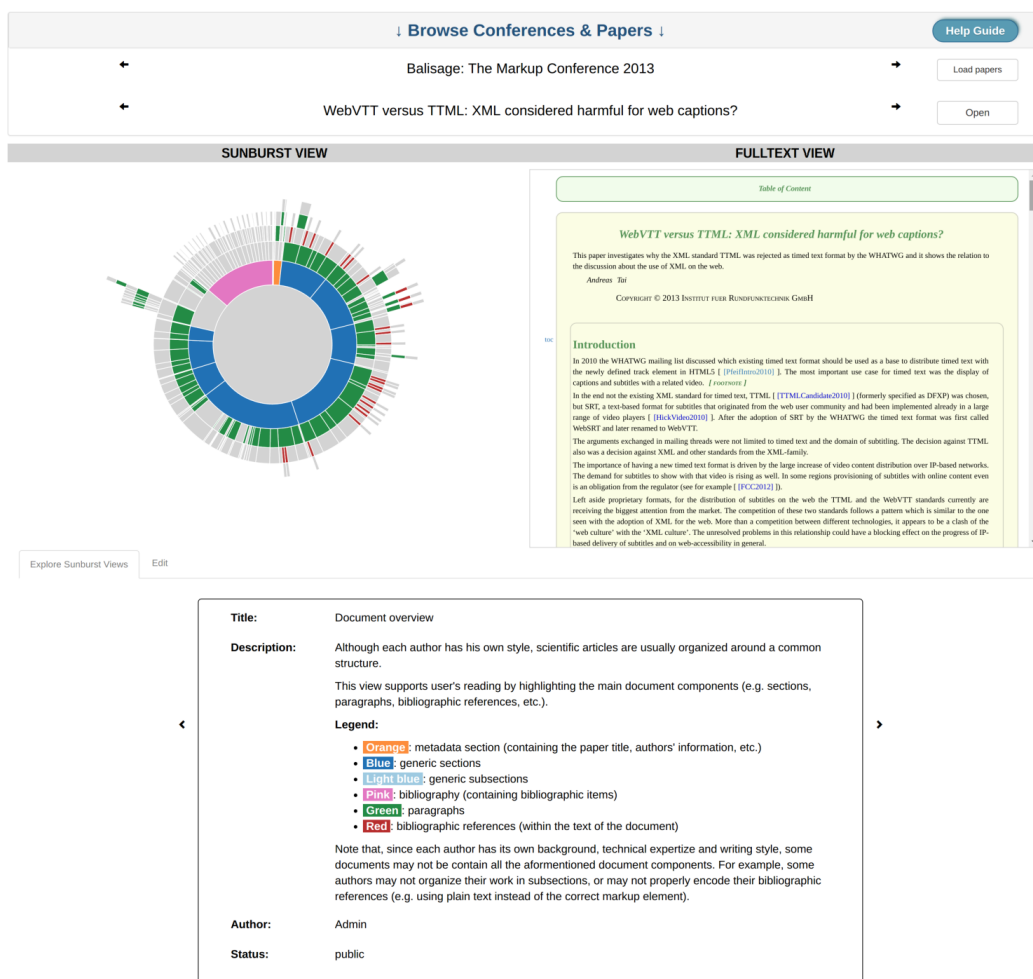


Figura 1.2: Una overview della precedente interfaccia DocuDipity

1.2.1 Sunburst view

Il Sunburst è una tecnica di visualizzazione che permette di rappresentare dati gerarchici su livelli circolari (detti anche corone circolari), il cui centro rappresenta l'elemento radice della collezione dati. Nello specifico, una corona circolare può essere composta da più segmenti che rappresentano gli elementi della collezione e che hanno una relazione gerarchica con il segmento della corona circolare esterna. Senza l'utilizzo di regole grafiche per distinguere la tipologia degli elementi, è difficilmente comprensibile la struttura del documento che si sta esplorando. Per rimediare al problema sopra citato, è stata aggiunta la funzionalità che raggruppa in maniera automatica segmenti dello stesso tipo in gruppi, ognuno dei quali ha associato un certo colore e mostrando la relativa legenda. In questo modo, è immediatamente riconoscibile la tipologia di appartenenza di un segmento e la possibilità di avere una veloce comprensione della struttura del progetto alla prima vista.



Figura 1.3: Esempi di visualizzazioni SunBurst dove i colori indicano i vari componenti strutturali dell'articolo

1.2.2 Hypertext view

La vista ipertestuale è costruita da una rappresentazione XML del documento alla quale si applica una ricerca basata su pattern per identificare la struttura del documento originale. Dal risultato ottenuto si produce una rappresentazione HTML basata su contenitori generici a cui si applicano regole CSS e JavaScript. La vista ipertestuale ha come vantaggio quello di avere una buona rappresentazione del dettaglio di documenti ma tende a offrire una scarsa panoramica del documento nel suo insieme, in quanto utilizzando una visualizzazione sequenziale con documenti di grandi dimensioni è possibile mostrare solo una parte del documento perdendo di vista la sua struttura.



Figura 1.4: Visualizzazione ipertestuale di un articolo scientifico

1.2.3 Coordinated views

La principale funzionalità implementata e messa a disposizione da DocuDipity riguarda la sincronizzazione delle visualizzazioni. Ovvero, durante l'analisi di un documento, quando l'utente seleziona un elemento del SunBurst, la visualizzazione ipertestuale si sincronizza evidenziando il medesimo elemento. Inoltre, quando si posiziona il cursore sopra ad un frammento di testo nella visualizzazione ipertestuale, nel SunBurst viene evidenziato sia il frammento di testo che gli elementi gerarchicamente correlati a tale frammento. Il beneficio di tale funzionalità è la possibilità di combinare i principali vantaggi delle due visualizzazioni, cioè la capacità del SunBurst di rappresentare in uno spazio relativamente ristretto l'organizzazione gerarchica di un documento, anche di grandi dimensioni, e il dettaglio offerto dalla visualizzazione ipertestuale.

1.3 DocuDipity “sul campo”: un esempio d’uso

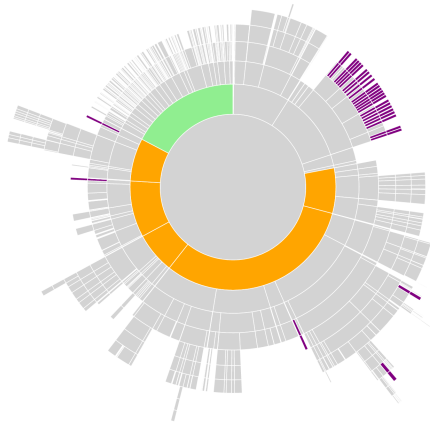
In questa sezione mostreremo le potenzialità offerte dal tool DocuDipity per analizzare documenti scientifici. Per tali analisi, sono stati utilizzati alcuni articoli scientifici presentati al convegno Balisage Markup Conference, che dal 2008 li mette a disposizione in formato XML. Le regole utilizzate dalle analisi sono:

- citations: evidenzia la posizione delle citazioni nel documento
- paragraph length: evidenzia le differenti lunghezze di un paragrafo, assegnando un colore per paragrafi lunghi, corti o medi

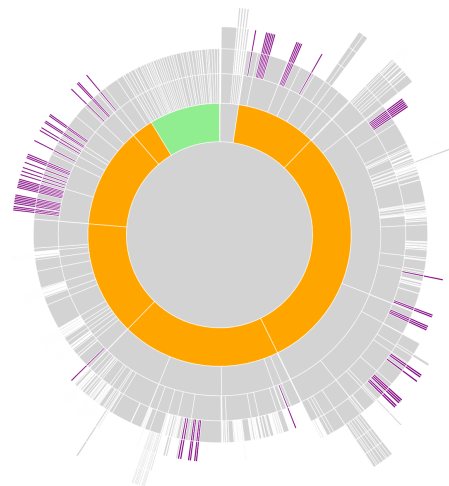
1.3.1 Posizione delle citazioni

A seconda della tipologia di articolo analizzato si può notare che la posizione dell’aggregazione delle citazioni può variare, non manifestarsi o essere diluita nell’articolo. L’utilizzo della visualizzazione SunBurst permette di evidenziare la disposizione delle citazioni nell’articolo in maniera molto più fruibile rispetto a una rappresentazione sequenziale offerta dalla visualizzazione ipertestuale. Utilizzando la regola CSS che evidenzia le citazioni nell’articolo (tag xref per i documenti XML di Balisage) permette di scorgere differenti strutture di documenti, dove per esempio:

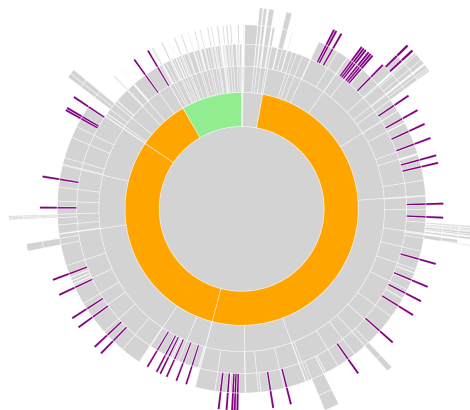
- se l’autore vuole introdurre al lettore il background necessario per comprendere l’argomento trattato nell’articolo, le citazioni tendono ad essere raggruppate nell’introduzione, come mostrato nella Figura 1.5a
- se l’autore, invece, vuole confrontare il proprio lavoro con articoli che trattano argomenti simili, le citazioni tendono ad essere raggruppate nella sezione dei lavori correlati, come mostrato nella Figura 1.5b
- infine, può capitare che la trattazione del contributo dell’autore sia profondamente collegata con la letteratura e questo causa la diluizione delle citazioni lungo tutto l’articolo, come mostrato nella Figura 1.5c



(a) Visualizzazione articolo con le citazioni (segmenti viola) raggruppate nell'introduzione



(b) Visualizzazione articolo con citazioni raggruppate nella sezione "related work"



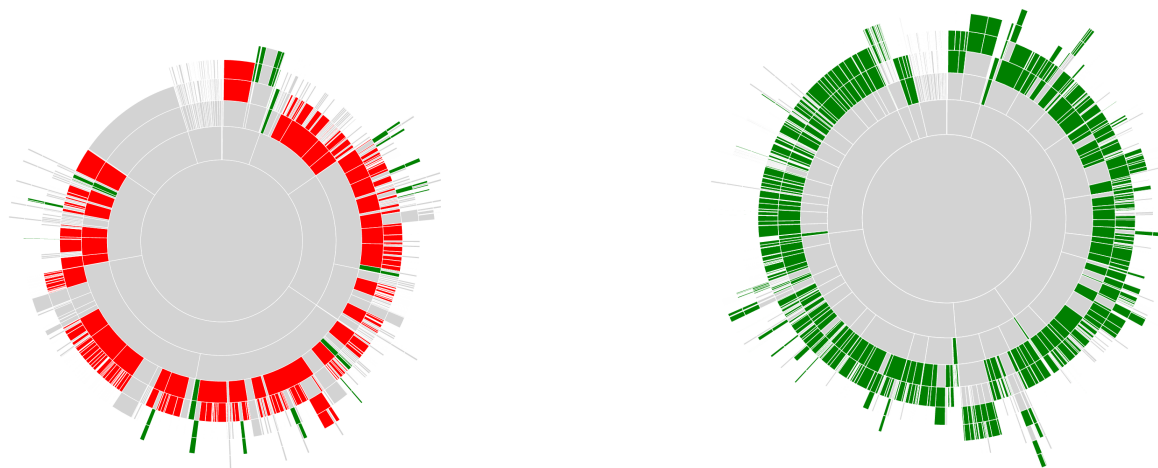
(c) Visualizzazione articolo con citazioni sparse

Figura 1.5: Rappresentazioni SunBurst che evidenziano differenti strutture di articoli scientifici

1.3.2 Stili di scrittura

Un'altra analisi che si potrebbe fare su un articolo è indagare sullo stile e sul linguaggio utilizzato da/dagli autori dell'articolo. L'idea alla base di questa analisi è che esistono differenti stili di scrittura che possono variare da autore ad autore, che vengono nascosti dalla visualizzazione dettagliata ipertestuale ma che con l'utilizzo della vista SunBurst si possono con maggior facilità scoprire ed evidenziare. Per esempio, considerando la lunghezza dei singoli paragrafi, utilizziamo la regola che colora di grigio i paragrafi con una lunghezza simile alla media e di rosso/verde i paragrafi con una lunghezza superiore/inferiore alla media. L'applicazione di questa regola, alla collezione di articoli precedentemente descritta, permette di evidenziare:

- nel caso di articoli con un solo autore lo stile adottato, chi predilige uno stile conciso e con paragrafi corti (come mostrato nella Figura 1.6b) e chi, invece, tende ad essere più prolisso e sviluppare il pensiero con frasi lunghe (come mostrato nella Figura 1.6a)
- nel caso di articoli con più autori di individuare le parti alle quali un autore ha contribuito, come mostrato nella Figura 1.7, in quanto emergono i differenti stili degli autori



(a) Articolo costituito principalmente da paragrafi lunghi (in rosso)

(b) Articolo costituito principalmente da paragrafi corti (in verde)

Figura 1.6: Rappresentazione stili autore

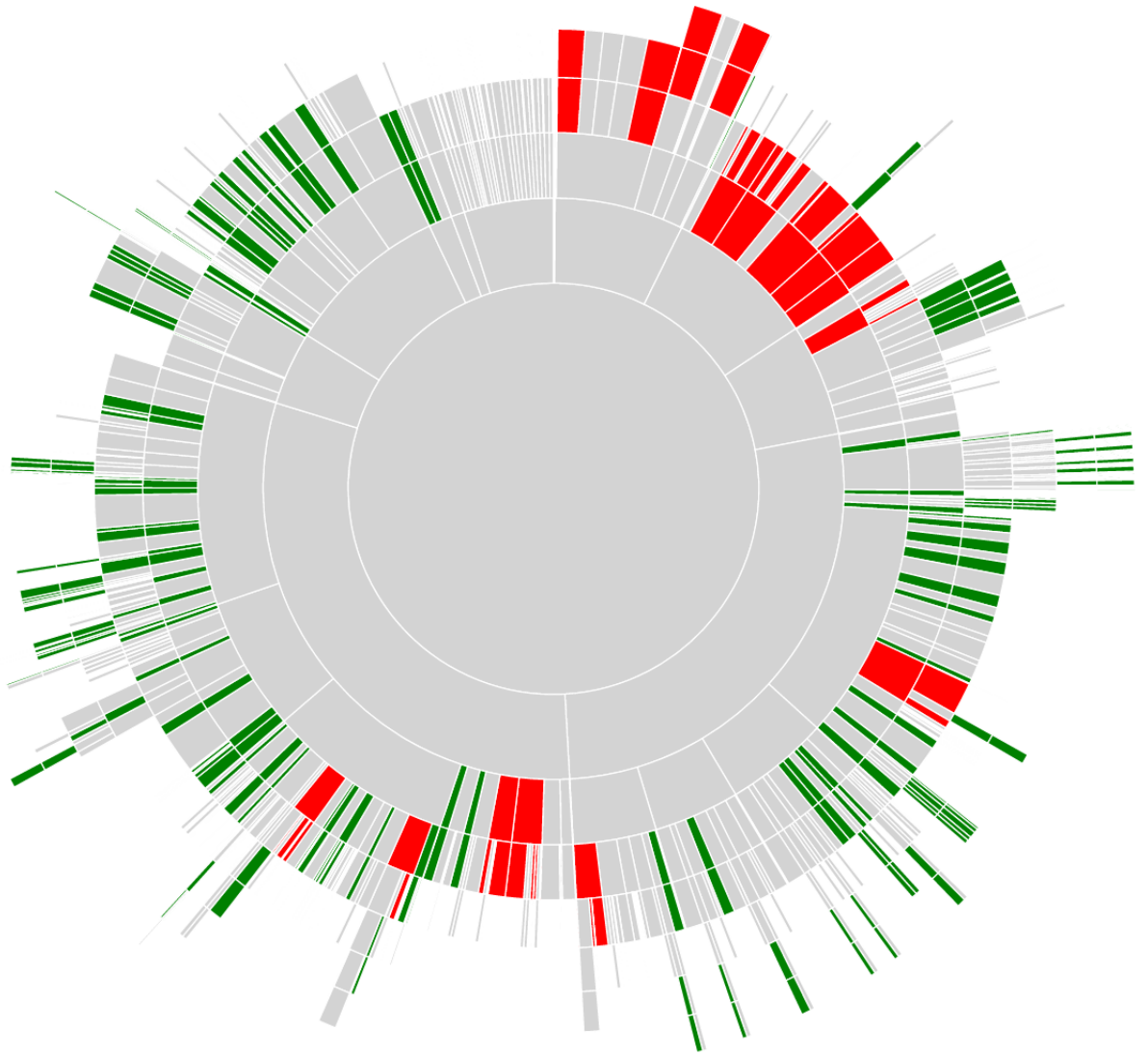


Figura 1.7: Articolo scritto da più autori con differenti lunghezze dei paragrafi

