Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

PROGETTAZIONE E SVILUPPO DI UN AMBIENTE DI LETTURA E CONFRONTO TRA ARTICOLI SCIENTIFICI

Relatore:
Chiar.mo Prof.
ANGELO DI IORIO

Correlatore:
Chiar.mo Prof.
FRANCESCO POGGI

Presentata da:
KEVIN MAIANI

Sessione straordinaria
Anno Accademico 2019/2020

Dedicato a mia madre
che mi ha sempre sostenuto.

Introduzione

L'oggetto di questo lavoro di tesi è la progettazione e lo sviluppo di un'interfaccia grafica per DocuDipity. Si tratta di un web tool interattivo preesistente sviluppato per supportare l'esplorazione e l'analisi di articoli scientifici attraverso viste differenti che ne consentono una visualizzazione alternativa a quella standard ipertestuale[16]. Lo sviluppo della nuova interfaccia si focalizza sulla risoluzione di una serie di criticità individuate in DocuDipity, il quale risultava essere non modulare e non indipendente dalle visualizzazioni, difficile da manutenere, non ben ingegnerizzato e non ben documentato.

Il seguente elaborato non si limita a proporre esclusivamente il percorso di ingegnerizzazione della nuova interfaccia, ma analizza il lavoro svolto in tutti i suoi aspetti, dall'individuazione dei requisiti, alla realizzazione di prototipi in piú fasi per rispondere ai requisiti raccolti, fino ad arrivare alla descrizione dell'architettura utilizzata ed infine alla fase di sviluppo. Il punto fondamentale su cui verte l'intero elaborato è la dimostrazione attraverso l'interfaccia DocuDipity, di come l'applicazione di tecniche di visualizzazione delle informazioni, possa portare benefici all'esperienza dell'utente che naviga nell'interfaccia per perseguire degli obiettivi. L'idea è quella di fornire una prima visualizzazione panoramica complessiva di un set di articoli scientifici che l'utente può selezionare in base ai propri interessi ed organizzare all'interno di un'area di lavoro, nella quale è possibile applicare dei filtri e mettere a confronto più articoli secondo specifiche visualizzazioni, con l'obiettivo di ricercare e scoprire nuove informazioni.

In particolare, in questo elaborato, verranno analizzate tecniche alternative di rappresentazione e visualizzazione in relazione ad articoli scientifici, con particolare attenzione a quelle utilizzate nel sistema di DocuDipity.

Negli ultimi anni, con l'incremento del volume di dati, la difficoltà nell'esplorare collezioni di informazioni è notevolmente aumentata, ed è infatti proprio in questo contesto che le tecniche di rappresentazione visiva di dati diventano di fondamentale importanza. Stiamo infatti assistendo ad un periodo di transizione, in cui le abitudini dei lettori si stanno evolvendo, il modo in cui questi consultano informazioni sta cambiando, basti pensare che il risultato di uno studio basato su una serie di sondaggi sottoposti nel corso di 30 anni a membri di facoltà scientifiche ha evidenziato come il numero di letture sia aumentato durante il periodo di tempo analizzato, a differenza del tempo medio impiegato per ogni lettura, che è invece diminuito [21][22] .

Indubbiamente, questo mutamento nelle necessità e nelle abitudini dei lettori, sta portando la progettazione e lo sviluppo di interfacce grafiche ad un nuovo livello, che consenta di far fronte a nuove esigenze e di sfruttare appieno le potenzialità dei piú recenti display ad alta risoluzione in modo da riuscire a rappresentare grandi quantità di dati in maniera ordinata e controllata dall'utente, attraverso tecniche alternative alla tradizionale lettura sequenziale.

Il seguente elaborato si inserisce all'interno di questo contesto, focalizzandosi sulla presentazione di un sistema flessibile per la visualizzazione di articoli scientifici, sfruttando in maniera efficiente tutto lo spazio a disposizione nel contesto dell'interfaccia DocuDipity, con l'obiettivo di migliorare l'esperienza utente e favorire la scoperta di nuove informazioni durante la visualizzazione degli articoli. Si vuole dare la possibilità agli utenti di compiere analisi ed esplorazioni, partendo da una vista di overview che mostra le collezioni di articoli scientifici presenti all'interno di un progetto, dal quale l'utente può selezionare articoli specifici o interi set per spostarli all'interno di un'area di lavoro in cui è possibile organizzarli attraverso una griglia che ne consente la gestione attraverso l'applicazione di filtri. Una volta organizzati, gli articoli possono essere selezionati ed aperti per essere visualizzati e messi a confronto secondo una specifica visualizzazione.

Questo obiettivo è stato realizzato attraverso la progettazione di un'interfaccia multipannello, dove ogni area della finestra svolge un ruolo preciso nella visualizzazione degli articoli. Per portare a termine l'obiettivo è stato completamente rivisto il modello concettuale del sistema precedente, introducendo nuove entità, nuove funzionalità e rivolgendo particolare attenzione alla fase di realizzazione dei vari prototipi.

Questa tesi è organizzata in diverse parti che ripercorrono le fasi di studio, analisi, design, progettazione ed implementazione che hanno portato allo sviluppo dell'interfaccia grafica:

- Il primo capitolo descrive il progetto DocuDipity sulla base del quale è stata realizzata l'interfaccia, il contesto in cui si inserisce e le tecniche di visualizzazione utilizzate nel progetto, soffermandosi su utilità e obiettivi di queste ultime, fornendo esempi concreti e citando opere correlate. In questa prima parte verranno anche trattate le limitazioni e le criticità individuate nella precedente soluzione di DocuDipity.

- Il secondo capitolo descrive gli obiettivi e i requisiti del progetto che definiscono i comportamenti e le qualità che il sistema deve avere. In questo capitolo ci si sofferma inoltre sulla fase di prototipazione, nella quale vengono realizzati una serie di wireframes.

- Il terzo capitolo descrive l'architettura utilizzata nel progetto, illustrando nel dettaglio la fase di valutazione relativa al tipo di framework da utilizzare e le tecnologie che si è deciso di adottare, spiegando infine le motivazioni alla base di determina-

te scelte di layout e mostrando graficamente i template realizzati per la versione desktop e tablet di DocuDipity.

- Il quarto capitolo si occupa della fase di progettazione e sviluppo ed ha lo scopo di mostrare come sono stati realizzati nello specifico alcuni componenti
- Il quinto capitolo conclude il lavoro ed espone i possibili ampliamenti futuri.

Indice

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"[10], 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 fisheye[1] ed una di "overview + detail". L'esperimento ha evidenziato come diffe-

[1]Nell'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

renti 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 complessivamente inferiori rispetto gli 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)[15].

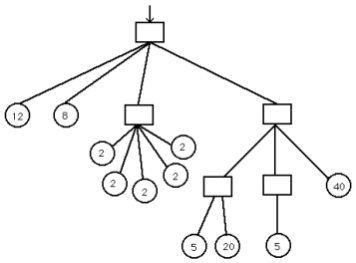
Una tra le tecniche alternative di visualizzazione più popolari è il Treemap[18], una tecnica "slice-and-dice" che riempie lo spazio a disposizione sulla base di una pianta rettangolare. Ogni nodo dell'albero è rappresentato da un rettangolo che contiene ulteriori rettangoli annidati tra loro, rappresentando 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[12], Triangular Aggregated Treemap[7], Quantum Treemap[6], Cascaded Treemap[14], ecc.).

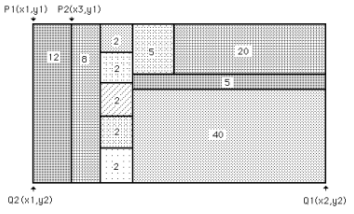
In questo elaborato, verrà data maggiore enfasi ad un'altra tecnica per la visualizzazione gerarchica di articoli, ossia la tecnica Sunburst[20], utilizzata in DocuDipity.

Un sommario di tutte le metodologie e approcci proposti negli ultimi 30 anni è descritto in [17].

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

essere inizialmente non leggibili, ma possono essere rese comprensibili cliccando su di esse con il mouse

1.2 DocuDipity: introduzione e concetti principali

L'intero lavoro è stato realizzato sulla base di un sistema già esistente chiamato DocuDipity[9]

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"[19].

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[13] 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[8], il cui algoritmo ha l'obiettivo di individuare un insieme minimale di elementi strutturali in grado di esprime 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.

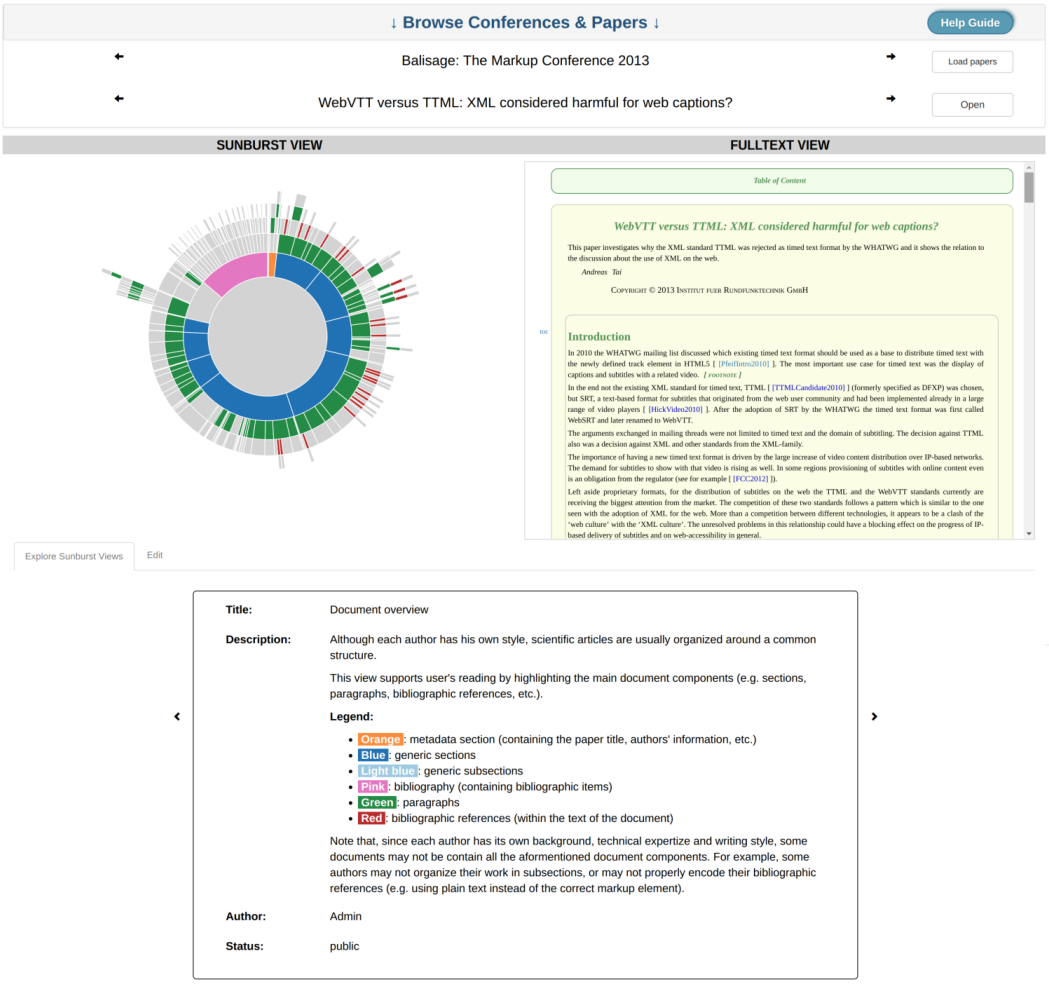


Figura 1.2: Una overview della precedente interfaccia DocuDipity

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.

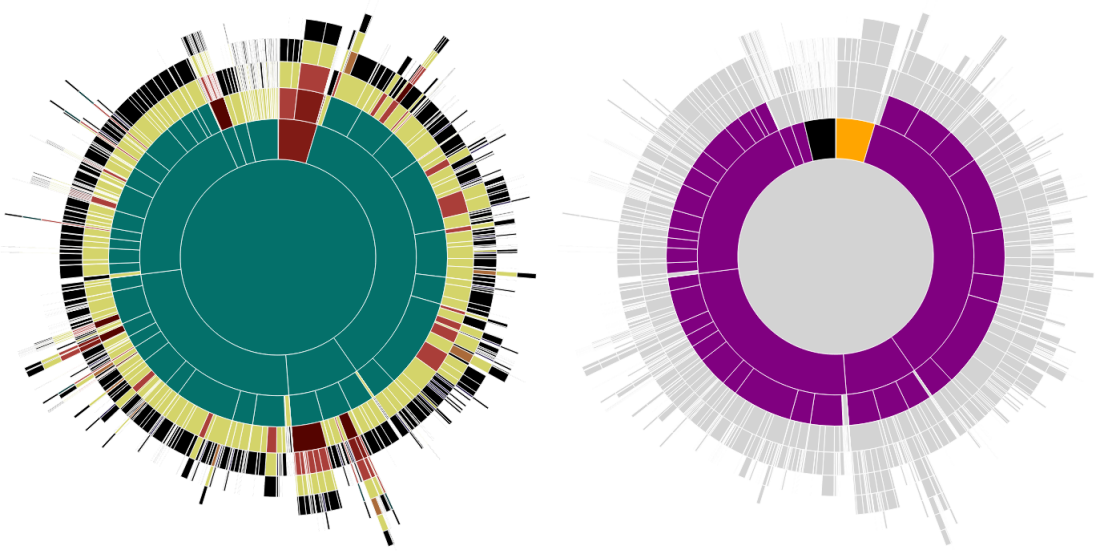


Figura 1.3: Esempi di visualizzazioni SunBurst

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 viene evidenziato nel SunBurst 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, cioà 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

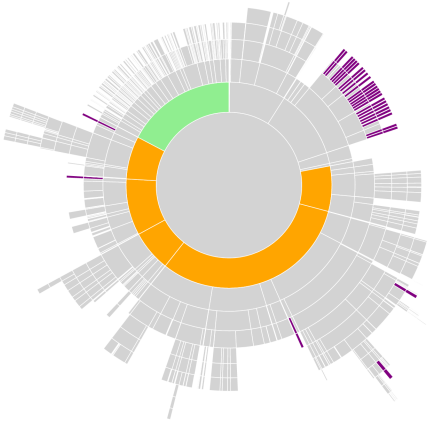
In questa sezione mostreremo le potenzialità offerte dal tool DocuDipity per analizzare documenti scientifici. In questa presentazione sono stati considerati alcuni articoli scientifici presentati al convegno Balisage Markup Conference che dal 2008 mette a disposizione gli articoli in formato XML. Le regole utilizzate dalle seguenti 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

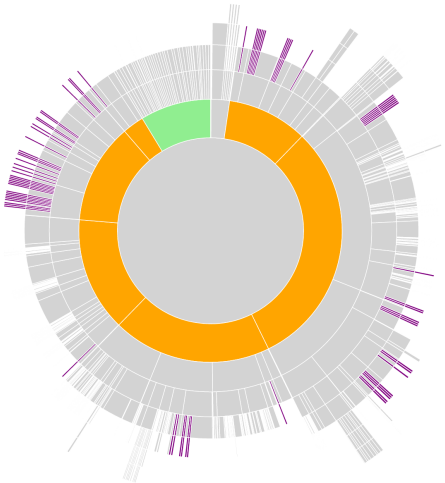
1.3.1 Posizione citazioni

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:

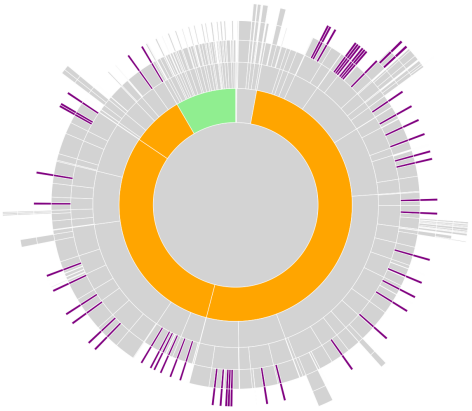
- se l'autore vuole introdurre al lettore il background necessario per comprendere l'argomento trattato nell'articolo, le citazioni tendono ad essere raggruppate nell'introduzione, come mostrato nella Figura 1.5a
- invece, se l'autore vuole confrontare il proprio lavoro con articoli che trattano argomenti 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