1.4 Limiti e criticità

Una tra le principali limitazioni presenti nella precedente interfaccia era l'interazione con un set documentale completamente fisso, non vi era infatti alcuna possibilità di aggiungere altri documenti oltre quelli precaricati, e nemmeno quella di creare sottoinsiemi di documenti su cui lavorare, di conseguenza non era neppure presente alcun meccanismo per filtrare, ordinare e ricercare articoli. Il sistema di navigazione degli articoli era molto rigido e limitato, l'utente poteva selezionare un solo articolo di una conferenza alla volta e aprirlo per avere una visualizzazione coordinata con due sole specifiche viste, non era prevista la possibilità di selezionare più articoli (magari appartenenti a set di articoli diversi), per poterli analizzare e metterli a confronto.

Altre importanti criticità erano invece relative all'interazione con le viste, era infatti presente uno schema molto rigido, non modulare, che limitava le funzionalità del sistema, non permettendo di aggiungere nuove viste oltre alle due per il quale il sistema precedente era stato concettualmente realizzato (Hypertext view e SunBurst). Non era inoltre esplicitata l'idea di confronto tra viste diverse relative allo stesso articolo o tra viste uguali relative ad articoli diversi, era sempre e solo possibile visualizzare un singolo documento alla volta con le due viste sopra citate. Anche la parte architettonica era caratterizzata da diverse criticità, non era infatti presente alcuna distinzione tra la parte front-end e la parte back-end, non vi era alcun supporto di API per interagire con l'interfaccia e non era prevista nessuna idea di riutilizzo di widgets, rendendo difficile il cross-browsing. Inoltre, DocuDipity risultava essere non ben ingegnerizzata, con un codice non manutenibile e poco documentato.

La progettazione della nuova interfaccia è stata realizzata tenendo conto di tutte le limitazioni e criticità descritte in questo capitolo, con l'obiettivo di ottenere un prodotto finale manutenibile, facilmente estendibile, rendendolo quindi indipendente da future viste che verranno implementate successivamente. Infine, il sistema di navigazione proposto in questo elaborato, risulta essere più flessibile rispetto al precedente, permette di gestire articoli appartenenti a set di documenti diversi, che attraverso un menù possono essere aggiunti a delle aree di lavoro nelle quali l'utente può eseguire diverse azioni, tra cui decidere di organizzare gli articoli applicando dei filtri, rimuoverli dall'area di lavoro o aprirli (anche più d'uno), per visualizzarli in una specifica vista, ed infine salvare la configurazione di lavoro creata per aprirla in un secondo momento. Obiettivi e requisiti della nuova versione dell'applicazione verranno analizzati più nel dettaglio nel capitolo successivo.

Capitolo 2

La progettazione delle API di DocuDipity

Dopo aver introdotto il progetto DocuDipity, verranno introdotte le principali fasi dello sviluppo dell'applicazione back-end. Nello specifico, iniziando dall'introduzione dei concetti di API (Application Programming Interface) e del modello architetturale REST (REpresentational State Transfer), con particolare attenzione su quest'ultimo in quanto risulta fondamentale per comprendere l'interazione dell'applicazione back-end con il front-end. Successivamente, si discuterà del modello dati sviluppato per formalizzare le risorse che l'applicazione dovrà gestire, con particolare enfasi sul modello FRBR utilizzato per catturare la possibilità di un articolo di avere molteplici rappresentazioni. Infine, si fornirà una descrizione dettagliata delle API prodotte, con qualche caso d'uso e commentando la struttura generale.

2.1 API e REST

Le sigle API e REST fanno riferimento a due concetti ben distinti. Il termine API si riferisce all'interfaccia che definisce la comunicazione tra il server (in generale può essere visto come l'erogatore di un servizio) e il client (in generale può essere visto come l'utilizzatore del servizio). Questo permette al client di utilizzare un generico servizio, senza essere a conoscenza dell'effettiva implementazione e basandosi unicamente sulla documentazione messa a disposizione dal server in cui si descrivono la struttura delle richieste e le relative risposte. L'utilizzo di questo concetto nello sviluppo di applicativi web si riferisce agli URI e ai relativi parametri (query string) per ottenere le informazioni memorizzate nel server ma, in generale, può anche essere utilizzato in altri contesti, come ad esempio le API per la manipolazione dei file messe a disposizione da UNIX. Se l'implementazione delle API rispetta i principi architetturali REST [11], che verranno approfonditi di seguito, allora è stata sviluppata un'applicazione API REST.

2.1.1 I principi REST

Il principio architetturale REST è stato sviluppato prendendo come riferimento il protocollo HTTP, con lo scopo di utilizzarne a pieno le potenzialità per sviluppare applicazioni web interattive. Nello specifico, i principi, ripresi dal protocollo HTTP, sono:

Client-Server: il server si occupa di manipolare i dati e di renderli disponibili alle richieste del client in maniera efficiente. Il client si occupa di reperire le informazioni e di visualizzarle all'utente finale. Questa separazione permette al client ed al server di evolvere in maniera indipendente

Stateless: ogni richiesta HTTP deve contenere tutte le informazioni necessarie per essere portata a termine, ovvero il server non utilizza informazioni ottenute con richieste precedenti

Cache: il client, il server e qualunque nodo intermedio può ricorrere all'utilizzo della cache per migliorare le performance e ottimizzare il traffico sulla rete

Uniform-Interface: specifica i vincoli da rispettare nella progettazione dell'interfaccia di comunicazione tra il client e il server, che sono quattro e verranno discussi nella sezione successiva

Layered-System: ogni componente vede e interagisce solamente con il componente adiacente. Quindi, un client che si connette ad un componente intermedio, come un proxy, non conosce i componenti che giacciono oltre. Questo permette ai componenti dell'architettura di essere indipendenti, estendibili e sostituibili

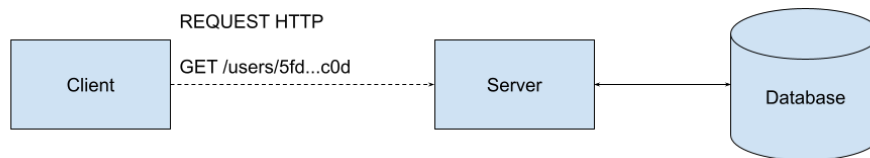
2.1.2 Approfondimento dei principi REST (Uniform Interface)

Per rispettare il vincolo dell'interfaccia uniforme (Uniform interface) è necessario seguire i quattro vincoli che definiscono l'interfaccia di comunicazione tra il client e il server, ovvero:

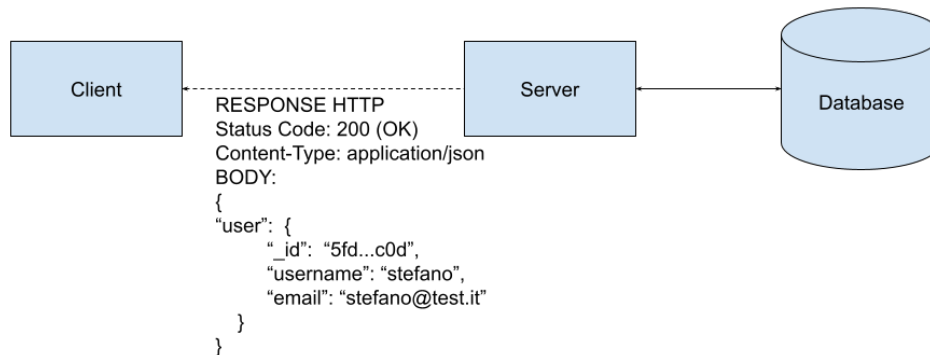
1. **Risorse e Identificativo di Risorse:** secondo lo standard REST ogni informazione/dato che può avere un nome è una risorsa, come ad esempio documenti/immagini. Ogni risorsa è associata ad un identificativo che prende il nome di URI (Uniform Resource Identifier [2]). Un esempio di risorsa è un utente registrato, il quale ha un URI del tipo `api/users/userId` dove l'`userId` viene generato dal server al momento della registrazione. Infine, un insieme di risorse è detto collezione, cioè l'insieme degli utenti registrati, ottenibili con l'URI `/api/users/`

2. **Rappresentazione di Risorse:** è ottenuta dalla concatenazione dei byte della risorsa stessa e dei metadati che la descrivono
3. Le richieste e risposte in ambiente REST devono essere **autoesplicative**, ovvero devono sempre contenere il relativo media-type (content-type nel caso di HTTP) per descrivere il formato utilizzato o richiesto
4. **HATEOAS (Hypermedia As The Engine Of Application State):** questo principio prevede che, a seguito di una richiesta, l'oggetto restituito dal server non solo contenga i dati della risorsa ma anche le azioni che si possono effettuare per cambiarne lo stato. In questo modo, il client si comporta in maniera simile a un browser ovvero l'unico scopo che ha è la presentazione dei dati

Nelle Figure 2.1a e 2.1b viene mostrato un esempio di comunicazione tra il client e il server.



(a) Richiesta HTTP per ottenere una rappresentazione della risorsa user



(b) Risposta del server, contenente la risorsa richiesta

Figura 2.1: Esempio di comunicazione client-server

2.1.3 I metodi HTTP

I metodi HTTP devono essere utilizzati per esplicitare l'azione da eseguire su una risorsa identificata da URI, nello specifico i metodi messi a disposizione dallo standard HTTP permettono di coprire il ciclo di vita di una risorsa (CRUD). Il ciclo di vita di una risorsa inizia nel momento della sua creazione (Create), successivamente è sottoposta a

delle interrogazioni (Read) e/o aggiornamenti (Update) e termina con la sua rimozione (Delete). Tuttavia, non esiste un mapping uno-a-uno tra le operazioni CRUD e i metodi HTTP poichè spesso, per determinare il giusto metodo, è necessario approfondire le specifiche del metodo HTTP per evitare errori. Nella seguente tabella si riportano alcuni esempi di utilizzo di metodi HTTP e delle operazioni CRUD associate.

Metodo	CRUD	Commento	Esempio
PUT/POST	CREATE	se presente l'identificativo della risorsa si utilizza il verbo PUT, altrimenti POST	POST /api/users/
GET	READ	operazioni che non modificano lo stato interno del server	GET /api/users/userId
PUT/POST	UPDATE	nel caso si sostituisca la risorsa si utilizza il verbo PUT, altrimenti se si aggiorna solo in parte la risorsa il verbo POST	POST /api/documentSets/id
DELETE	DELETE	eliminazione risorsa	DELETE /api/documentSets/id

Come è evidente dalla tabella, i metodi DELETE e GET sono di facile comprensione e difficilmente si prestano a interpretazioni errate, diverso il discorso della POST. Infatti, nel [10] è stata aggiornata la definizione come segue:

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics.

Ovvero, il metodo POST può essere utilizzato in numerosi casi con un significato deciso localmente purchè questo non sovrascriva quello di altri verbi HTTP. Nel nostro caso, POST è stato utilizzato per aggiornare parzialmente le risorse, ciò non va in conflitto con i metodi:

- PUT: in quanto, richiede che la risorsa sia totalmente sostituita dalla rappresentazione presente nel body della richiesta [10]
- PATCH: in quanto, richiede che si descrivano le operazioni per aggiornare la risorsa [8]

2.1.4 Livelli di maturità di un'applicazione REST

Precedentemente sono stati introdotti i vincoli che l'applicazione deve seguire per potersi definire REST anche se, in generale, la distinzione non si limita ad applicazioni REST o non REST. Infatti, in base ai principi che l'applicazione segue, può porsi ad un differente livello di maturità rispetto a tale stile di architettura [16, 31].

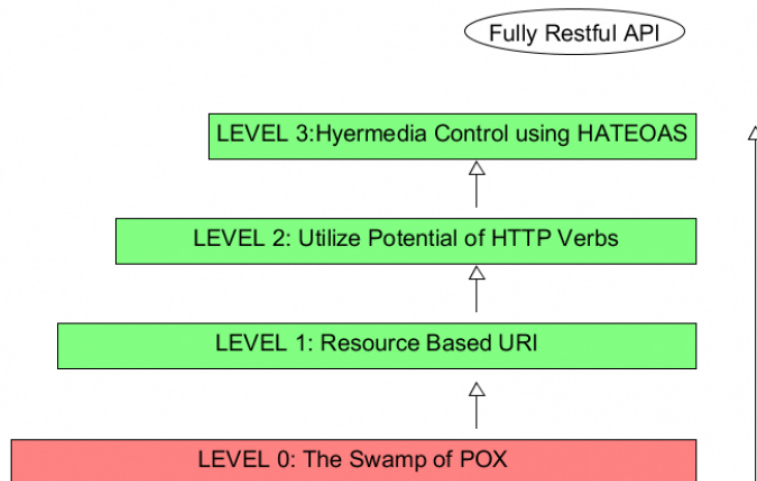


Figura 2.2: Overview dei possibili livelli di maturità di un'applicazione, immagine disponibile nel blog redhat all'indirizzo <https://developers.redhat.com/sites/default/files/blog/2017/09/model-1.png>

Come mostrato nella Figura 2.2 esistono quattro possibili livelli:

1. **Level 0 The Swamp of POX:** applicativi non REST, ovvero che non utilizzano gli URI per identificare le risorse, non utilizzano nè i metodi HTTP e nè il principio HATEOAS
2. **Level 1 Resource Based Address/URI:** in questo caso l'applicazione utilizza URI differenti per identificare risorse diverse ma continua a non utilizzare le potenzialità dei metodi HTTP per distinguere le richieste e continua a non utilizzare il principio HATEOAS
3. **Level 2 Utilize Potential of HTTP:** l'applicazione non solo utilizza gli URI per identificare le risorse ma utilizza i metodi HTTP per distinguere l'operazione che esegue su di esse
4. **Level 3 Use Hypermedia (or HATEOAS):** ultimo livello di maturità, l'applicazione oltre a ritornare la risorsa richiesta aggiunge anche gli URI che specificano le azioni che si possono fare per cambiarne lo stato oltre a seguire i principi del livello precedente

La nostra applicazione, come apparirà chiaro dalla prossima sezione quando si discuterà la documentazione delle API, si colloca al secondo livello di maturità in quanto utilizza gli URI per identificare le risorse ed i metodi HTTP per specificare l'azione da intraprendere.