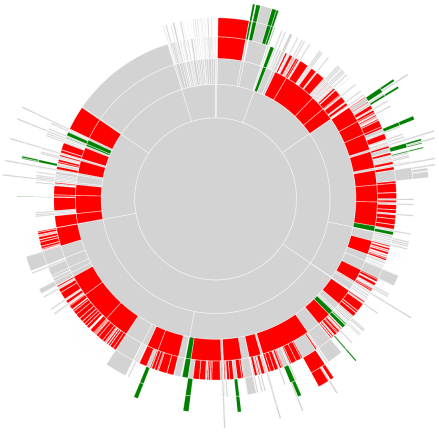
Figura 1.5: Rappresentazioni SunBurst che evidenziano differenti strutture di articoli scientifici

1.3.2 Stili di scrittura

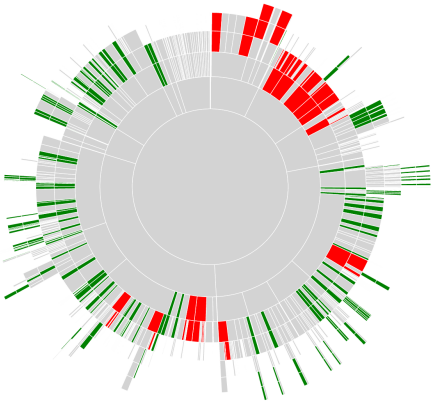
Un'altra analisi che si potrebbe fare su un articolo è indagare sullo stile e sul linguaggio utilizzato da/dagli autori dell'articolo. L'idea che sta dietro a questa analisi è che esistono differenti stili di scrittura che possono variare da persona a persona, che vengono nascosti dalla visualizzazione dettagliata ipertestuale ma che con l'utilizzo della vista SunBurst si possono con maggior facilità scoprire e 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 rossi/verdi i paragrafi con una lunghezza superiore/inferiore alla media. L'applicazione di questa regola, sempre 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 1.6c) e chi invece tende a essere più prolisso e sviluppare il pensiero con frasi lunghe (come mostrato nella Figura 1.6a)

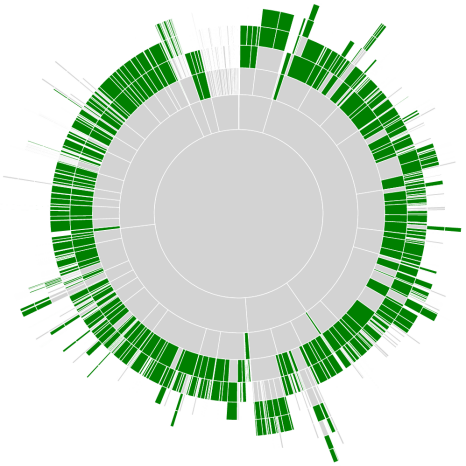
- nel caso di articoli con più autori permette di evidenziare a quali parti un autore ha contribuito, come mostrato nella Figura 1.6b, in quanto emerge il proprio stile che differisce da quello degli altri autori



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



(b) Articolo scritto da più autori con differenti lunghezze dei paragrafi



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

Figura 1.6: Rappresentazione stili autore

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 architetturale 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 aprili (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

Analisi dei requisiti e design

Durante questa fase sono stati individuati gli obiettivi e i requisiti del progetto, è stata definita una specifica terminologia per far riferimento ad ogni entità facente parte del nuovo sistema di DocuDipity e progettati una serie di prototipi sulla base dei casi d'uso identificati. Ad ogni iterazione successiva i prototipi sono stati affinati, partendo da dei semplici schizzi su fogli di carta fino ad arrivare a mockup realizzati con il software Balsamiq[2].

2.1 Gli obiettivi del progetto

Gli obiettivi sono stati inizialmente identificati sulla base delle limitazioni e criticità riscontrate nella precedente interfaccia DocuDipity (vedi capitolo 1.4: Limiti e Criticità). Il sistema doveva essere completamente rivisto, non solo dal punto meramente ingegneristico, doveva essere riprogettata sia la parte front-end di cui mi sono occupato personalmente, sia la parte di back-end che è stata realizzata da un altro collega universitario in concomitanza alla mia. La necessità era quella di progettare e sviluppare un sistema, che fosse il più flessibile possibile, basato su un'architettura plugin-based, che permettesse di gestire più articoli contemporaneamente, anche appartenenti ad una o più collezioni diverse e non più ad unico gruppo come accadeva nella precedente versione, in modo che l'utente potesse mettere a confronto la visualizzazione di articoli appartenenti a gruppi differenti.

La progettazione del front-end doveva quindi garantire il rispetto dei seguenti principi:

- **filtering per overview**: possibilità di selezionare i documenti da analizzare, da una o più collezioni esistenti
- **overview**: applicare una vista ai documenti da analizzare, iniziando da una vista a "mosaico" in cui viene applicata la stessa vista a tutti i documenti, che può essere

una vista di preview con i dati riassuntivi del documento, o una vista sunburst per un confronto diretto

- **zoom**: applicare viste coordinate su un documento selezionato, attraverso SunBurst e la vista testuale

L'applicazione doveva inoltre essere responsive, per gestire al meglio il ridimensionamento delle varie viste, in modo da poterci lavorare anche su tablet. L'obiettivo era quello di avere a disposizione una soluzione solida, facilmente comprensibile, che favorisse la manutenibilità (convenzioni, refactoring, estendibilità) ed il riuso del codice, per permettere a futuri programmatori di lavorarci nella maniera più agevole possibile.

2.2 La terminologia del progetto

In concomitanza alla definizione dei requisiti, è stata definita una specifica terminologia da utilizzare per identificare le varie entità del sistema DocuDipity. La terminologia che si è deciso di utilizzare per descrivere il modello concettuale di DocuDipity deriva da un modello esistente chiamato FRBR[23]. Si tratta di uno schema concettuale sviluppato dalla International Federation of Library Associations and Institutions (IFLA), realizzato tramite modello entità-relazione allo scopo di dare una rappresentazione semiformale alle informazioni bibliografiche. L'obiettivo di FRBR è di elaborare un modello concettuale che permetta di individuare i requisiti essenziali del record bibliografico, definendone le modalità di struttura e le finalità. Le FRBR sono il risultato di un'analisi che parte dai bisogni dell'utente, inteso non soltanto come lettore o personale delle biblioteche ma anche come editore, distributore, rivenditore ecc. Questa analisi ha permesso di distinguere le entità fondamentali per l'utente, gli attributi di tali entità e le relazioni tra le stesse. Sono stati presi in prestito due vocaboli specifici da questo schema: il concetto di Expression, ossia la realizzazione di un'opera, intesa come creazione intellettuale, e il concetto di Manifestation, per far riferimento alla materializzazione di un'espressione.

Viene descritta di seguito tutta la terminologia utilizzata per la realizzazione del modello dati ed un'immagine in cui viene mostrato il relativo diagramma (Figura 2.1):

- **User**: utente che carica dei DocumentSet e lavora su dei Projects.

- **DocumentSet**: un insieme di documenti caricati nel sistema, caratterizzati da alcune proprietà. Un DocumenSet contiene a sua volta DocumentSet oppure dei Document (Expression). In questo modo possiamo avere ad esempio un DocumentSet per un determinato gruppo, che contiene a suo volta altri DocumentSet appartenenti a sottogruppi, ciascuno dei quali contiene i singoli articoli (Document Expression) appartenenti a quel sottogruppo.

- **Document (Expression)**: un documento singolo, caratterizzato da alcune proprietà (titolo, anno di pubblicazione, autore). Può esistere in vari formati e generare quindi diverse manifestazioni (Document Manifestation). Le manifestazioni hanno lo stesso contenuto ma in formati diversi, utili per generare viste diverse
- **Document (Manifestation)**: la versione concreta di un documento in uno specifico formato
- **Schema**: lo schema (JSON, XML), con cui un Document (Manifestation) viene memorizzato
- **Project**: sessione di analisi fatta su un gruppo di Document (Expression) attraverso una serie di View. Anche il Project è caratterizzato da alcuni metadati (es. descrizione, keywords, ecc.). I Document nel Project sono caricati da DocumentSet.
- **Project View**: View che fornisce una overview di tutti gli articoli presenti nel Project
- **ViewType**: tipi di views (es. SunBurst, TreeView, Hypertext)
- **View**: ViewType applicate a uno o più Document (Manifestation). Sono quindi le viste attualmente aperte nell'interfaccia di DocuDipity per il Project corrente.
- **Document View**: View che può essere applicata ad un solo Document (Manifestation), es. il Sunburst applicato ad uno specifico articolo
- **Collection View**: View che può essere applicata ad un sottoinsieme di Document selezionati dall'utente tra quelli appartenenti al Project corrente.
- **Sinchronized**: View sincronizzata ad un'altra View
- **View Conf/Skin**: CSS differenti per la stessa View

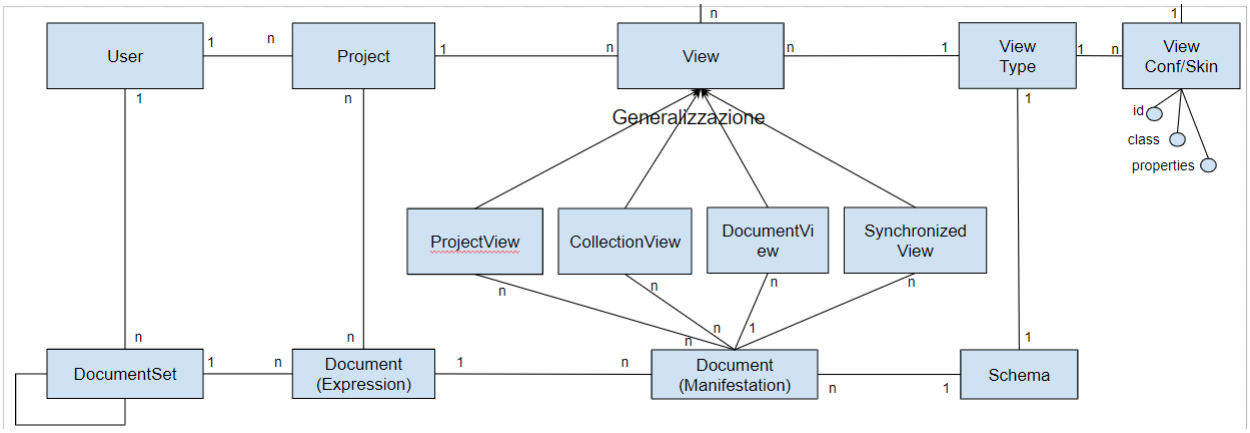


Figura 2.1: Modello dati DocuDipity

2.3 I requisiti dell'applicazione

Sulla base degli obiettivi sono stati definiti i requisiti dell'applicazione. Rispetto alla precedente interfaccia DocuDipity è stata introdotta una separazione tra il concetto di **area di lavoro** e il concetto di **progetto**.

L'utente deve poter visualizzare tutti i suoi progetti, e scegliere tra questi su quale voler iniziare un lavoro di analisi. Il lavoro di analisi consiste nel selezionare gli articoli presenti nel progetto, eventualmente importandone dei nuovi all'interno, per metterli a confronto con la stessa visualizzazione ma di articoli diversi o con lo stesso articolo ma in una visualizzazione diversa. Al termine del lavoro di analisi l'utente deve poter salvare la configurazione, per poterla riaprire in un secondo momento.

Le funzionalità previste per DocuDipity, assumendo che l'utente parta da un set di documenti già caricati sul server, sono raccolte nei seguenti punti:

- L'utente vede l'elenco dei progetti
- L'utente crea un nuovo progetto
- L'utente apre un progetto esistente
- L'utente cerca documenti da aggiungere al progetto (importa documenti nel progetto)
- L'utente aggiunge/elimina documenti dal progetto
- L'utente seleziona alcuni documenti e visualizza i metadati principali dei singoli documenti

- L'utente elimina qualche documento dalla visualizzazione
- L'utente aggiunge altri documenti
- L'utente sceglie una visualizzazione diversa sulla stessa collezione, ad esempio un mosaico di SunBurst
- L'utente apre un singolo documento e vede un SunBurst interattivo e più dettagliato
- L'utente visualizza lo stesso documento con una view diversa (es. View ipertestuale)
- L'utente visualizza lo stesso documento con due view sincronizzate, ad esempio SunBurst e View ipertestuale
- L'utente confronta il SunBurst di due documenti
- L'utente aggiunge uno o più documenti al confronto, usando sempre SunBurst
- L'utente seleziona altri documenti e li apre in un nuovo mosaico ma con una view diversa
- L'utente salva il progetto per poterlo nuovamente aprire e/o condividere con altri studiosi

2.4 I Wireframe

I Wireframe sono illustrazioni organizzative schematiche dei contenuti presenti nel progetto che stiamo realizzando. La funzione principale dei wireframe è comunicare l'idea del progetto, focalizzando l'attenzione sull'architettura piuttosto che sul design. Rappresenta una base per il prototipo da sviluppare e rende bene l'idea della consistenza del progetto. La fase di realizzazione dei Wireframe è fondamentale per gettare le basi di una buona user experience, infatti permette di:

- Fare concentrare cliente e committente sulla struttura e la fruibilità dei contenuti
- Capire come l'utente interagisce con l'interfaccia
- Non mescolare le fasi del processo di costruzione dell'interfaccia
- Focalizzarsi su funzionalità e obiettivi di un ambiente web, prima che sulla sua vestizione grafica (il progettista può pianificare il layout e l'interazione di un'interfaccia senza preoccuparsi di colori, scelte di carattere tipografico o testi)

La struttura dell'interfaccia è stata realizzata inizialmente attraverso un Wireframe cartaceo, per poi procedere alla successiva raffinazione di questo in diverse iterazioni attraverso il software Balsamiq Wireframes.

Balsamiq Wireframes è un tool grafico per sviluppatori, designer e progettisti che permette di realizzare schizzi di interfacce utente e schermate (wireframe) per siti web e applicazioni. Grazie a questo tool, è stato testato il flusso di interazioni dell'utente definito nel capitolo precedente.

Come già accennato, sono state create piú versioni dei vari Wireframe durante la fase di progettazione, a cui sono state apportate di volta in volta diverse modifiche. Le prime versioni realizzate per DocuDipity sono illustrate in Figura 2.2 e Figura 2.3.

La Figura 2.2 mostra la maschera per la gestione dell'area di lavoro all'interno di un progetto.

In questa prima iterazione non è chiara la distinzione tra il progetto che l'utente può aprire/creare e la parte di effettiva lettura ed analisi degli articoli presenti nel progetto.

Nella parte sinistra della maschera era stato previsto un componente che permettesse all'utente di visualizzare gli articoli presenti nel progetto ed eventualmente importarne dei nuovi attraverso il popup 1 o alternativamente con il popup 3 di Figura 2.3, che rappresenta una modifica effettuata successivamente al popup 1.

Nel popup 3 veniva fornita la possibilità all'utente di fare un "pre-filtraggio" delle collezioni di articoli prima di importarle all'interno del componente dedicato alle collezioni.

La parte destra era stata progettata per consentire all'utente di visualizzare i progetti, eliminarli, crearne dei nuovi, ed eventualmente permettere attraverso il popup 2 di cambiare la configurazione con cui visualizzare gli articoli all'interno della parte centrale.

La parte centrale era invece progettata per fornire una prima preview degli articoli presenti nel progetto selezionato, mostrando una serie di metadati (titolo, autore, formato, ecc.).

Questa prima iterazione, era ancora molto instabile, non era presente una netta distinzione tra progetto e area di lavoro e l'utente non era in grado di organizzare in maniera efficiente le varie collezioni presenti nel progetto.