2.2 Il modello dati e risorse

Come riportato nel capitolo di introduzione del progetto, DocuDipity è un web tool che permette la fruizione di articoli scientifici utilizzando diverse visualizzazioni (SunBurst e HyperText). Tuttavia, la peculiarità di DocuDipity è quella di offrire all'utente una fruizione sincronizzata di uno stesso documento con visualizzazioni differenti. Questa particolarità introduce due problemi fondamentali:

1. sviluppare un meccanismo che consenta all'articolo di avere molteplici rappresentazioni
2. sviluppare un meccanismo per poter supportare in maniera dinamica nuove visualizzazioni

Il primo problema deriva dal fatto che uno stesso articolo può esistere in rappresentazioni differenti, come ad esempio la rappresentazione per la visualizzazione ipertestuale piuttosto che la visualizzazione in SunBurst. Per catturare questa peculiarità è stato utilizzato il modello FRBR [40], il quale permette di separare le molteplici rappresentazioni dell'articolo dai metadati dello stesso.

2.2.1 FRBR

Il Functional Requirements for Bibliographic Record (FRBR) è uno schema entità-relazione sviluppato dall'International Federation of Library Associations and Institutions (IFLA) con l'obiettivo di definire una struttura generale per la gestione di dati bibliografici secondo le necessità degli utenti. Nello specifico individua quattro entità: **work**, **expression**, **manifestation** ed **item**.

Il termine **work** si riferisce ad un'opera o ad una creazione artistica nella forma astratta e può essere vista come l'idea dell'opera che l'autore ha nella sua mente.

L'**expression**, pur essendo sempre un'entità astratta, fa riferimento alla realizzazione intellettuale o artistica di un'opera in una notazione alfanumerica (la brutta utilizzata per produrre un articolo scientifico).

La **manifestation** rappresenta la materializzazione fisica dell'opera intellettuale, ad esempio la pubblicazione di un articolo scientifico.

Infine, l'**item** è il singolo esemplare di una manifestation, come la versione cartacea dell'articolo precedentemente pubblicato. Per il nostro modello dati sono stati ripresi i concetti di:

- **expression**: individua l'insieme dei metadati di un articolo, ad esempio il titolo, la descrizione, l'anno di pubblicazione, ecc.
- **manifestation**: identifica le possibili rappresentazioni dell'articolo, ad esempio la rappresentazione ipertestuale e SunBurst

2.2.2 Modello Entità-Relazione

Con il modello FRBR sono stati catturati i concetti fondamentali che l'applicazione dovrà gestire, tuttavia, per rappresentare le altre risorse necessarie per il funzionamento dell'applicazione è stato utilizzato il modello E-R. Il modello entità-relazione (abbreviato con la sigla E-R) è una rappresentazione grafica che permette, attraverso l'utilizzo di certi costrutti, di visualizzare le informazioni presenti nel database in maniera indipendente dal metodo di organizzazione adottato.

Il modello E-R si basa sul concetto di entità, relazione e attributo. Le entità possono essere viste come l'insieme degli oggetti rappresentabili, i quali sono composti da certe proprietà dette attributi.

Le relazioni rappresentano i legami logici tra le diverse entità, significativi per il funzionamento dell'applicazione.

La definizione del modello E-R è effettuato partendo da una descrizione del funzionamento dell'applicazione in linguaggio naturale, nel quale si accentua la descrizione del comportamento desiderato dall'applicazione. L'estrazione dei concetti necessari alla costruzione di tale modello, è stata resa possibile grazie al metodo di [18] il quale suggerisce di investigare la tipologia delle parole presenti nel testo per derivare il modello E-R. Infatti, le entità spesso sono espresse mediante l'uso di nomi, aggettivi per descriverne le proprietà di interesse e verbi per descriverne le relazioni. Nella figura 2.3 viene riportato il modello E-R dell'applicazione, si procederà con la sua descrizione.

I **DocumentSet** sono degli insiemi di articoli (**expression**) che condividono alcune proprietà (come ad esempio l'anno di pubblicazione o la conferenza a cui sono stati presentati). Ogni utente registrato può inserire dei nuovi DocumentSet, ma ogni DocumentSet appartiene solo all'utente che lo ha caricato e che, quindi, può apportarvi modifiche. Gli utenti non registrati o non proprietari di un DocumentSet possono solo visualizzarlo ma non apportarvi modifiche.

I DocumentSet sono composti dalle **expression**, che contengono i metadati dell'articolo (autore, anno di pubblicazione, ecc.) e dalle **manifestation** che contengono le rappresentazioni di un articolo secondo un certo Schema.

La cardinalità della relazione tra le entità **expression** e **manifestation** è *one-to-many*, ovvero ogni manifestation appartiene solo ad una expression mentre una expression può avere diverse manifestation (in virtù dei molteplici tipi di visualizzazioni supportate).

Da notare che, per caricare una nuova manifestation, è necessario che esista l'expression a cui fa riferimento.

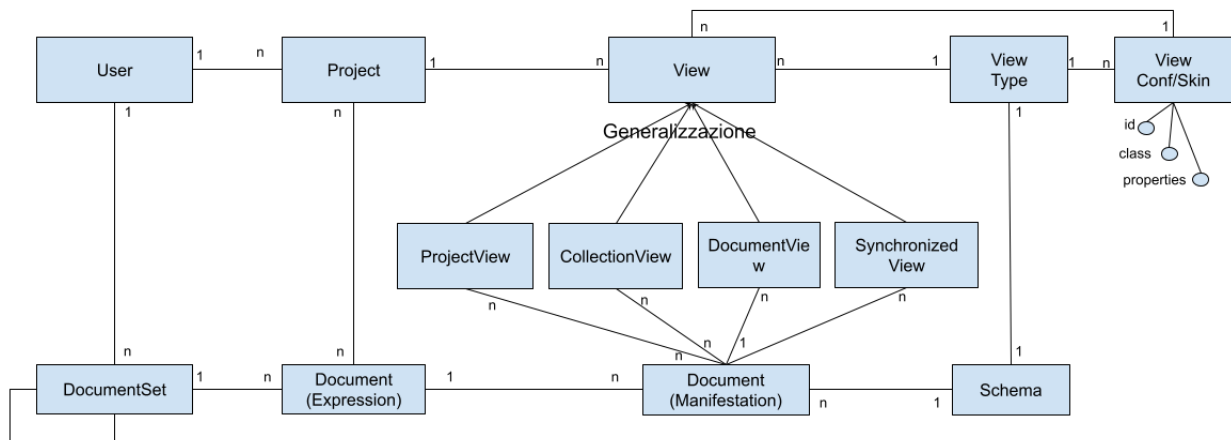


Figura 2.3: Overview del modello E-R dell'applicazione

Ogni istanza dell'articolo (manifestation) per poter essere correttamente interpretata deve seguire delle regole che sono raccolte nell'entità **Schema**. L'associazione adottata è quella *one-to-one* in quanto l'insieme delle regole viene utilizzato solo da una visualizzazione e visualizzazioni differenti non possono avere regole condivise.

La **ViewType** contiene le informazioni di una View, come ad esempio il nome e una breve descrizione. Quindi, ogni View è associata ad una ViewType mentre una ViewType può essere utilizzata da più View, quindi la relazione utilizza la cardinalità *one-to-many*. Inoltre, una ViewType è associata univocamente ad uno Schema con l'obiettivo di poter risalire, per ogni manifestation caricata nel sistema, al tipo di View a cui la manifestation fa riferimento.

Ogni utente può creare dei **Project** che sono un insieme di expression su cui condurre l'analisi. Come detto precedentemente, gli utenti non registrati possono solo visualizzare i progetti di altri e gli utenti registrati possono modificare solamente i progetti da loro creati.

Ogni project è composto da un sottoinsieme (non stretto) di expression su cui si possono applicare differenti View. L'entità **view** contiene i metadati necessari per il ripristino dello stato dell'interfaccia nel front-end. Inoltre, una view ha quattro specializzazioni:

1. **ProjectView**: si applica una ViewType (SunBurst o Hypertext) a tutte le expression del progetto su cui si sta svolgendo l'analisi
2. **CollectionView**: applicazione di una ViewType ad un sottoinsieme di expression del progetto su cui si sta svolgendo l'analisi
3. **DocumentView**: applicazione di una ViewType ad un singolo documento (expression) del progetto su cui si sta svolgendo l'analisi

4. **SynchronizedView**: confronto di due ViewType applicate ad uno stesso documento (expression)

Infine, l'entità **ViewSkin** viene utilizzata come contenitore di regole CSS e JS per la personalizzazione dello stile. Da notare la possibilità di utilizzare una stessa regola in view differenti, anche se non c'è la possibilità di applicare diverse regole alla stessa view contemporaneamente (one-to-many).

2.3 Documentazione delle API di DocuDipity

Una volta ottenuto il modello dati, inizia la fase di progettazione delle API REST, in cui si definiscono le operazioni da implementare per ciascuna risorsa (entità) individuata. Per documentare le API implementate dal back-end è stato utilizzato Swagger [36], in questo modo il front-end può accedere ad una documentazione completa su come invocare correttamente le API.

2.3.1 Swagger

Per comprendere Swagger bisogna prima introdurre il concetto di OpenAPI. OpenAPI è un formato che permette di descrivere le API REST in modo comprensibile per i computer. Nello specifico, permettono di descrivere le API messe a disposizione dall'applicazione esplicitando il path delle richieste, le query string per filtrare i risultati, i token necessari per l'autorizzazione, le possibili risposte specificando i messaggi di risposta con i relativi status code. Swagger è un insieme di tool costruiti intorno al concetto di OpenAPI che permettono di definire, documentare e testare le API. I tre tool principali sono:

1. **SwaggerEditor**: permette di descrivere le API secondo le specifiche di OpenAPI
2. **SwaggerUI**: costruisce la documentazione delle API partendo dalle specifiche OpenAPI scritte
3. **Swagger Codegen**: genera il client e il server dalle specifiche OpenAPI scritte

2.3.2 Organizzazione delle API e struttura base

Prima di presentare nel dettaglio le chiamate alle API, si procede con una descrizione ad alto livello della struttura della documentazione. Innanzitutto, le API sono state raggruppate in base alla risorsa di appartenenza e, come risulta dalla Figura 2.4, sono presenti otto gruppi di risorse. Tutte le API appartenenti ad un certo gruppo possono essere ricondotte al modello CRUD, precedentemente introdotto. Tuttavia, le

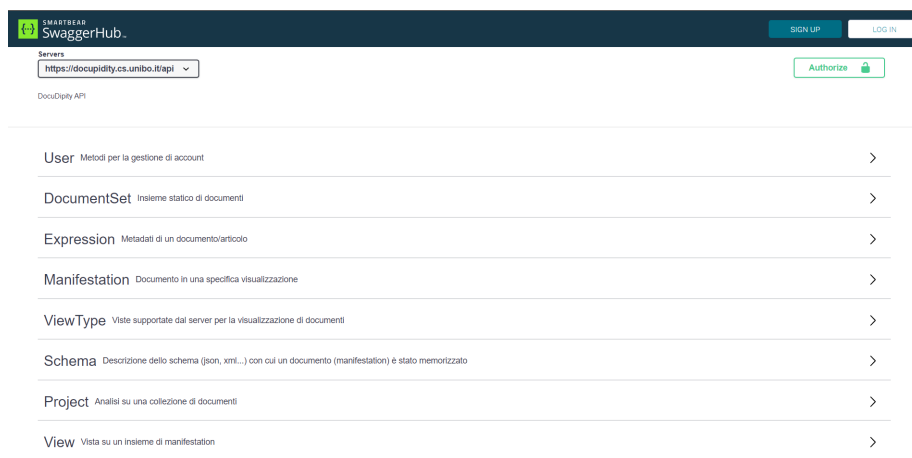


Figura 2.4: Overview del raggruppamento delle risorse

risorse Schema e ViewType espongono solo le API di lettura e tale limitazione è dovuta al fatto che solamente gli utenti amministratori possono modificare tali risorse. In questa versione non è stata implementata la predetta funzionalità, poichè rimandata a futuri sviluppi. La discussione procede considerando un esempio di API della collezione documentSet, riportata nella Figura 2.5. Le API che non modificano lo stato interno del server (metodo GET) sono invocabili da un qualunque utente (anche non autenticato) e nella documentazione è possibile distinguerle grazie alla presenza di un lucchetto posizionato alla destra della descrizione. Invece, per le API che modificano lo stato interno del server viene richiesto il token di autorizzazione rilasciato dal server al momento dell'autenticazione email/password. Un altro dettaglio che si può notare sempre dalla Figura 2.5, è l'uso del carattere '/' per distinguere le API che operano su una collezione di risorse (che terminano con tale carattere) dalle API che operano su una singola risorsa (che terminano con l'identificativo della risorsa stessa).