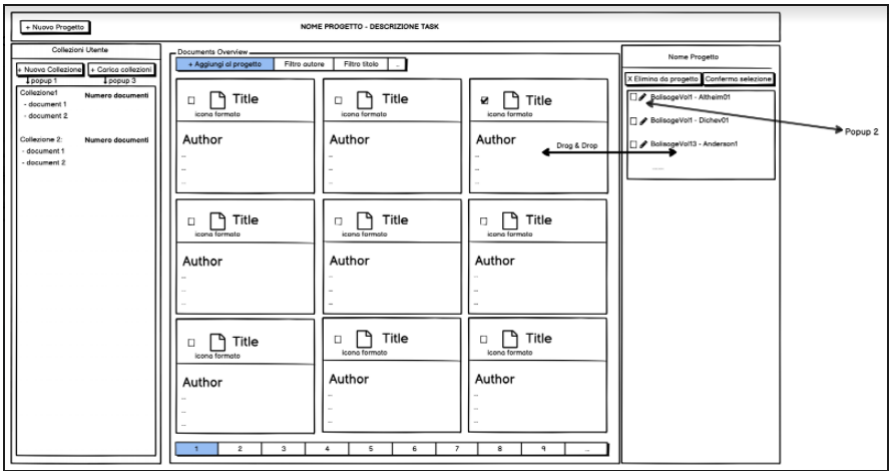


Figura 2.2: Wireframe maschera filtering-overview - prima iterazione

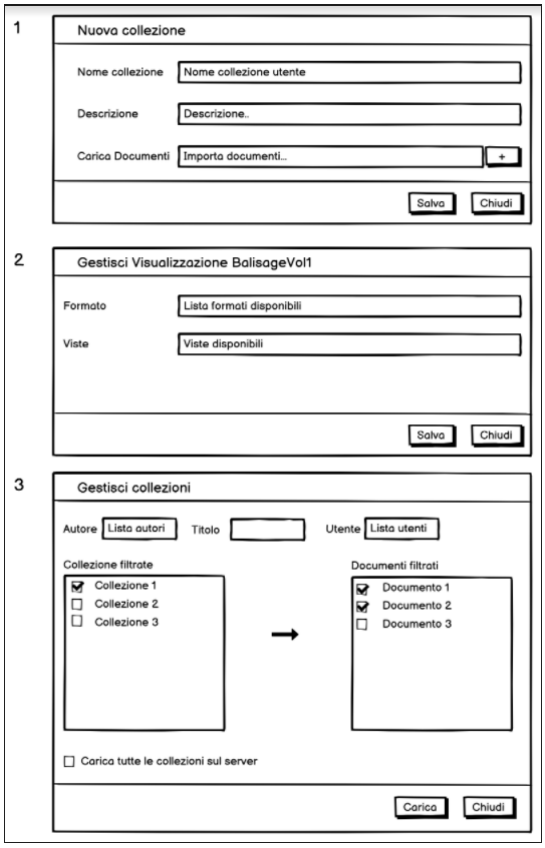


Figura 2.3: Wireframe popups - prima iterazione

Nelle versioni successive si è fatta chiarezza circa le instabilità evidenziate nel precedente prototipo.

Come possiamo notare in Figura 2.4, è stata aggiunta una nuova maschera dedicata alla gestione dei progetti, completamente separata dall'area di analisi dei documenti. Da questa maschera l'utente può creare/modificare un progetto (attraverso il popup di Figura 2.4), eliminarlo, ed aprirlo attraverso il bottone play (posizionato sulla riga della griglia), che porterà l'utente all'area di analisi illustrata in Figura 2.5 e in Figura 2.6.

Nella versione del Wireframe prevista dalla Figura 2.5, la parte destra della maschera che veniva precedentemente utilizzata per gestire i progetti è stata completamente rimossa per essere sostituita dall'attuale parte sinistra, ossia la Project View, che fornisce una overview degli articoli presenti nel progetto aperto.

Dalla Project View è possibile filtrare gli articoli, importarne dei nuovi mantenendo il popup 3 illustrato in Figura 2.3, salvare lo stato del progetto e gestire gli articoli che si vuole analizzare nella parte destra (Collection View), grazie ad un menù contestuale che permette di aggiungere gli articoli selezionati alla Collection View corrente (tab selezionato), ad una nuova Collection View (nuovo tab), o aprirli direttamente in una Document View (Figura 2.7), per ottenere il documento in una specifica visualizzazione (Hypertext o Sunburst).

La Collection View sulla destra è ora meglio organizzata grazie all'utilizzo di tab, ciascuno dei quali è in grado di fornire una rappresentazione sia tabellare (Figura 2.5) che a mosaico (Figura 2.6), di un sottoinsieme di documenti selezionati dalla Project View. All'interno dei tab è prevista la possibilità di rimuovere gli articoli dalla Collection View, aprirli nella Document View attraverso l'icona a forma di occhio, duplicare gli articoli selezionati in un nuovo tab e modificare il titolo del tab (icona della matita).

Nella Collection View a mosaico, oltre alla ComboBox integrata all'interno dei singoli visualizzatori che consente di cambiare vista al singolo documento, ne è stata prevista anche un'altra per permettere all'utente di cambiare vista contemporaneamente a tutti i documenti che sono stati selezionati e aperti dalla Project View.

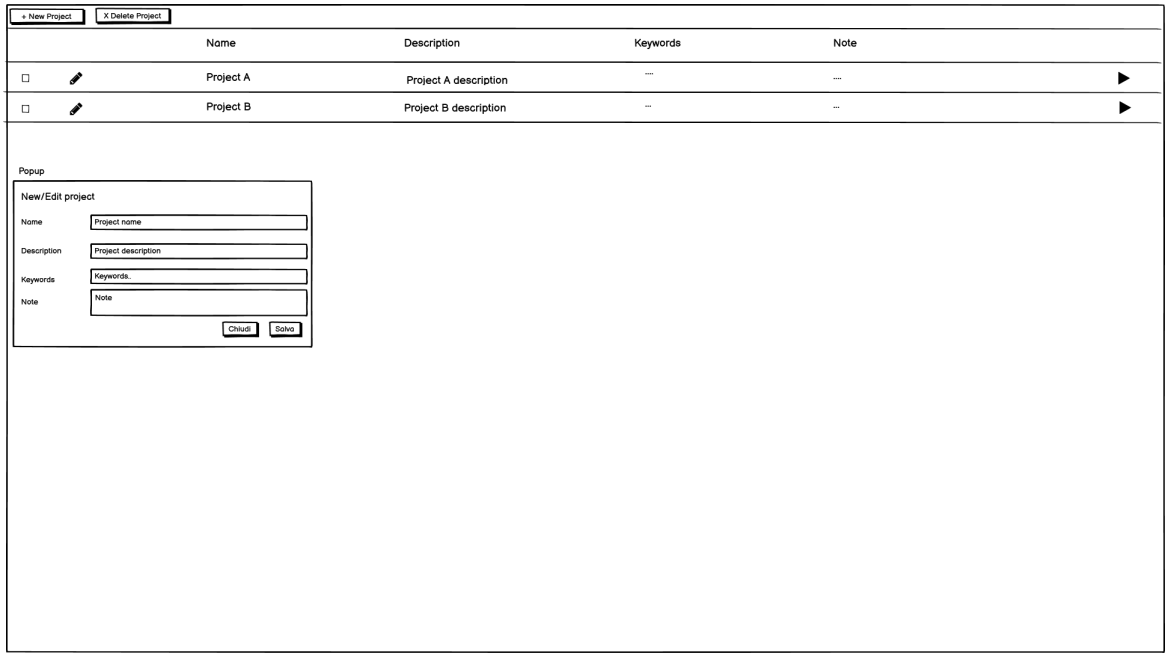


Figura 2.4: Wireframe maschera gestione progetti

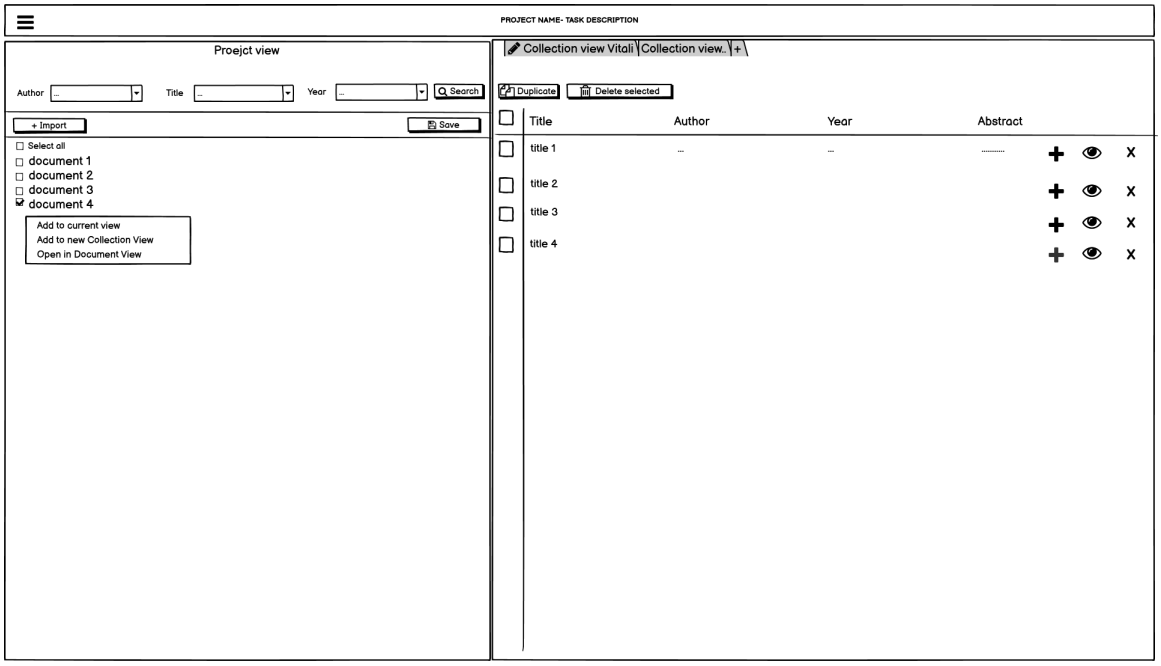


Figura 2.5: Wireframe maschera per gestione area di lavoro - view tabellare

Figura 2.6: Wireframe maschera per gestione area di lavoro - view mosaico

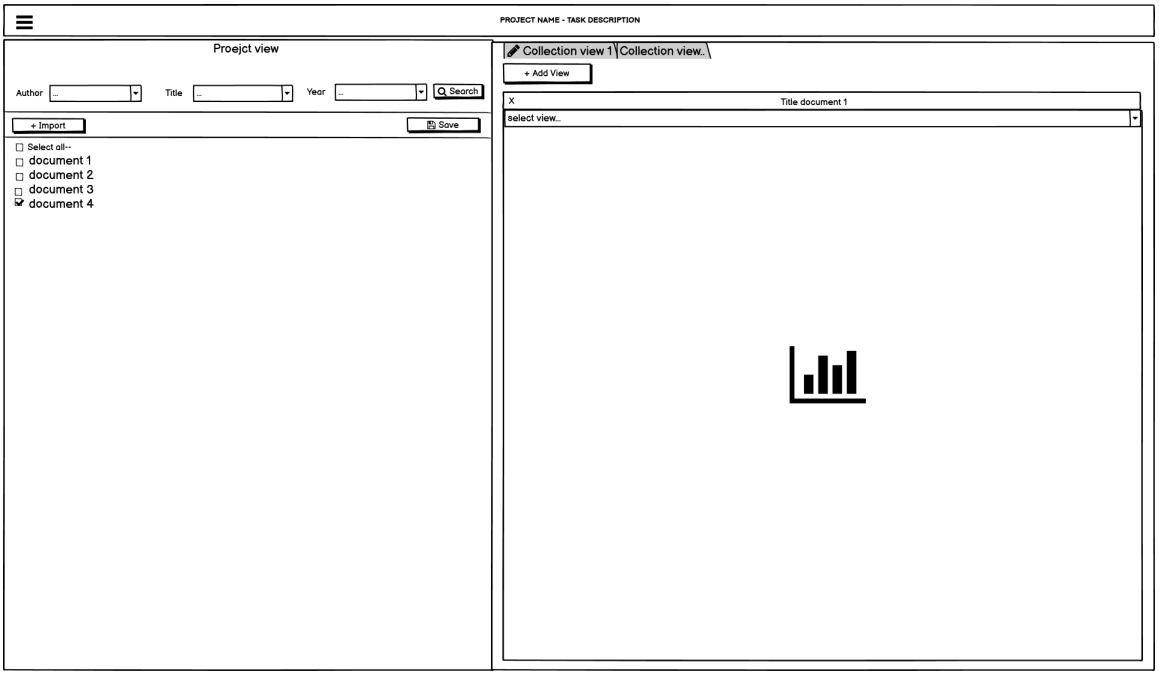


Figura 2.7: Wireframe maschera per gestione area di lavoro - document view

In Figura 2.8 e in Figura 2.9 vengono presentati i Wireframe finali di DocuDipity, realizzati nell'ultima iterazione e relativi rispettivamente alla maschera dell'area di lavoro e al popup di configurazione delle viste.

Sono state apportate una serie di modifiche rispetto alla versione precedente, tra le principali troviamo:

- Nella Collection View è stato aggiunto un bottone (icona ingranaggio) che permette di aprire un popup di configurazione (Figura 2.9) per gestire titolo, descrizione e vista di default dei documenti, ed una serie di filtri per organizzare i documenti nella view sottostante.

- Nella parte inferiore è stata integrata la Document View, ossia la parte di visualizzazione concreta dei singoli articoli.

  Nella Document View è stata prevista la possibilità di aprire più articoli per metterli a confronto, cambiare vista sull'articolo selezionandone una tra quelle disponibili mostrate nella ComboBox e selezionare l'articolo visualizzato nella corrispondente vista tabellare o mosaico attraverso un click sulle frecce situate nel bottone posizionato in alto a sinistra all'interno dei visualizzatori, che consente di scorrere gli articoli verso il basso o verso l'alto, cambiando la vista nella Document View ogni volta che ci si muove da un documento all'altro.

  All'interno di ogni componente della Document View è stato introdotto un meccanismo di "bloccaggio" delle viste e dei documenti.

  Il lucchetto diventa selezionabile solo quando sono presenti valori uguali, stesse viste o stessi documenti. L'idea è quella di ottenere su coppie di documenti diversi un confronto con lo stesso tipo di visualizzazione da entrambe le parti, mentre nel caso di coppie di documenti uguali, si vuole invece ottenere un confronto con visualizzazioni differenti. Ad esempio, in Figura 2.8, viene confrontata la stessa visualizzazione su documenti diversi, bloccando con il lucchetto la vista SunBurst a destra, in modo che quando l'utente seleziona dalla ComboBox a sinistra una nuova vista questa viene cambiata anche a destra.

  Lo stesso comportamento si verifica quando l'utente vuole invece confrontare visualizzazioni diverse dello stesso documento, in questo caso deve essere bloccato il documento nella parte destra, in modo che quando viene selezionata dalla ComboBox a sinistra un nuovo documento, questo viene cambiato anche nella parte destra.