Cliccando su una delle chiamate API, è possibile usufruire di una documentazione più

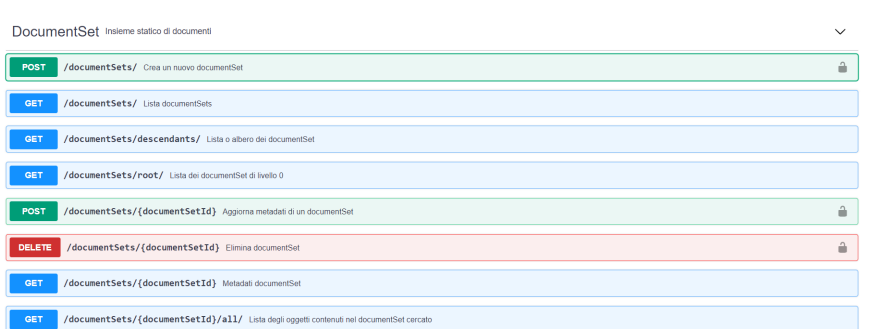
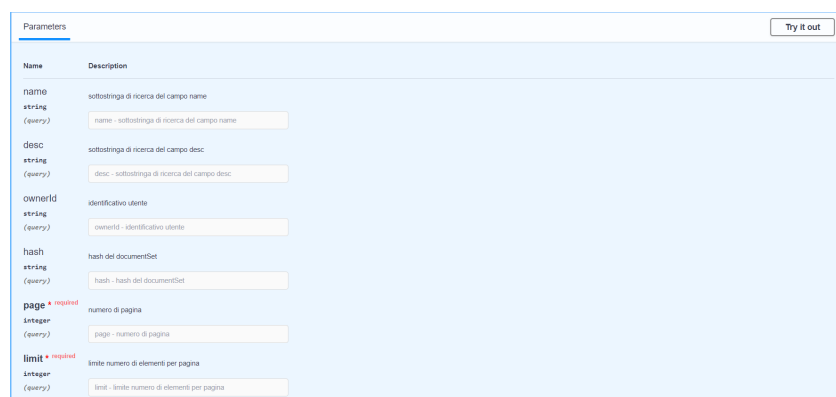


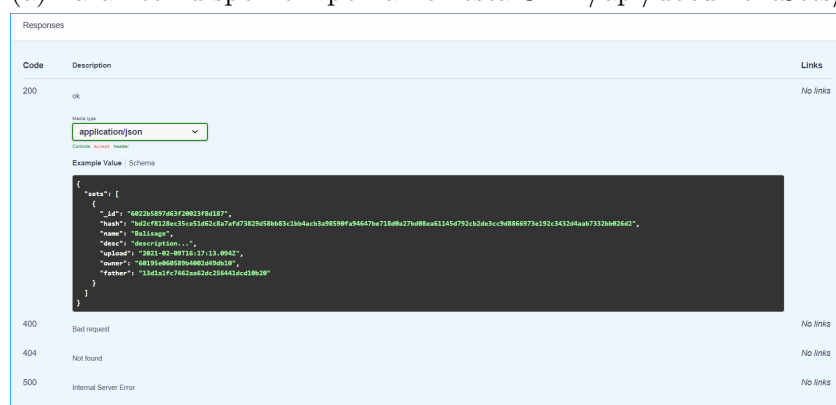
Figura 2.5: API disponibili per la collezione documentSets

estesa che permette di specificare la struttura delle richieste e risposte HTTP, tra cui:

- **Parameters:** permette di specificare le query string accettate dall'API per filtrare o modificare la risposta del server, nella Figura 2.6a è riportato un esempio con i parametri dell'API GET /documentSets/
- **Body (se il metodo HTTP contiene tale campo):** permette di specificare la struttura del body, utile per documentare i campi necessari per la creazione della risorsa o per un suo aggiornamento
- **Response:** documenta le possibili risposte del server, specifica lo status code ritornato e il relativo messaggio, nella Figura 2.6b è riportato un esempio con le risposte possibili dell'API GET /documentSets/



(a) Parametri disponibili per la richiesta GET /api/documentSets/



(b) Risposte possibili per la richiesta GET /api/documentSets/

Figura 2.6: Esempio di documentazione Swagger per API

Infine, dopo aver esposto la struttura della documentazione, si riporta un esempio di utilizzo di tali API, nel quale l'applicazione back-end è stata eseguita in locale (con

l'insieme degli articoli presentati alla conferenza Balisage) e la libreria curl [39] per generare le richieste.

Supponiamo che l'utente voglia cercare tutti gli articoli che contengono nel titolo la parola "context", di seguito è riportato il comando per generare tale richiesta.

```
curl --get \  
"http://localhost/api/expressions/?title=context&page=0&limit=5"
```

Eseguendo tale comando, otteniamo la risposta contenente una lista con tutti gli articoli che presentano nel titolo la parola cercata.

```
{  
  "expressions": [  
    {  
      "_id": "6057108358a0fa00211d556b",  
      "title": "Markup Formats In Context",  
      "desc": "A comparison of the strengths of some widely-  
        used markup systems ",  
      "owner": "602a95d29c644700200475f5",  
      "upload": "2021-03-21T09:23:15.781Z",  
      "lastModified": "2021-03-21T09:23:15.781Z",  
      "documentSet": "6057108058a0fa00211d553e",  
      "authors": "Liam R. E. Quin ",  
      "year": 2014,  
      "__v": 0  
    }  
  ]  
}
```

Dalla risposta, inoltre, possiamo risalire all'identificativo dell'articolo (`_id`) ed ottenere la lista di tutte le rappresentazioni disponibili nel sistema. Nella richiesta che segue, si limita il numero di oggetti ritornati ad uno, per contenere il numero di righe presenti nella risposta.

```
curl --get \  
"http://localhost/api/expressions/6057108358a0fa00211d556b/  
  manifestations/?page=0&limit=1"
```

Dalla risposta che segue, è possibile ricavare informazioni riguardanti l'utente che ha caricato la risorsa, oltre all'espression e allo schema a cui la manifestation fa riferimento.

```
{  
  "manifestations": [  
    {  
      "_id": "6057108c58a0fa00211d555eb",
```

```

    "expression": "6057108358a0fa00211d556b",
    "schemaRef": {
      "_id": "60228f6a483b5f04449c4701",
      "desc": "SunBurst",
      "type": "JSON",
      "ext": ".json",
      "rule": "test....",
      "contentType": "application/json"
    },
    "owner": "602a95d29c644700200475f5",
    "upload": "2021-03-21T09:23:24.681Z",
    "lastModified": "2021-03-21T09:23:24.681Z",
    "__v": 0
  }
]
}

```

In questa sezione è stata presentata la documentazione delle API e alcuni esempi di utilizzo delle stesse. Da notare che, avendo a disposizione la documentazione di Swagger, non è necessario riportare l'elenco completo dei casi d'utilizzo poichè la predetta documentazione contiene tutte le informazioni necessarie per la preparazione della richiesta da effettuare.

Capitolo 3

Implementazione delle API e servizi server-side

Dopo aver fornito una panoramica d'alto livello del lavoro effettuato per produrre le API e la relativa documentazione, in questo capitolo si approfondirà la fase di implementazione delle API. L'applicazione è composta da diversi servizi che, per poter essere correttamente eseguiti, necessitano di un ambiente preconfigurato. Tale configurazione è possibile eseguirla a mano ma causando numerose inefficienze, in quanto è legata al singolo computer che si sta utilizzando e deve essere ripetuta per ogni nuovo computer utilizzato. Per dare la possibilità al front-end di eseguire l'applicazione in autonomia è stato utilizzato Docker [7], il quale permette di condividere un ambiente preconfigurato in cui verrà eseguita l'applicazione senza necessità di interventi manuali. Tale soluzione ha avuto come beneficio di accelerare la fase del deploy sul server dipartimentale, in quanto il lavoro aggiuntivo si è limitato alla configurazione di Docker. Il capitolo inizia con l'introduzione delle tecnologie utilizzate per lo sviluppo dell'applicazione, per poi continuare con la descrizione di Docker e terminare con l'illustrazione dell'architettura dell'applicazione.

3.1 Le tecnologie utilizzate

Gli argomenti approfonditi in questo paragrafo riguardano:

- il linguaggio utilizzato per lo sviluppo dell'applicazione e i principali framework utilizzati per agevolare lo sviluppo
- il database utilizzato per memorizzare i dati, esponendo prima le principali tipologie e infine spiegando le motivazioni che hanno portato a scegliere MongoDB come database dell'applicazione

- la tecnologia dei token utilizzata per autorizzare le richieste HTTP che modificano lo stato interno del server

3.1.1 Node.JS e Express

Node.js [14] è un runtime system (software che fornisce i servizi necessari per l'esecuzione del programma, pur non facendo parte del SO) che permette l'esecuzione di codice javascript per la scrittura di programmi lato server. Il principale punto di forza di Node.js è la struttura asincrona orientata agli eventi, che permette di semplificare la scrittura di applicazioni che devono gestire numerose chiamate concorrenti. Per comprendere meglio questo concetto è utile capire la differenza tra Blocking e non Blocking in Node.js.

```

1 const fs = require('fs')
2 ...
3 const file = fs.readFileSync('readme.md')

```

Listing 3.1: Esempio di lettura sincrona

Nella Figura 3.1 viene riportato un esempio di un'operazione che blocca l'esecuzione del processo Node.js fino a quando l'operazione di lettura del file è terminata. Da notare che, in caso di errore, causa la terminazione dell'esecuzione dell'applicazione in quanto l'eccezione non viene gestita.

```

1 const fs = require('fs')
2 ...
3 fs.readFile('readme.md', (err, data) => {
4     if(err){
5         console.log(err)
6         return
7     }
8     return doStuff(data)
9 })

```

Listing 3.2: Esempio di lettura asincrona

Nell'esempio riportato in Figura 3.2, la lettura del file non blocca l'esecuzione di codice JavaScript, quando l'operazione è portata a termine verrà invocata una callback con i dati del file o le informazioni relative all'errore incontrato.

Tuttavia, per semplificare lo sviluppo di applicazioni web è stato utilizzato, insieme a Node.js, il framework Express [12], che permette di semplificare e velocizzare lo sviluppo di applicativi web. In particolare, i principali vantaggi del suo utilizzo sono:

- gestire le route dell'applicazione basata su HTTP, specificando metodo e URL

- personalizzazione dell'error handler delle route
- utilizzo di middleware (personali e non) per aggiungere funzionalità nella gestione delle richieste e risposte, come ad esempio l'autenticazione mediante il token, la validazione dell'oggetto richiesta, ecc.

3.1.2 Persistenza e Database

I database relazionali sono basati sul concetto di modello relazionale.

In base al predetto modello, i dati gestiti vengono organizzati secondo tabelle, in cui le righe (chiamate anche tuple) rappresentano un singolo strutturato elemento della tabella, mentre le colonne rappresentano un insieme di valori in uno specifico formato. Ogni riga possiede un insieme minimale di colonne che viene utilizzato per identificare univocamente ogni riga della tabella, detto Primary Key (PK).

Se una PK viene utilizzata in un'altra tabella per creare una relazione tra esse, questa prende il nome di Foreign Key.

I database non relazionali offrono diversi modelli di dati (come ad esempio key-value o document-oriented) che permettono la gestione di dati semi (o non) strutturati. Le principali categorie sono:

- **Key-value store:** il modello di dati adottato è quello di un array associativo (anche chiamato dizionario), in cui i dati vengono rappresentati come una collezione di chiave-valore. Ogni chiave è unica nel database
- **Document Store:** il predetto modello si concentra sulla gestione di documenti. Ogni documento è caratterizzato da un certo insieme di proprietà ed è identificato univocamente da una chiave. Inoltre, un'importante caratteristica di questa tipologia è l'esposizione di un linguaggio per l'interrogazione del database in base al contenuto dei documenti. Da notare, la differenza tra i concetti di collezione e documento e i concetti di tabella e riga del modello relazionale, consiste nella possibilità di avere documenti con strutture differenti appartenenti ad una stessa collezione, cosa non permessa nel modello relazionale

I vantaggi dei database non relazionali si concentrano sulle migliori performance per l'interrogazione di grandi quantità di dati e la flessibilità del modello dati (la possibilità di espanderlo in quanto non è fisso).

La scelta è ricaduta sul modello non relazionale in quanto l'applicazione dovrà gestire documenti di cui non sarà sempre possibile predeterminare la struttura e per le migliori performance di interrogazione anche in presenza di grandi quantità di dati. Infine, visto che l'applicazione gestirà principalmente oggetti di tipo JSON, è parso naturale scegliere

un modello basato su Document Store come MongoDB [29].

Da notare che il modello E-R, precedentemente discusso, è stato utilizzato solo come formalismo grafico per descrivere le risorse dell'applicazione e che, per le ragioni sopra elencate, è stato adottato un database non relazionale (MongoDB).

3.1.3 Autorizzazione mediante JWT (JSON Web Tokens)

Le richieste che modificano lo stato interno del server necessitano di un token di autorizzazione, che viene rilasciato dal server dopo la fase di autenticazione mediante email/password.

Inoltre, il token deve essere incluso nell'header Authorization, utilizzato in generale per autenticare le richieste. L'header presenta la seguente struttura

Authorization: <tipo> <credenziali>

Dove:

- **<tipo>**: corrisponde alla tipologia di autenticazione (basic, token, ecc.)
- **<credenziali>**: contiene in generale la stringa necessaria per autenticarsi

Nel caso di autorizzazione mediante token [22], il campo <tipo> conterrà la stringa 'Bearer' e il campo <credenziali> il token precedentemente ottenuto.

Una volta ricevuta la richiesta, il server si occuperà della validazione del token per verificare che non sia scaduto e che non ci siano state modifiche al payload. Un'importante osservazione è che il token non garantisce la confidenzialità, ovvero le informazioni contenute sono visibili a chiunque entri in possesso del token ma queste non possono essere modificate in quanto sono firmate dal server. Nello specifico, il token utilizzato per lo sviluppo di questa applicazione è JWT, basato sullo standard [21]. La struttura del token è composta da tre parti separate da un punto: un header, un payload e la firma (signature).

I tre campi che compongono il token hanno il seguente significato (nella Figura 3.1 è riportato un esempio di token JWT):

1. **Header**: contiene principalmente due campi, uno il tipo di token (JWT) e l'altro il tipo di algoritmo utilizzato per firmare il token
2. **Payload**: contiene i claims del token, che consistono in affermazioni sul token e in generale sull'entità

3. **Signature:** utilizzata per verificare che le informazioni contenute nei due precedenti campi non siano state modificate durante la trasmissione. La firma è ottenuta utilizzando l'algoritmo di hash, specificato nell'header, applicato alla stringa composta dai due campi precedenti

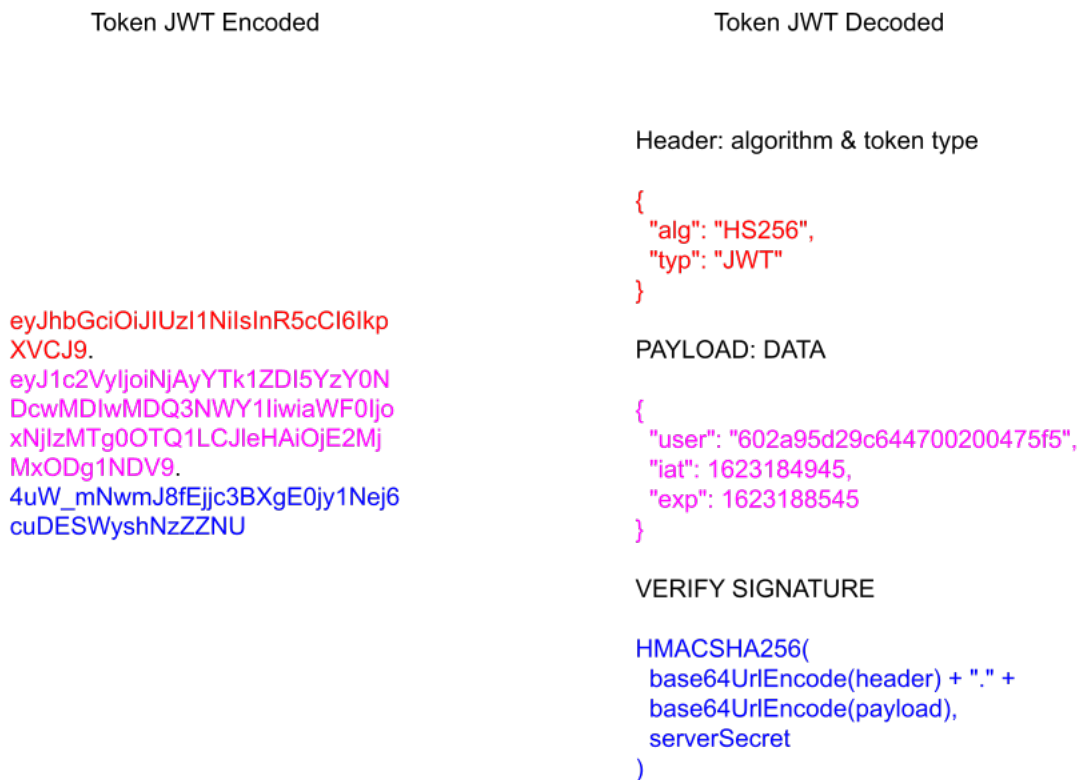


Figura 3.1: Esempio di token generato dall'applicazione per un certo utente

3.2 Docker e Docker-Compose

Docker è una piattaforma che permette di separare l'applicazione dall'infrastruttura locale, ovvero di impostare l'ambiente necessario per l'esecuzione dell'applicazione e di semplificare la condivisione. Docker è basato sull'architettura client/server mostrata dalla Figura 3.2, in cui il client invia al server (Docker daemon) i comandi inseriti da

utente e il server si occupa di eseguire le richieste ricevute e di gestire gli oggetti Docker. I principali oggetti Docker sono:

- **Image:** l'immagine contiene l'insieme di istruzioni necessarie per creare il container. In generale, un'immagine può essere costruita in locale utilizzando il Dockerfile che contiene le necessarie istruzioni (importazione sorgenti, installazione dipendenze, ecc.), oppure utilizzare un'immagine creata da altri utenti e pubblicata su un registro pubblico (come ad esempio l'immagine di un database o di un framework)
- **Container:** quando l'immagine viene eseguita, prende il nome di Container. Un container ha vari stati (created, running, exited, ecc.) che possono essere controllati attraverso i comandi del docker client

Docker-Compose è un tool che consente di definire ed eseguire applicativi che utilizzano più container, attraverso un file di tipo YAML. Nello specifico, invece di gestire tutti i container necessari individualmente, permette di accentrare la loro configurazione ed esecuzione.

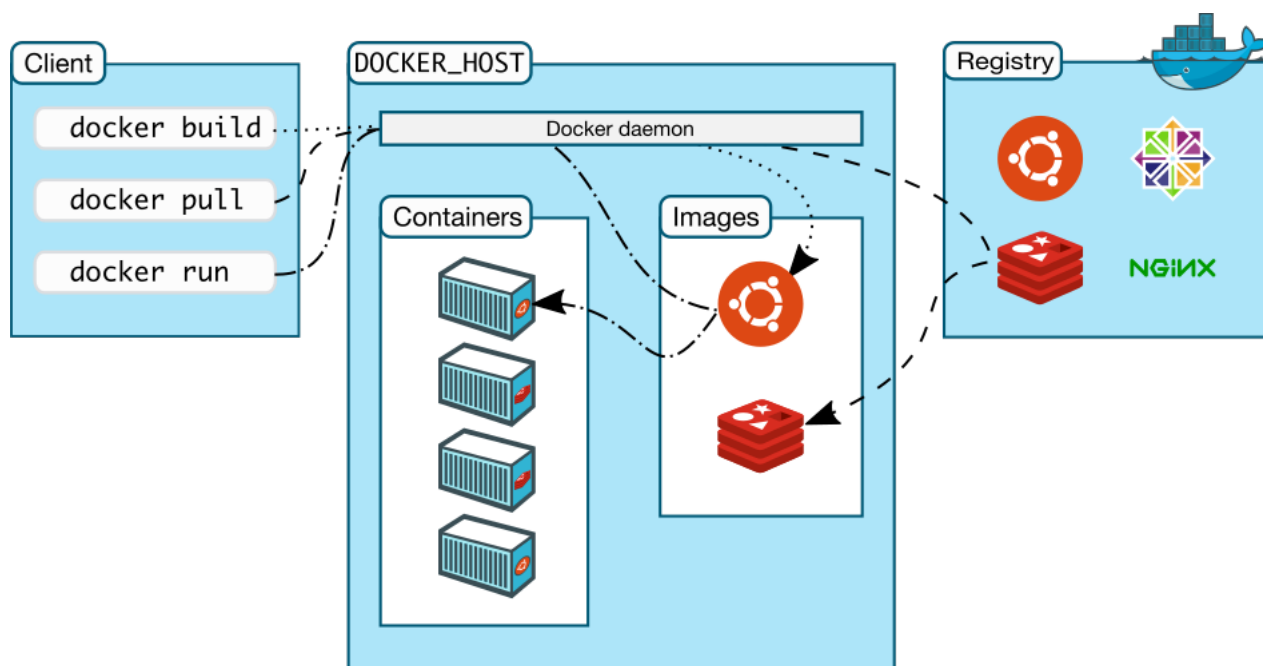


Figura 3.2: Architettura di Docker, immagine disponibile sul sito di Docker all'indirizzo <https://docs.docker.com/engine/images/architecture.svg>

3.2.1 Persistenza dei dati in Docker

I file creati dall'esecuzione dell'applicazione sono limitati al tempo di vita del container e quindi non sono persistenti. Docker, per consentire la persistenza dei dati mette a disposizione tre metodi:

- **Volumes:** preso in input il path che si vuole rendere persistente (directory o file), viene gestito internamente da Docker con il vantaggio di essere indipendente dal tipo di Sistema Operativo che si sta utilizzando (Figura 3.3a)
- **Bind Mount:** preso in input il path dell'host machine, Docker monta il contenuto del path sorgente nel container, da notare che è possibile specificare il path di destinazione. Lo svantaggio del bind mount dipende dal fatto che è legato al file system dell'host machine locale (Figura 3.3b)
- **Tmpfs mount:** i dati vengono temporaneamente memorizzati nella memoria centrale e non su disco (Figura 3.3c)

Per il corretto funzionamento, l'applicazione deve rendere persistenti:

- i file contenenti le rappresentazioni degli articoli
- la directory data, utilizzata dal database MongoDB per salvare gli oggetti inseriti da utente

In entrambi i casi è stato configurato un bind mount per permettere non solo ai dati di essere persistenti ma anche di essere più facilmente controllabili durante la fase di sviluppo. Nella Figura 3.3 viene riportata la configurazione dell'ambiente in cui si esegue l'applicazione.

version: '3'

services:

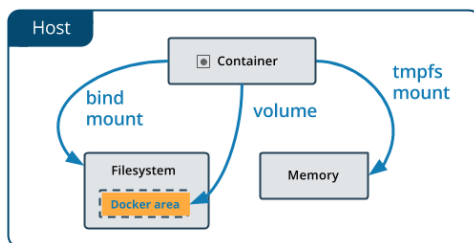
```
docudipity-backend:
  container_name: docudipity_api
  restart: unless-stopped
  build: .
  ports:
    - '80:3000'
  volumes:
    - ./docudipity_res:/usr/src/docudipity/src/api/res
  links:
    - mongo
mongo:
  container_name: mongo
```

```

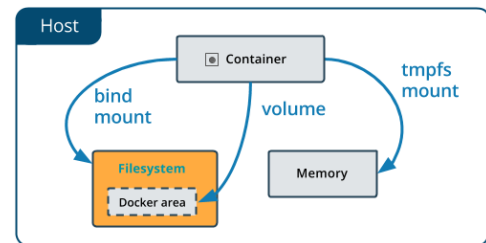
image: mongo
environment:
  - MONGO_INITDB_DATABASE=docudipity
  - MONGO_INITDB_ROOT_USERNAME=stefano
  - MONGO_INITDB_ROOT_PASSWORD=*****
volumes:
  - ./data:/data/db
ports:
  - '27017:27017'

```

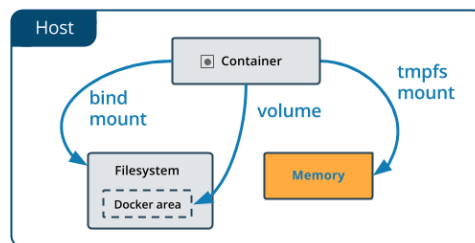
Listing 3.3: docker-compose.yml contenente la dichiarazione dei servizi utilizzati dall'applicazione



(a) Volumes, immagine disponibile sul sito di Docker all'indirizzo <https://docs.docker.com/storage/images/types-of-mounts-volume.png>



(b) Bind Mount, immagine disponibile sul sito di Docker all'indirizzo <https://docs.docker.com/storage/images/types-of-mounts-bind.png>



(c) Tmpfs, immagine disponibile sul sito di Docker all'indirizzo <https://docs.docker.com/storage/images/types-of-mounts-tmpfs.png>

Figura 3.3: Esempi di meccanismi per persistenza dati in Docker

3.3 Architettura dell'applicazione

Per la realizzazione di questa applicazione è stata utilizzata un'architettura simile a quella presentata nell'articolo [42], il quale presenta un raggruppamento tipo (MVC) per ogni componente sviluppato, consentendo l'estensione dell'applicazione con nuovi componenti in maniera flessibile. Dopo una breve introduzione del modello MVC e dei vantaggi che esso offre, verrà presentata in maggior dettaglio la suddivisione adottata.

3.3.1 Model-View-Controller

Model-View-Controller (conosciuto come MVC [24]) è un pattern architetturale, tradizionalmente utilizzato per lo sviluppo di applicazioni desktop che implicano l'implementazione di un'interfaccia utente. Il crescente utilizzo di questo pattern, anche per strutturare il codice delle applicazioni web, ne ha aumentato notevolmente la popolarità. Secondo la visione di MVC ogni porzione di codice ha uno scopo, che può differire da quello di altre porzioni. Quindi, da questa consapevolezza nasce l'idea di raggruppare le porzioni di codice che condividono uno stesso scopo, evitando così insiemi eterogenei di codice che complicano la lettura e il debug dell'applicazione. Secondo MVC ogni porzione di codice può essere ricondotta ad uno dei tre seguenti scopi:

- **Model:** si occupa di gestire i dati dell'applicazione in maniera indipendente dalla visualizzazione
- **View:** si occupa della presentazione dei dati nell'interfaccia grafica
- **Controller:** riceve gli input di utenti e li converte in comandi per il model o la view

Nello sviluppo di applicazioni REST, gli input sono le richieste HTTP che il front-end invia al server per ottenere il contenuto di pagine o di dati da visualizzare seguendo la struttura specificata nella documentazione delle API. Nella Figura 3.4 viene mostrata una panoramica di quanto precedentemente detto. Le richieste HTTP possono essere identificate attraverso due parametri fondamentali:

- **metodo:** identifica il tipo di richiesta (GET, DELETE, POST, ecc.)
- **path:** identifica il percorso della risorsa cercata sul web server

La route è il modulo che specifica, per ogni collezione e risorsa, il metodo HTTP supportato con il relativo URI (o path della risorsa) ed assegnando il relativo controller per gestire la richiesta.

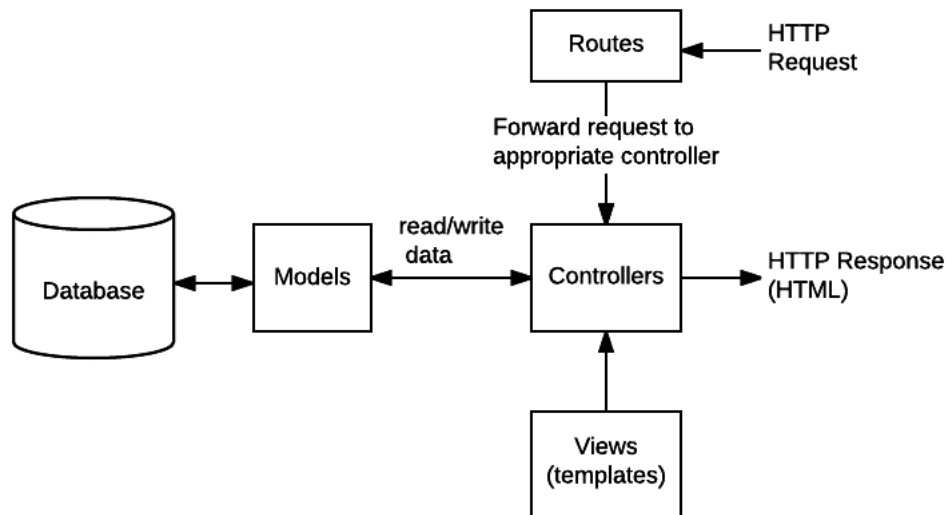


Figura 3.4: Panoramica dell'architettura di un'applicazione Express, immagine disponibile sul sito MDM Web Docs all'indirizzo https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes/mvc_express.png

3.3.2 Esempio di implementazione Endpoint

In questa sezione verrà analizzato il codice utilizzato per l'implementazione delle API REST. Nello specifico verrà presentata:

- l'implementazione della route per la registrazione di un nuovo utente, utilizzata per l'instradamento della richiesta e della verifica del body
- l'implementazione del controller utilizzato per la registrazione di un nuovo utente, responsabile dell'inserimento dei dati nel database
- l'implementazione dell'error handler, responsabile dell'invio di una risposta in caso di errore
- la discussione dello schema view

```

1 router
2   .post('/',
3     [
4       check('username')
5         .isLength({min: 4})
  
```

```

6         .withMessage('The minimum length of username is 4
characters'),
7
8         check('email')
9             .isEmail()
10            .withMessage('Invalid email address')
11            .normalizeEmail(),
12
13        check('password')
14            .isLength({min: 8, max: 16})
15            .withMessage('The lenght of password must be
between 8 and 16 characters'),
16    ],
17    (req, res, next) => {
18        const error = validationResult(req).formatWith(({msg})
=> msg)
19
20        if(error.isEmpty()) {
21            return next()
22        }
23        return res.status(StatusCode.BAD_REQUEST).json({
error: error.array() })
24    },
25    userController.signUp)

```

Listing 3.4: Route user

La Figura 3.4 evidenzia la route per registrare un utente (POST /users/). Per determinare la risposta è necessario verificare che i campi del body rispettino i vincoli imposti (come ad esempio controllare che la lunghezza della password sia nei limiti prestabiliti), per cui:

1. se la validazione fallisce, si invierà al client una risposta di errore (400 BAD REQUEST)
2. se la validazione termina con successo, la richiesta viene passata alla funzione che implementa la registrazione (userController.signUp)

```

1 signUp: async (req, res, next) => {
2     console.log('POST user (signup)')
3     const salt = utils.generateSalt()
4     const object = utils.hashPassword(req.body.password,
salt)

```



```

5     const userObject = {
6         username: req.body.username,
7         email: req.body.email,
8         password: object.hashPassword,
9         salt: object.salt
10    }
11    try {
12        const userToSave = new user(userObject)
13        const doc = await userToSave.save()
14        return res.status(StatusCode.CREATED).json({ id:
doc._id })
15    } catch(err) {
16        return next(err)
17    }
18 }

```

Listing 3.5: SignUp

Una volta terminata la fase di verifica, inizia la fase di registrazione vera e propria in cui verranno inseriti i dati dell'utente nel database (mostrata nella Figura 3.5). La fase della registrazione prevede la creazione dell'oggetto utente, con i campi del body ed eventuali campi popolati dal server (come ad esempio il campo salt) e l'inserimento dell'oggetto creato nella relativa collezione del database. Se il salvataggio viene eseguito con successo, il server invia una risposta affermativa al client (201 CREATED). Altrimenti se durante il salvataggio si verifica un errore, come ad esempio un problema di connessione o di campi duplicati, l'errore verrà gestito dall'implementazione personalizzata dell'error handler (invocato con il comando `next(err)`).

```

1 app.use(function (err, req, res, next) {
2     console.log(err)
3
4     if(err.name === 'MongoError' && err.code === 11000){
5         return res.status(StatusCode.CONFLICT).json({error: '
Duplicate'})
6     }
7
8     if(!err.statusCode) {
9         err.statusCode = StatusCode.INTERNAL_SERVER_ERROR
10    }
11
12    return res.status(err.statusCode).json({error: err.message
})