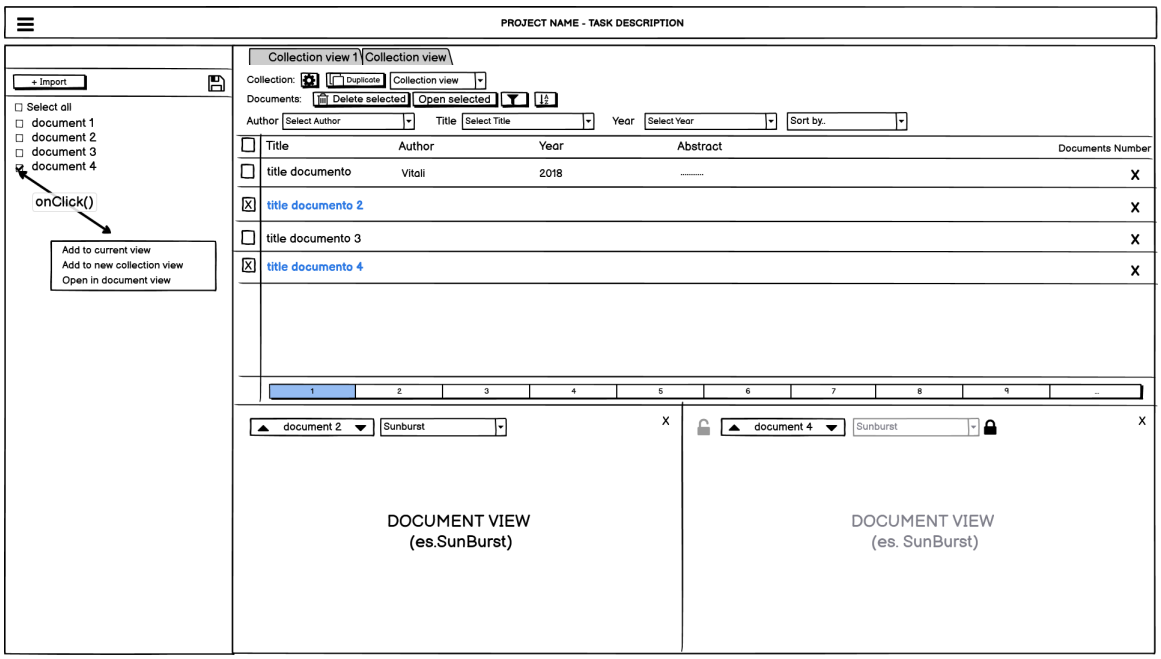


Figura 2.8: Wireframe GUI ultima iterazione DocuDipity

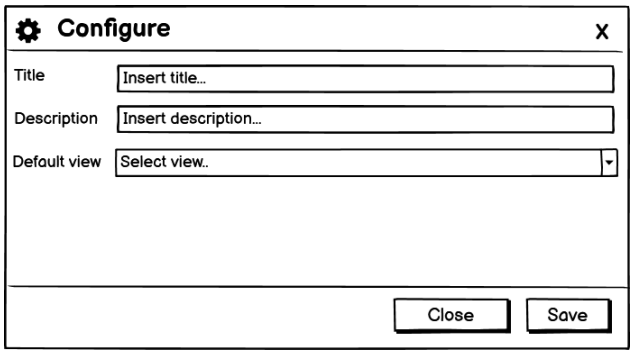


Figura 2.9: Wireframe popup configurazione viste

2.5 Un confronto con l'interfaccia finale

Nel capitolo precedente sono state descritte le tappe salienti del percorso di realizzazione dei Wireframe che hanno portato alla realizzazione dell'interfaccia, in questo vengono illustrate due immagini in cui è mostrato il risultato finale.

L'interfaccia è stata realizzata seguendo come linea guida uno dei principi cardine della rappresentazione visiva di informazioni, ossia "Overview first, zoom and filter, then details-on-demand", discussa in dettaglio nel capitolo 1.1.

Come possiamo notare in Figura 2.10 ed in Figura 2.11, questa nuova versione permette all'utente di avere una prima visualizzazione di tutti gli articoli scientifici presenti nel progetto (*Overview*). L'utente può selezionare da questo set gli articoli che gli suscitano maggiore interesse e aggiungerli al componente situato nella parte destra dell'interfaccia (*Zoom*), da cui può anche applicare dei filtri (*Filter*). Una volta aggiunti all'area di lavoro gli articoli di interesse, l'utente può selezionarne alcuni per aprirne una visualizzazione concreta attraverso i componenti posizionati nella parte inferiore dell'interfaccia (*Details-on-demand*).

Le parti dell'interfaccia da completare in future iterazioni, verranno trattate nel capitolo conclusivo dell'elaborato.