```

Listing 3.6: Error handler

Per sovrascrivere il default error handler di Express è necessario implementare una funzione con quattro parametri: error, request, response e next. In questo modo, è possibile la gestione centralizzata degli errori che si verificano durante l'esecuzione dell'applicazione. La Figura 3.6 mostra l'implementazione dell'error handler utilizzato per la gestione degli errori che possono verificarsi durante l'esecuzione dell'applicazione. Tali errori si suddividono in due categorie:

1. **Errori procedurali:** causati non da un bug interno dell'applicazione ma da una causa esterna, come ad esempio un errore di connessione oppure ad un input sbagliato dell'utente
2. **Errori del programmatore:** causati da un errore di implementazione, come ad esempio il tentativo di modificare una costante oppure l'uso di una variabile non dichiarata

La distinzione di queste due tipologie di errori è stata resa possibile grazie all'utilizzo degli status code, in quanto se l'errore rientra in uno dei casi previsti durante la creazione del messaggio di errore, viene aggiunto il relativo status code (401, 403, 404, ecc.). La mancanza dello status code, invece, indica un errore non previsto causato, probabilmente, da un errore nel codice.

```
1 const viewSchema = new schema({
2   title: {
3     type: String,
4     required: true
5   },
6   desc: {
7     type: String,
8     required: true
9   },
10  manifestations: [String],
11  viewTypeRef: {
12    type: mongoose.Schema.Types.ObjectId,
13    ref: 'viewType',
14    required: true
15  },
16  projectRef: {
17    type: mongoose.Schema.Types.ObjectId,
18    ref: 'project',
```

```

19     required: true
20   },
21   additionalProperties: {
22     type: mongoose.Schema.Types.Mixed,
23     required: false
24   },
25   upload: {
26     type: Date,
27     required: true
28   },
29   lastModified: {
30     type: Date,
31     required: true
32   }
33 })

```

Listing 3.7: View Schema

Infine, una breve discussione dello schema della view. In generale, lo scopo dello schema è quello di fornire la forma dei documenti che verranno inseriti in una collezione e nella Figura 3.7 viene mostrato lo schema della view. I campi della view si distinguono in due categorie:

1. la prima contenente i metadati della view (title, desc, manifestations, viewTypeRef, projectRef, owner, additionalProperties)
2. la seconda contiene i campi utilizzati per il logging, che sono l'upload e lastModified

Da notare la dichiarazione di tipo Mixed, del campo additionalProperties, in quanto permette di avere documenti con strutture differenti. Tale campo verrà utilizzato dall'applicazione front-end per salvare i dati necessari al ripristino dello stato della view.

3.3.3 Discussione dell'architettura del progetto

Nella root del progetto (mostrata nella Figura 3.5), oltre ai file di configurazione dell'applicazione (ecosystem.config.js) contenenti le variabili d'ambiente (credenziali autenticazione database, segreto per firmare il token) e di tracciamento delle dipendenze (package.json, package-lock.json), sono presenti i file per la configurazione di docker.

Il Dockerfile contiene i comandi necessari per la costruzione dell'immagine dell'applicazione.

Il docker-compose.yaml, invece, definisce i servizi necessari per eseguire l'applicazione:

- **mongo**: container che esegue un'istanza di mongoDB
- **docudipity-backend**: contiene i parametri di configurazione dell'immagine generata dal Dockerfile

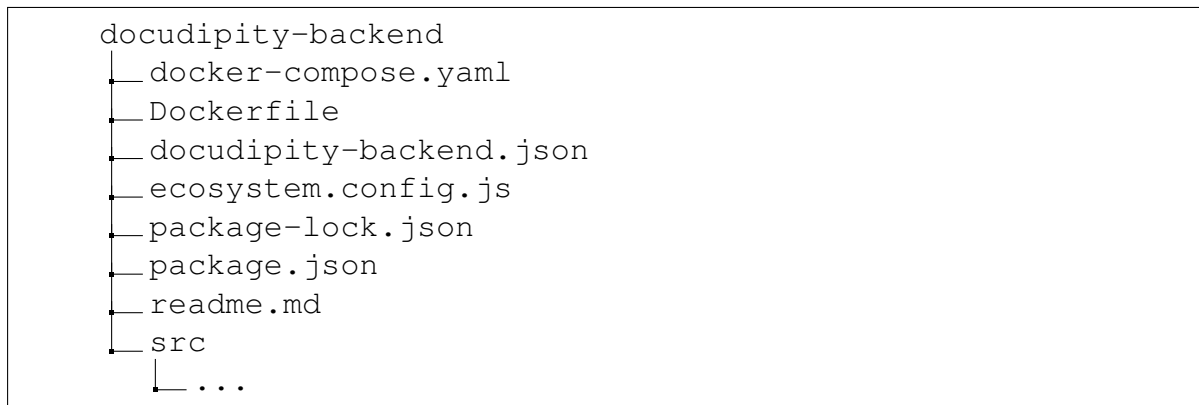


Figura 3.5: Albero della root del progetto DocuDipity backend

Nella Figura 3.6 vengono rappresentati gli elementi della cartella `src`, i cui elementi principali sono:

- **api**: contiene la definizione ed implementazione degli endpoint
- **database**: contiene lo script per l'inizializzazione del db (`mongo.seed.js`) e il modulo per la gestione della connessione al db (`connection.js`)
- **init.js**: script per il caricamento automatico di conferenze con relativi articoli
- **mongo-data**: contiene i dati (in formato JSON) utilizzati per inizializzare le collezioni. Ogni sub-directory rappresenta una collezione ed il nome segue lo schema `x-yyy` dove:
 - `x` è un numero che rappresenta l'ordine con il quale la collezione sarà inizializzata
 - `yyy` rappresenta il nome della collezione da inizializzare
 - all'interno della directory si possono aggiungere un numero arbitrario di file contenenti i dati da inserire nel db
- **test.js**: file contenente i test degli endpoint

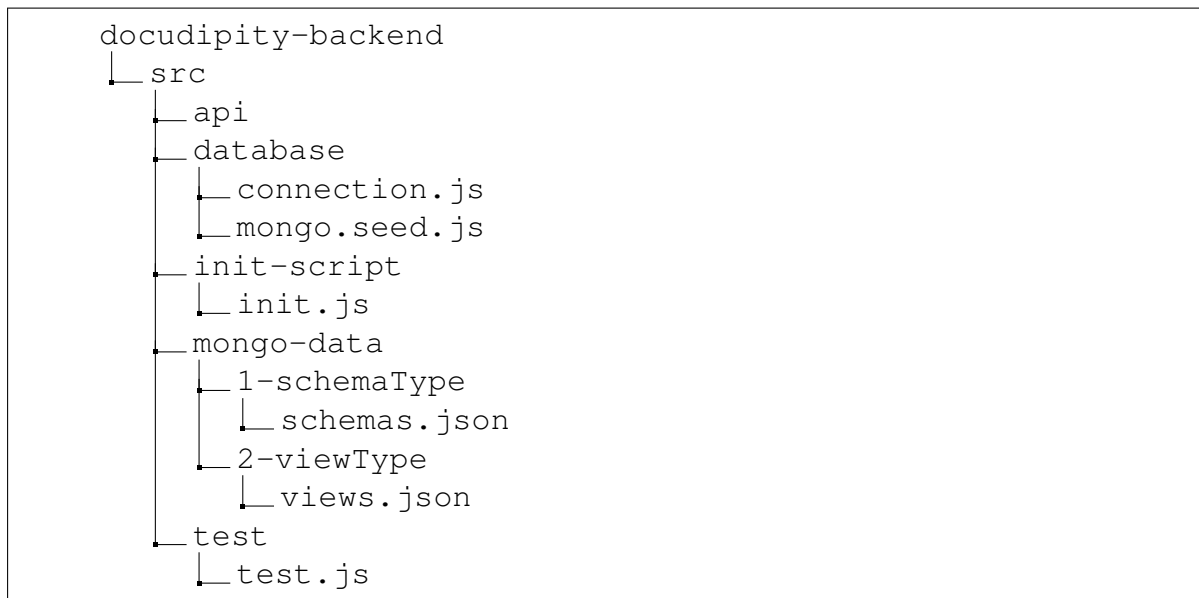


Figura 3.6: Albero che mostra il contenuto della cartella src

Un middleware è una funzione che riceve gli oggetti request e response. In più, ha un terzo parametro next che invoca la funzione successiva. Il vantaggio dell'utilizzo dei middleware è quello di implementare una qualunque funzione di utilità (servizio), come ad esempio la validazione della richiesta o verificare l'autenticazione, separata dalla singola route che non ne integra l'implementazione ma solo la chiamata, così da condividere la funzione con le route che ne necessitano l'utilizzo. Durante lo sviluppo di questa applicazione è emersa la necessità di implementare due servizi (mostrati nella Figura 3.7):

1. **auth.js**: implementa i servizi di autenticazione e di verifica dei permessi
2. **upload.js**: gestisce l'upload dei file, assegnando il nome al file caricato e la cartella di destinazione

La directory components raggruppa le risorse gestite dal server, mostrata nella Figura 3.8. Ogni risorsa ha una directory con il proprio nome (es. documentSet) e tre file:

1. ***.model.js**: contiene la definizione dello schema (proprietà e vincoli) e di eventuali middleware
2. ***.controller.js**: contiene l'implementazione degli endpoint e interagisce con il *.model.js per le interrogazioni al database
3. ***.routes.js**: associa ad ogni route la relativa implementazione contenuta nel controller e la validazione della richiesta



Figura 3.7: Albero che mostra il contenuto della directory api

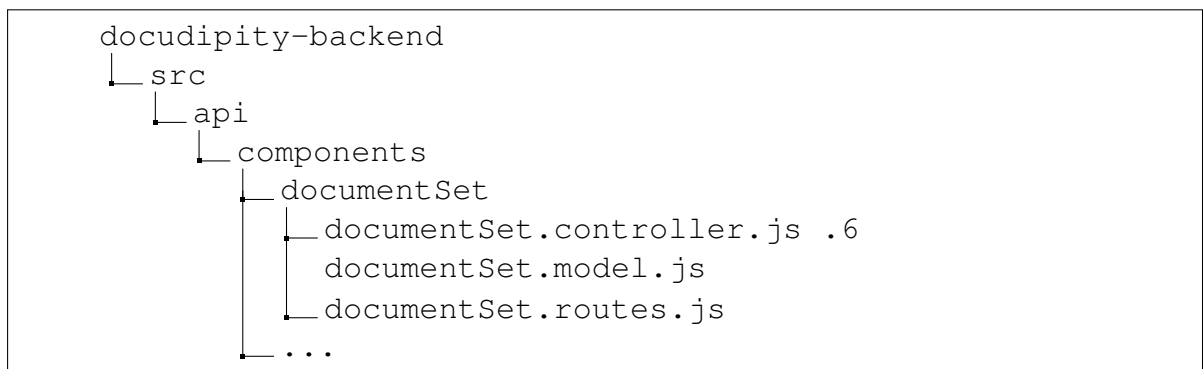


Figura 3.8: Albero che mostra il contenuto della directory components

Capitolo 4

Valutazione

Dopo aver discusso della fase di progettazione e implementazione dell'applicazione, verrà affrontata la corrispondente fase di test. La trattazione è suddivisa in due parti:

1. nella prima parte si descrivono i test di tipo BDD (Behaviour Driven Development [35, 28]) utilizzati per automatizzare la verifica della corretta implementazione delle API
2. nella seconda parte, invece, si descrivono gli scenari utilizzati per testare le prestazioni dell'applicazione

4.1 I requisiti funzionali

Il termine “requisiti funzionali” indica l'insieme dei comportamenti o delle funzioni che l'applicazione deve implementare per consentire all'utente di completare il proprio incarico. Alcuni esempi di requisiti funzionali sviluppati nella nostra applicazione sono:

1. implementazione API per la registrazione di un utente
2. implementazione meccanismo di autorizzazione basato su token
3. implementazione API per l'upload di file contenenti le rappresentazione degli articoli

Per verificare la corretta implementazione delle funzionalità richieste è stato utilizzato il metodo di sviluppo BDD. Tale metodo consiste nell'utilizzare un linguaggio, condiviso all'interno del team di sviluppo, con cui scrivere i test per verificare il comportamento dell'applicazione su un certo numero finito di casi. Col procedere dello sviluppo e con la progressiva implementazione di nuove funzionalità, si aggiungono nuovi test. Questo permette di avere un primo riscontro della correttezza del codice scritto, in quanto verifica

in modo automatico non solo i flussi principali ma anche i casi di errore e simili. La principale differenza tra il metodo BDD e TDD (Test Driven Development [28]) è data da chi scrive il codice per testare l'applicazione, ovvero il BDD è scritto dagli utenti o dai tester, mentre il TDD è scritto dagli sviluppatori. Comunque, nei team di piccole dimensioni, come il nostro, queste due metodologie possono essere considerate uguali poiché spesso i ruoli si sovrappongono.

4.1.1 Gli strumenti utilizzati

Per costruire il test con cui verificare la correttezza del codice scritto è stato necessario utilizzare le seguenti librerie di npm:

- **Mocha.js** [13]: JavaScript test framework eseguibile sia su Node.js che su browser, permette di semplificare il testing di codice asincrono. Mocha mette a disposizione dell'utente numerose interfacce di testing (TDD, BDD, Export, QUnit e require) da scegliere in base alle proprie preferenze
- **Chai** [5]: libreria per i modelli BDD/TDD con cui verificare asserzioni, disponibile sia per node che per browser e che può essere facilmente integrata con un qualsiasi framework di testing JavaScript (Mocha.js)
- **Supertest** [41]: libreria che espone un'implementazione dei metodi HTTP, utile per testare gli endpoint implementati da una web app
- **Mochawesome** [17]: libreria che permette di visualizzare i risultati dei test, eseguiti tramite il framework Mocha, in una pagina HTML/CSS

4.1.2 La struttura dei test

I test sono suddivisi in gruppi, ognuno dei quali è associato ad una risorsa quindi abbiamo un totale di otto gruppi che, rapportandoli al relativo metodo CRUD, si ottengono centododici test. Nel seguito verrà spiegato in maggior dettaglio i test effettuati per ciascuna API con gli eventuali casi particolari da tenere in considerazione. La suddivisione è stata fatta attraverso il costrutto `describe`, il quale permette di raggruppare sia i singoli test che altri `describe` associati al medesimo gruppo. Nella Figura 4.1 si può osservare il gruppo root (API test) e due livelli nidificati:

1. il gruppo che aggrega tutte le API che operano su una stessa risorsa, ad esempio `DocumentSets API`
2. il gruppo che aggrega i singoli test (`it`) su una specifica API, ad esempio `POST /documentSets/:documentSetId`


```

1 describe('API test', () => {
2     ...
3     before(async () => {
4         // recupera i token di autorizzazione necessari per l'
5         // esecuzione dei test
6         ...
7         describe('DocumentSet API', () => {
8             before(async () => {
9                 // inizializza l'ambiente di test relativo alla
10                // risorsa DocumentSet
11                describe('POST /documentSets', () => {
12                    it('Check security', async () => {
13                        ...
14                    })
15                    it('Malformed body', async () => {
16                        ...
17                    })
18                })
19                describe('POST /documentSets/:documentSetId', () => {
20                    it('Check security', async () => {
21                        ...
22                    })
23                    it('Check update', async () => {
24                        ...
25                    })
26                    it('Malformed body', async () => {
27                        ...
28                    })
29                    it('Malformed param (:documentSetId)', async () =>
30                    {
31                        ...
32                    })
33                })
34                describe('GET /documentSets', () => {
35                    it('Check response', async () => {
36                        ...
37                    })
38                    it('Missing query', async () => {

```

```

38         ...
39     })
40     it('Wrong query type', async () => {
41         ...
42     })
43 })
44 ...
45 describe('DELETE /documentSets/:documentSetId', () =>
46 {
47     it('Check security', async () => {
48         ...
49     })
50     it('Malformed param', async () => {
51         ...
52     })
53     after(async () => {
54         // ripristina lo stato iniziale dell'ambiente
55     })
56 })
57 ...
58 })

```

Listing 4.1: Struttura test

Il costrutto “before” viene eseguito prima dei test. Nel nostro caso, se il before è al livello root si occupa dell’inizializzazione dei token di sicurezza, mentre i before nidificati si occupano dell’inizializzazione dell’ambiente su cui eseguire i test. Similmente, il costrutto “after” viene eseguito al termine dei test per ripristinare lo stato iniziale dell’ambiente. I casi test che sono stati scritti per l’API dipendono, principalmente, da due fattori:

1. il tipo di operazione eseguita dall’API, rispetto alle operazioni CRUD
2. la presenza di parametri nell’URI o di query string

Nella nostra applicazione sono state sviluppate quaranta API, che riportandole al tipo di operazione eseguita (Create, Read, Update, Delete) e alla presenza o meno di query string o param, otteniamo un numero di test pari a centododici. Tenendo presente che abbiamo due eccezioni:

1. per l’API di login, oltre al test malformed body, contiene altri due test non presenti nel modello precedentemente presentato. Nello specifico sono:

CRUD	Nome Test	Breve descrizione	Obbligatorio
C-UD	Check security	Controlla che le richieste senza token vengano respinte	Y
C-U-	Malformed body	Controlla che le richieste con il body vuoto o malformato vengano respinte	Y
CRUD	Malformed param	Controlla che le richieste con param invalidi o inesistenti vengano respinte	N
C—	Duplicate	Controlla che le richieste di inserimento di una nuova risorsa che violino il vincolo di unicità vengano respinte	N
-R-	Check response	Controlla che le risposte contengano i dati cercati	Y
	Missing query	Controlla che le richieste senza le query string obbligatorie vengano respinte	N
	Wrong query type	Controlla che le richieste con query string con tipo diverso da quello aspettato vengano respinte	N
-U-	Check update	Controlla che le richieste di aggiornamento vengano effettivamente applicate	Y

Tabella 4.1: Tabella che riporta per ogni operazione CRUD i relativi test

- (a) **Wrong password:** verifica che in caso di password sbagliata l'accesso venga negato
- (b) **Check login:** verifica che il token venga regolarmente rilasciato in caso di successo dell'autenticazione

2. l'API di inserimento di un nuovo utente non richiede il token di autorizzazione, quindi non è presente il test check security

4.1.3 I risultati

Nella Figura 4.1 si possono osservare alcune informazioni di carattere generale dello svolgimento del test, come ad esempio il tempo impiegato per l'esecuzione di tutti test e il numero di test terminati con successo o falliti. Da tale immagine possiamo osservare che il tempo impiegato per l'esecuzione dei centododici test è pari a 3.1s e che sono terminati tutti con successo. Considerando che per ciascuna API, oltre a verificare il corretto

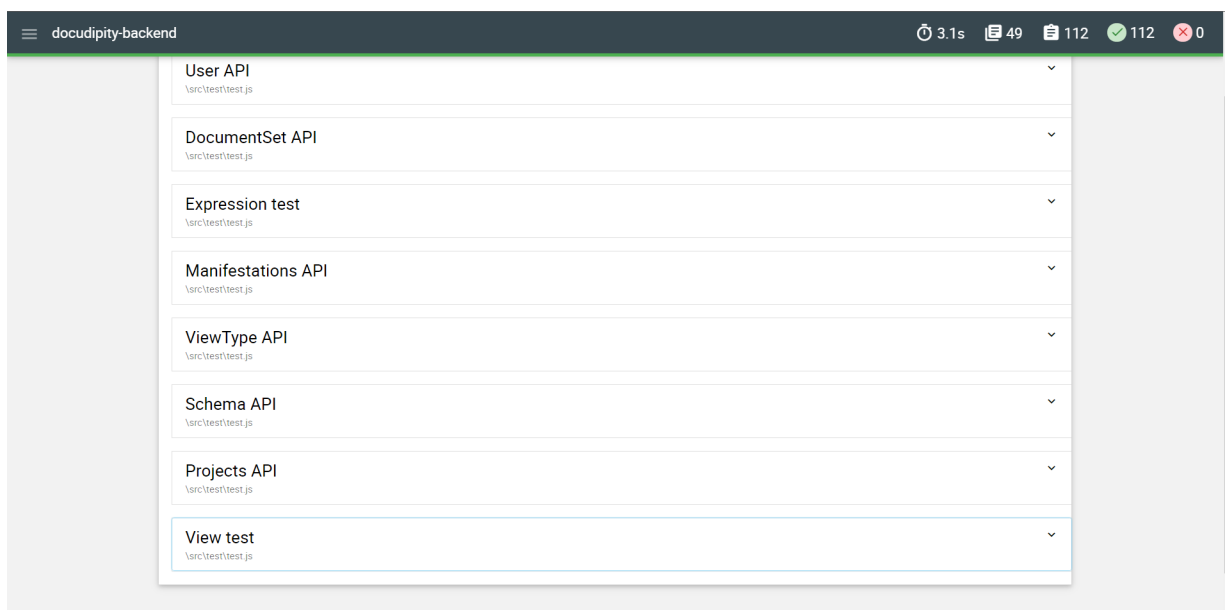
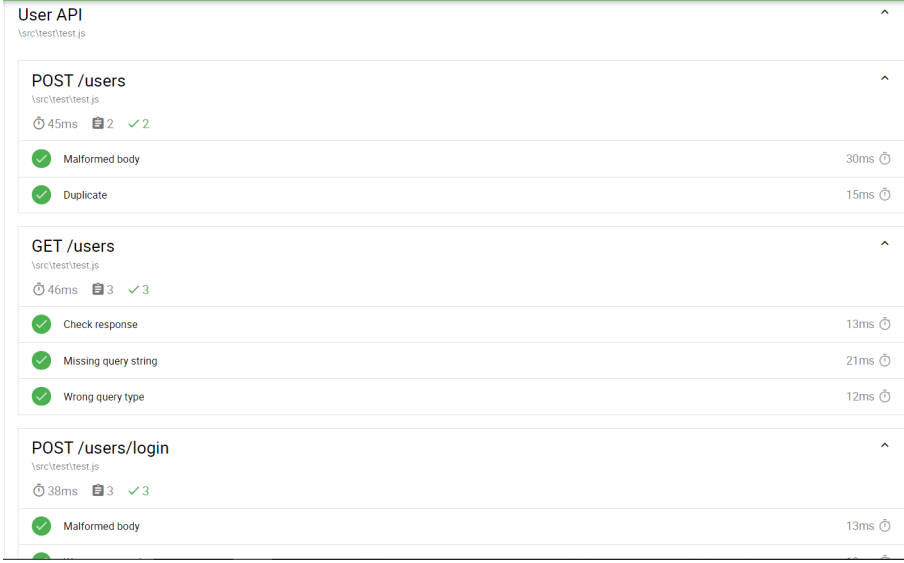


Figura 4.1: Overview del report prodotto al termine del test

funzionamento con le richieste che rispettano i vincoli previsti, sono stati controllati anche i comportamenti di richieste con parametri (query string o path) obbligatori ma assenti o di tipo diverso da quello aspettato e con body che non rispettano lo schema previsto dalla risorsa. Possiamo, allora, affermare che l'applicazione prodotta mantiene un comportamento corretto sia nel caso di richieste che rispettino i vincoli previsti sia nel caso di richieste malformate. Infatti, tutte le versioni dell'applicazione rilasciate sul server del dipartimento, per mettere a disposizione del front-end le API necessarie per

eseguire la sua applicazione, che hanno superato i test precedentemente descritti non hanno presentato comportamenti anomali. Questo fatto deriva dalla qualità dei test a cui l'applicazione è stata sottoposta. Infine, nella Figura 4.2 viene riportato, a titolo dimostrativo, il report dettagliato dell'esecuzione dei test di due API legate alla risorsa users.



User API	
↳src/test/test.js	
POST /users	
↳src/test/test.js	
🕒 45ms 📄 2 ✓ 2	
✓ Malformed body	30ms 🕒
✓ Duplicate	15ms 🕒
GET /users	
↳src/test/test.js	
🕒 46ms 📄 3 ✓ 3	
✓ Check response	13ms 🕒
✓ Missing query string	21ms 🕒
✓ Wrong query type	12ms 🕒
POST /users/login	
↳src/test/test.js	
🕒 38ms 📄 3 ✓ 3	
✓ Malformed body	13ms 🕒

Figura 4.2: Esempio di dettaglio del report per le API legate alla risorsa users

4.2 I requisiti non funzionali

I requisiti non funzionali specificano un criterio con il quale valutare la qualità del software prodotto, invece di concentrarsi sul comportamento dell'applicazione. Alcuni esempi di requisiti funzionali, estratti dall'articolo [9], sono:

- **Performance:** termine con il quale si indica la capacità complessiva del sistema di gestire richieste concorrenti, raccogliendo statistiche riguardanti il numero di richieste per secondo (rps), i tempi di risposta e altri indicatori
- **Survivability:** la capacità del sistema di continuare a gestire le operazioni pre-configurate gestendo gli eventuali errori
- **Availability:** la capacità del sistema di rendere disponibili i servizi richiesti per l'esecuzione di una task da parte dell'utente
- **Interoperability:** la capacità dell'applicazione di interfacciarsi con altri servizi/applicazioni

- **Portability**: la capacità di eseguire l'applicazione in un ambiente software/hardware diverso

Il test utilizzato è quello delle performance, che può essere distinto in due categorie:

1. **Load Test**: verificare che l'applicazione sia in grado di gestire il numero di richieste concorrenti stimate
2. **Stress Test**: il suo obiettivo è individuare il punto di rottura dell'applicativo

La nostra applicazione è stata testata con il Load Test, utilizzando come strumenti Python [15] e il modulo Locust [3].

Il modulo Locust permette di definire, con un semplice file python, varie tipologie di utenti ed assegnare, a ciascuna istanza di utenti creati, un insieme di task (operazioni da simulare, come ad esempio una GET di una pagina del sito). I componenti principali di tale modulo, sono:

- **UserClass**: rappresenta una categoria di utenti che possono effettuare un insieme di task, in generale è possibile configurare più categorie di utenti con task differenti
- **Task**: rappresenta le richieste che l'utente generato effettuerà, è possibile assegnare una priorità ad ogni task, se assente ogni task ha la stessa possibilità di essere eseguito
- **Events**: permette di definire una funzione che viene richiamata quando si verificano certi eventi, un esempio di evento è la partenza o l'arresto del test utile per inizializzare valori necessari per eseguire le richieste, come ad esempio i token di autenticazione
- **HttpUser**: implementazione dei principali metodi HTTP, permette di inviare richieste al server utilizzando JSON

4.2.1 La struttura del test

Nel caso di test preso in considerazione, riportato nella Figura 4.2, è stata utilizzata una singola classe di utenti che possono effettuare quattro tipi di operazioni:

1. interrogazione della lista di manifestation disponibili per una certa expression
2. interrogazione dei metadati del progetto precedentemente creato
3. richiesta di aggiornamento dei metadati di un expression con dati fittizi (generati con la funzione random)

4. richiesta di aggiornamento dei metadati del project con dati fittizi (generati con la funzione random)

Per tali operazioni non è stato assegnato un livello di priorità, perciò hanno tutte la stessa probabilità di essere eseguite. Per poter effettuare correttamente queste operazioni, al momento della creazione dell'istanza dell'utente (evento on_start) è necessario:

1. autenticarsi, per ottenere il token di autorizzazione
2. inizializzazione dell'array di expression, necessaria per interrogare gli endpoint con dati reali
3. creare il progetto di prova

```
1 import time, random
2 from locust import HttpUser, task, constant_pacing
3
4 class ProjectTest(HttpUser):
5     wait_time = constant_pacing(1)
6
7     def on_start(self):
8         self.token = 'Bearer '
9         self.expressions = []
10        self.projectId = ''
11        with self.client.post('/api/users/login', json={"email": '
admin@example.it', "password": 'adminadmin'}) as response:
12            response_json = response.json()
13            self.token = self.token + response_json['token']
14            with self.client.get('/api/expressions/?page=0&limit=15') as
response:
15                response_json = response.json()
16                for exp in response_json['expressions']:
17                    self.expressions.append('/api/expressions/' + exp['_id'])
18            with self.client.post('/api/projects/', json={"name": 'locust
project', "desc": "locust desc", "expressions": self.expressions},
headers={'Authorization': self.token}) as response:
19                response_json = response.json()
20                self.projectId = response_json['id']
21
22        def on_stop(self):
23            self.client.delete('/api/projects/' + self.projectId, headers={'
Authorization': self.token})
24
25        @task
26        def readManifestations(self):
27            index = random.randrange(0, len(self.expressions))
28            self.client.get(self.expressions[index] + '/manifestations/?page
=0&limit=5')
```

```

29
30
31 @task
32 def readProject(self):
33     self.client.get('/api/projects/?page=0&limit=50')
34
35 @task
36 def writeExpression(self):
37     index = random.randrange(0, len(self.expressions))
38     title = random.randrange(0, 1000)
39     comp_title = '00000' + str(title)
40     desc = random.randrange(0,1000)
41     self.client.post(self.expressions[index], json={'title': str(
42         comp_title), 'desc': str(desc)}, headers={'Authorization': self.token})
43
44 @task
45 def writeProject(self):
46     name = random.randrange(0,1000)
47     comp_name = '00000' + str(name)
48     desc = random.randrange(0,1000)
49     self.client.post('/api/projects/' + self.projectId, json={'name':
50         str(comp_name), 'desc': str(desc)}, headers={'Authorization': self.
51         token})

```

Listing 4.2: File Python utilizzato per il Load Test

4.2.2 I risultati

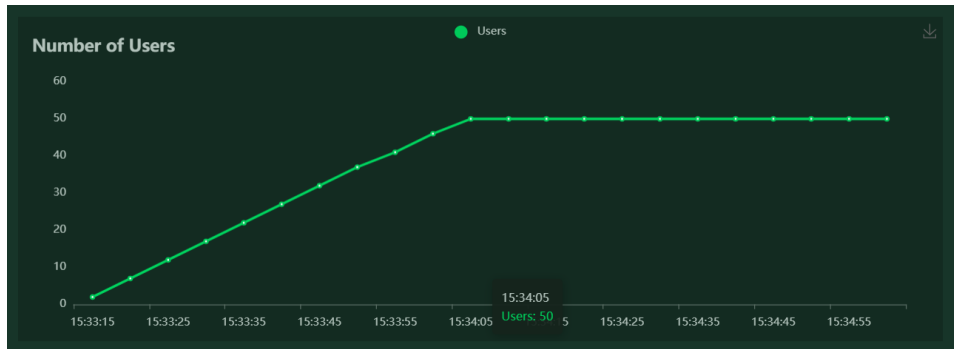
Per questo test sono state eseguite due differenti simulazioni, entrambe sono state svolte sullo stesso PC, dotato di un processore i7-6500U con una RAM di 12 GB e come HDD HTS541010A7E630. Nella prima esecuzione sono stati simulati 50 utenti con uno spawn rate di 1 utente per secondo, riportato nella Figura 4.3a, e nella seconda sono stati simulati 100 utenti con uno spawn rate di 2 utenti per secondo, riportato nella Figura 4.3b.

Ogni istanza di utente utilizza la funzione `constant_pacing`, che permette di impostare un timer adattivo che garantisce ad ogni utente di eseguire un'operazione ogni X secondi.

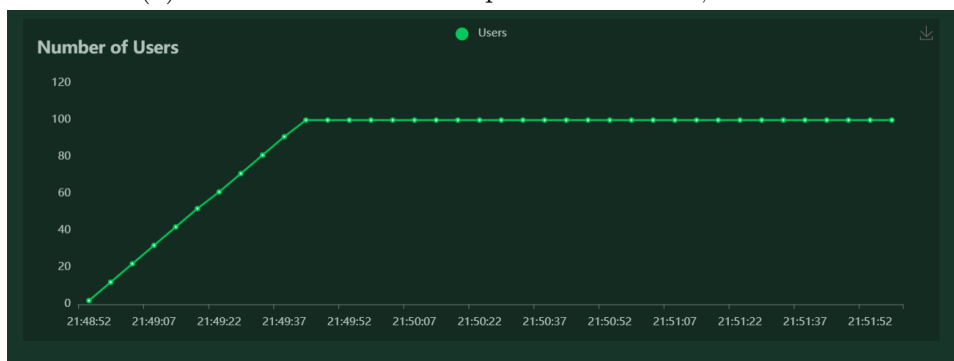
Per le simulazioni effettuate è stato imposto che ogni utente esegua almeno un'operazione ogni secondo (il valore assegnato a X è 1s), quindi ci aspettiamo che per il caso di 50 utenti non si superi la soglia di 50 rps e per il caso di 100 utenti la soglia di 100 rps. I relativi grafici sono riportati rispettivamente nella Figura 4.4c e 4.4d.

Infine, esaminando i tempi di risposta ottenuti dalle due simulazioni è emerso:

1. per il caso di 50 utenti un tempo di risposta medio di 80ms, riportato nella Figura 4.4a



(a) Numero di utenti della prima esecuzione, 50 utenti

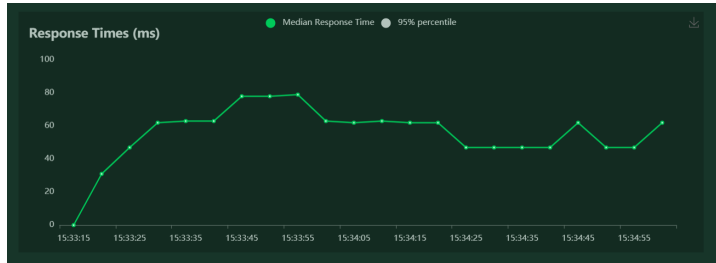


(b) Numero di utenti della seconda esecuzione, 100 utenti

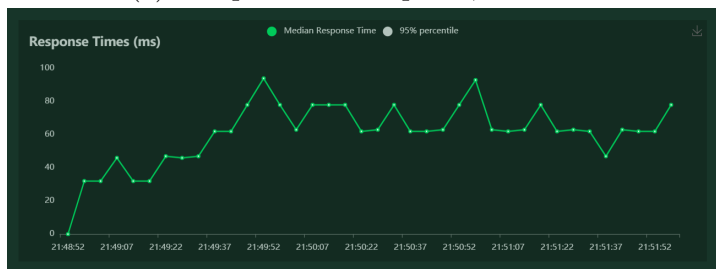
Figura 4.3: Numero di utenti simulati

- per il caso di 100 utenti un tempo di risposta medio di 103ms, riportato nella Figura 4.4b

Prima di procedere con l'analisi dei risultati è necessario specificare che l'applicazione back-end e l'applicazione di test sono state eseguite sullo stesso PC in locale, le cui specifiche sono state precedentemente esplicitate. La prima considerazione da fare sui grafici ottenuti è quella di osservare che, al raddoppio delle richieste per secondo non è corrisposto un raddoppio dei tempi di risposta. Più precisamente, abbiamo riscontrato un aumento del 28,75%. Osservando i grafici dei tempi di risposta e la relativa media notiamo una discrepanza, nello specifico la media ha un valore maggiore di quella deducibile dai grafici. Questo problema è causato dal fatto che i grafici ignorano le richieste eseguite dopo il termine del test (`on_stop`), con lo scopo di ripristinare lo stato iniziale dell'ambiente. Comunque, considerando che le richieste di eliminazione in un contesto reale sono inferiori rispetto a quelle di creazione/lettura e tenendo presente delle limitazioni del PC, ci riteniamo soddisfatti dei risultati ottenuti.



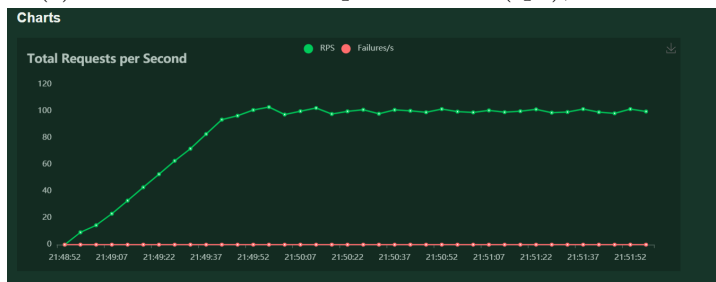
(a) Tempi medi di risposta, 50 utenti



(b) Tempi medi di risposta, 100 utenti



(c) Numero di richieste per secondo (rps), 50 utenti



(d) Numero di richieste per secondo (rps), 100 utenti

Figura 4.4: Grafici che riportano le richieste per secondo e i relativi tempi di risposta medi

Capitolo 5

Conclusioni

Lo scopo di questa tesi era la progettazione e lo sviluppo di un'API REST per il progetto DocuDipity. Nello specifico sono stati individuati i seguenti obiettivi:

1. possibilità di inserire nuovi articoli nel sistema con le corrispondenti operazioni di aggiornamento o eliminazione su articoli già presenti
2. sviluppare la ricerca di articoli basata su metadati, come ad esempio l'autore o il titolo
3. sviluppare un meccanismo in grado di gestire le molteplici rappresentazioni di uno stesso articolo in formati diversi
4. possibilità di inserire nuove visualizzazioni, con i relativi metadati
5. implementazione dell'autorizzazione basata su token

Utilizzando il modello FRBR è stato possibile dividere il concetto di articolo in due parti distinte, una contenente i metadati e l'altra contenente le possibili rappresentazioni secondo un certo Schema. Questo ha reso possibile lo sviluppo di API per l'interrogazione e il filtraggio di articoli in base ai metadati, oltre a consentire l'inserimento dinamico di nuovi articoli o gli eventuali aggiornamenti. Il modello dati così sviluppato però non è completo, in quanto non considera le altre risorse necessarie per il funzionamento dell'applicazione. Problema superato attraverso il modello E-R, utilizzato per rappresentare le risorse e le relazioni necessarie per supportare le operazioni che l'utente può eseguire durante l'utilizzo dell'applicazione. Quindi, al termine della fase di progettazione è stato sviluppato un modello di dati in grado di soddisfare tutti gli obiettivi preposti. Tuttavia, non di minore importanza, è stata la fase di implementazione in quanto è avvenuto il passaggio critico di come strutturare effettivamente l'applicazione. Infatti, con l'utilizzo di una cattiva struttura e una non ottimale ingegnerizzazione del codice, si avrà come conseguenza una crescente difficoltà nel mantenere l'applicazione e nell'implementazione

di nuove funzionalità. In tale capitolo, pertanto, è stato dato ampio spazio ad illustrare l'architettura adottata, in quanto unita ad una documentazione completa fornisce un'ottima base su cui implementare funzionalità più complesse. Inoltre, viene illustrato il meccanismo di autorizzazione implementato mediante l'utilizzo di JWT, rilasciato a seguito dell'autenticazione mediante email/password.

Nell'ultima parte sono stati illustrati i test a cui l'applicazione è stata sottoposta, per verificarne il corretto funzionamento e misurare le performance in caso di molteplici richieste concorrenti.

Lo sviluppo dell'applicazione presentata in questa tesi non si esaurisce con essa ma, al contrario, può essere considerata come la struttura base su cui aggiungere nuove e più complesse funzionalità.

Un primo esempio è l'aggiunta di gruppi di lavoro, attraverso i quali sia possibile collaborare nell'analisi di articoli scientifici. Oppure, aggiungere la possibilità di generare le varie rappresentazioni degli articoli direttamente nel back-end, attraverso l'integrazione dei software che generano i file contenenti i dati necessari per supportare le varie visualizzazioni.

Per quanto riguarda la gestione delle visualizzazioni supportate dal sistema, si potrebbe aggiungere un'interfaccia per amministratori attraverso la quale sia possibile inserire o modificare le visualizzazioni supportate dal sistema.

Infine, per la gestione degli account si potrebbe sviluppare un meccanismo per il ripristino della password o per la verifica dell'email in fase di registrazione.

Bibliografia

- [1] Benjamin B Bederson. Photomesa: a zoomable image browser using quantum tree-maps and bubblemaps. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 71–80, 2001.
- [2] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005.
- [3] Carl Byström, Joakim Hamrén, Hugo Heyman, and Jonatan Heyman. An open source load testing tool. <https://locust.io/>, 2021. Ultimo accesso: 06.6.2021.
- [4] Mei C Chuah. Dynamic aggregation with circular visual designs. In *Proceedings IEEE Symposium on Information Visualization (Cat. No. 98TB100258)*, pages 35–43. IEEE, 1998.
- [5] Chai Community. Chai Assertion Library. <https://www.chaijs.com/>, 2021. Ultimo accesso: 06.6.2021.
- [6] Angelo Di Iorio, Silvio Peroni, Francesco Poggi, and Fabio Vitali. Dealing with structural patterns of XML documents. *Journal of the Association for Information Science and Technology*, 65(9):1884–1900, 2014.
- [7] Inc. Docker. Docker. <https://www.docker.com/>, 2021. Ultimo accesso: 06.6.2021.
- [8] Lisa M. Dusseault and James M. Snell. PATCH Method for HTTP. RFC 5789, March 2010.
- [9] Petter LH Eide. Quantification and traceability of requirements. Master’s thesis, NTNU, Norwegian University of Science and Technology, 2005.
- [10] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.
- [11] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

- [12] OpenJS Foundation. Express: Fast, unopinionated, minimalist web framework for Node.js. <http://expressjs.com/>, 2021. Ultimo accesso: 05.6.2021.
- [13] OpenJS Foundation. Mocha. <https://mochajs.org/>, 2021. Ultimo accesso: 06.6.2021.
- [14] OpenJS Foundation. Node.js. <https://nodejs.org/en/>, 2021. Ultimo accesso: 05.6.2021.
- [15] Python Software Foundation. Python. <https://www.python.org/>, 2021. Ultimo accesso: 06.6.2021.
- [16] Martin Fowler. Richardson maturity model. <https://martinfowler.com/articles/richardsonMaturityModel.html>, 2010.
- [17] Adam Gruber. mochawesome. <https://github.com/adamgruber/mochawesome>, 2021. Ultimo accesso: 13.6.2021.
- [18] Sven Hartmann and Sebastian Link. English sentence structures and eer modeling. In *ACM International Conference Proceeding Series*, volume 247, pages 27–35. Citeseer, 2007.
- [19] Kasper Hornbæk and Erik Frøkjær. Reading patterns and usability in visualizations of electronic documents. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 10(2):119–149, 2003.
- [20] Brian Scott Johnson. *Treemaps: Visualizing hierarchical and categorical data*. PhD thesis, 1993.
- [21] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.
- [22] Michael Jones and Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, October 2012.
- [23] Kelly Patricia Kingrey. Concepts of information seeking and their presence in the practical library literature. *Library philosophy and practice*, 4(2):1–14, 2002.
- [24] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
- [25] Hao Lü and James Fogarty. Cascaded treemaps: examining the visibility and stability of structure in treemaps. In *Proceedings of graphics interface 2008*, pages 259–266, 2008.

- [26] Kevin Maiani. Progettazione e sviluppo di un ambiente di lettura e confronto tra articoli scientifici. Master's thesis, Università di Bologna, Maggio 2021.
- [27] Michael J McGuffin and Jean-Marc Robert. Quantifying the space-efficiency of 2d graphical representations of trees. *Information Visualization*, 9(2):115–140, 2010.
- [28] Myint Myint Moe. Comparative study of test-driven development (TDD), behavior-driven development (BDD) and acceptance test-driven development (ATDD). *International Journal of Trend in Scientific Research and Development (ijtsrd)*, Volume-3(10):231–234, 2019.
- [29] Inc. MongoDB. MongoDB: The database for modern applications. <https://www.mongodb.com/>, 2021. Ultimo accesso: 05.6.2021.
- [30] F. Poggi, P. Ciancarini, A. Di Iorio, S. Peroni, and F. Vitali. Exploiting coordinated views for scholarly reading and analysis. pages 113–124, 2019.
- [31] Leonard Richardson. Justice will take us millions of intricate moves - act three: The maturity heuristic. <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>, 2009.
- [32] Hans-Jorg Schulz, Steffen Hadlak, and Heidrun Schumann. The design space of implicit hierarchy visualization: A survey. *IEEE transactions on visualization and computer graphics*, 17(4):393–411, 2010.
- [33] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.
- [34] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, pages 364–371. Elsevier, 2003.
- [35] John Smart. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster, 2014.
- [36] SmartBear Software. Swagger: API Development for Everyone. <https://swagger.io/>, 2021. Ultimo accesso: 05.6.2021.
- [37] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 57–65, 2000.
- [38] John Stasko and Eugene Zhang. Focus+ context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 57–65. IEEE, 2000.

- [39] Daniel Stenberg. cURL. <https://curl.se/>, 1997. Ultimo accesso: 30.6.2021.
- [40] Barbara Tillett. Functional requirements of bibliographic records. In *in International Federation of Library Associations and Institutions*. Citeseer, 1998.
- [41] visionmedia. superagent. <https://github.com/visionmedia/superagent>, 2021. Ultimo accesso: 06.6.2021.
- [42] Lars Wächter. How i structure my node.js rest apis. <https://medium.com/swlh/how-i-structure-my-node-js-rest-apis-4e8904ccd2fb>, 2020.

Ringraziamenti

Vorrei ringraziare il Prof. Angelo Di Iorio e il Prof. Francesco Poggi per la disponibilità e i preziosi consigli ricevuti durante lo svolgimento della tesi e nella sua stesura.

Infine, vorrei ringraziare tutte le persone che hanno creduto in me, supportandomi in questo percorso.