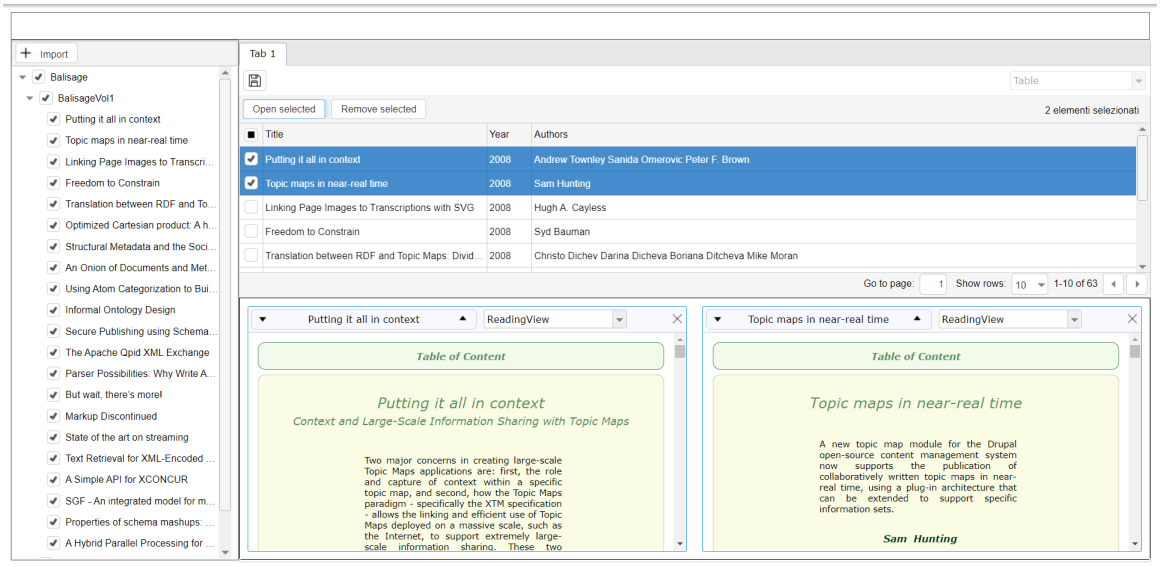


Figura 2.10: Interfaccia DocuDipity in versione Desktop

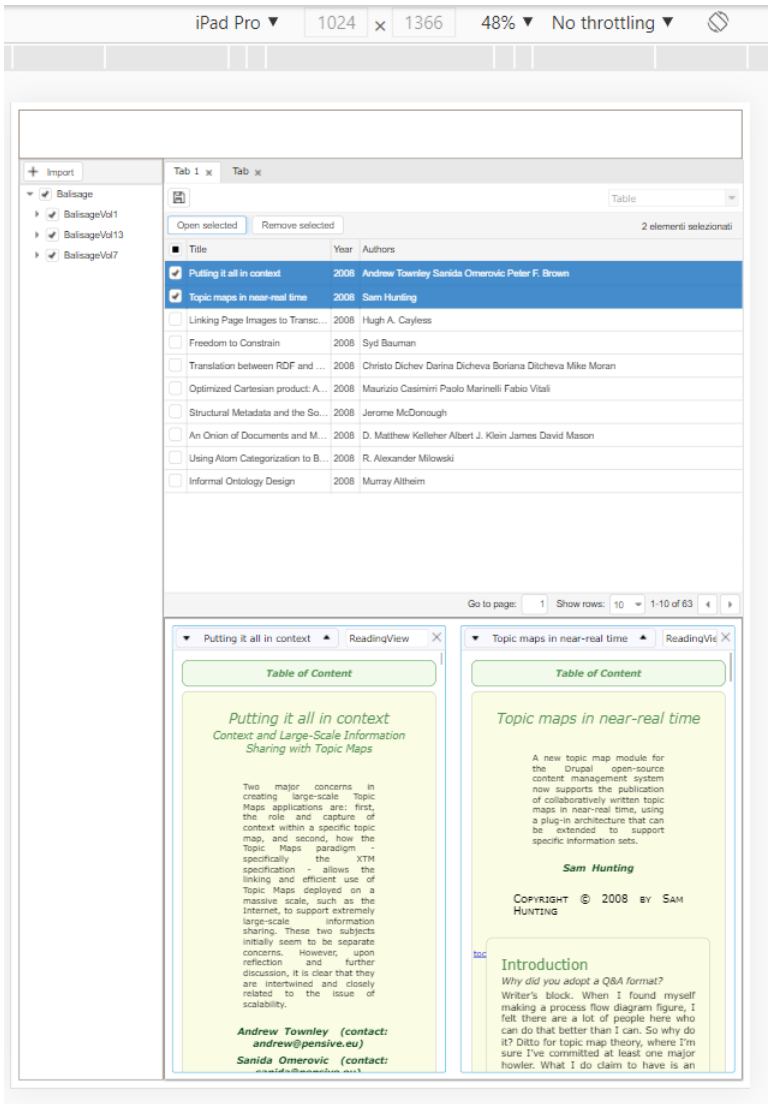


Figura 2.11: Interfaccia DocuDipity in versione Tablet

Capitolo 3

L'architettura modulare per il progetto DocuDipity

In questo capitolo viene descritta l'architettura del sistema progettato e realizzato nell'ambito del progetto DocuDipity, partendo da un'analisi delle scelte progettuali che sono state effettuate e soffermandosi sulla fase di realizzazione del layout responsive.

Il sistema creato si appoggia ad una parte di back-end[11] (sviluppata in concomitanza alla mia da un altro collega universitario), che mette a disposizione una serie di API REST per interagire con l'ambiente da me realizzato.

La progettazione di API server side, ha permesso la creazione di un'architettura capace di sopperire alle carenze e criticità identificate nella precedente versione, assicurando una netta separazione tra client e server, evitando in questa maniera qualsiasi sovrapposizione di competenze, come accadeva precedentemente.

3.1 Analisi e valutazione dei framework

Prima di procedere alla pura fase implementativa è stata fatta una valutazione dei framework da utilizzare per lo sviluppo della nuova interfaccia di DocuDipity. Fin da subito, la ricerca del framework è stata fatta tenendo conto di una serie di caratteristiche principali che dovevano essere necessariamente presenti:

- Widget in grado di ricercare, ordinare, filtrare e visualizzare articoli, anche in maniera gerarchica
- Widget responsive
- Widget facilmente estendibili

- Supporto dalla maggior parte dei browser
- Licenza free di utilizzo
- Compatibilità con Angular, React, Vue e Web Components

Dati i prerequisiti, era di fondamentale importanza l'appoggio ad un framework che mettesse a disposizione widget come griglie e treeview.

3.1.1 Le alternative

Durante il processo di valutazione sono stati identificati i seguenti plugin/framework:

- Ng-bootstrap[4]
- Ag-Grid[1]
- jQWidgets[3]

A seguito di alcuni test eseguiti sui widgets e in base alla compatibilità con i prerequisiti, la scelta si è concretizzata verso jQWidgets.

L'altro aspetto fondamentale da valutare è stato quello relativo all'identificazione della tecnologia da sfruttare per realizzare un componente che fosse in grado di fornire una visualizzazione ipertestuale degli articoli, per metterli a confronto e analizzarli. Il componente doveva essere riutilizzabile (indipendentemente dal framework) ed estendibile. Per quest'ultimo caso la scelta si è invece indirizzata verso la tecnologia dei Web Components.

3.1.2 La scelta: jQWidgets

jQWidgets è un framework jQuery utilizzato per creare GUI responsive che mette a disposizione una vasta libreria di widgets. Tra i vari widgets utilizzati per il rinnovamento della GUI di DocuDipity troviamo:

- *jqxTree*: utilizzato per gestire la struttura gerarchica degli articoli presenti sul server. Gli articoli sono organizzati in set di documenti (conferenze), ciascuna dei quali può contenere altri set (volumi), che a loro volta contengono i vari articoli che è possibile analizzare
- *jqxGrid*: utilizzata per visualizzare gli articoli (con una serie di metadati) che vengono selezionati e aggiunti a partire dalla struttura ad albero creata utilizzando il jqxTree. Attraverso la griglia è possibile ricercare, ordinare, selezionare vari articoli ed aprire la loro Document View (visualizzazione ipertestuale)

- *jqxTabs*: utilizzati per organizzare le Collection View (View applicata ad un sottoinsieme di documenti selezionati dal jqxTree), e le Document View relative ai singoli articoli
- *jqxButton*: utilizzati per gestire le interazioni con i vari componenti dell'interfaccia
- *jqxComboBox*: utilizzati per gestire la scelta delle View da applicare.

Ciascuno di questi componenti è stato incapsulato in classi separate ed integrato con funzionalità aggiuntive (vedi capitolo: 4.1 I widget jQWidgets).

Si è deciso di optare per questo framework per una serie di ragioni, tra le principali c'è sicuramente l'ampia varietà di widgets che vengono messi a disposizione, la garanzia di sostenibilità a lungo termine grazie all'utilizzo degli ultimi standard W3C Custom Element, l'elevato grado di personalizzazione e di adattabilità dei widgets su diversi schermi e risoluzioni, la possibilità di integrare ed utilizzare i widgets nella maggior parte dei framework o CMS (Joomla, WordPress, Knockout, AngularJS, Breeze, RequireJS, Twitter Bootstrap, Vue, ecc..), e il supporto alla maggior parte dei browser Mobile e Desktop (Internet Explorer 7.0+, Firefox 2.0+, Safari 3.0+, Opera 9.0+, Google Chrome, IE Mobile, Android, Opera Mobile, Mobile Safari)

3.2 Tecnologie utilizzate

3.2.1 TypeScript

Il TypeScript è un superset di JavaScript, sviluppato da Microsoft, che permette di sviluppare con il paradigma Object-oriented in modo molto simile a Java e C. La scelta di utilizzo del TypeScript per la realizzazione dell'interfaccia utente è stata dettata da una serie di vantaggi che questo linguaggio porta con sé, tra i più importanti troviamo la tipizzazione delle variabili che consente di ridurre gli errori, la facilitazione di lettura del codice e delle fasi di test ed il supporto a classi di oggetti, interfacce ed ereditarietà. Il codice scritto in TypeScript viene transpilato in codice JavaScript, segnalando eventuali errori in fase di scrittura del codice. Il TypeScript aggiunge un pò di complessità al setup iniziale del progetto ma ripaga con la produzione di un codice finale più solido.

3.2.2 jQuery

jQuery è una libreria JavaScript open source che semplifica la manipolazione dell'albero DOM HTML, la gestione degli eventi, l'uso di funzionalità AJAX e la manipolazione del CSS, ed è attualmente la libreria più utilizzata in internet. Secondo i report pubblicati da W3Techs[5], questa libreria è presente nel 77.8% di tutti i siti web. La scelta

di utilizzo di jQuery all'interno del progetto è stata dettata dal fatto che questa libreria ha permesso di facilitare l'astrazione delle interazioni a basso livello con l'HTML, semplificando la creazione dell'interfaccia e rendendo il codice più ordinato e leggibile.

3.2.3 Web components

Web Components è una suite di tecnologie che permettono di creare elementi personalizzati, la cui funzionalità è incapsulata e separata dal resto del codice sorgente. I Web Components hanno come obiettivo quello di risolvere il problema del riutilizzo di porzioni di codice all'interno dell'applicativo.

La scelta di utilizzare i Web Components e di creare degli standard per la realizzazione dei vari componenti della nuova interfaccia DocuDipity, permette ai futuri sviluppatori di apprendere più velocemente il codice sviluppato e di integrarli in maniera più agevole in un eventuale nuovo framework.

3.3 Il layout

Nei seguenti due sottocapitoli verranno analizzate e motivate le scelte che hanno portato alla realizzazione dell'attuale layout di DocuDipity, con relative immagini dei template responsive progettati per la versione Desktop (Figura 3.1) e per la versione Tablet (Figura 3.2).

3.3.1 Lo stile del progetto - HTML e CSS

Il design dell'intera interfaccia è stato realizzato senza l'ausilio di framework come Bootstrap, Angular Material UI, ecc. Il motivo di questa scelta è principalmente legato al fatto di voler creare un'interfaccia che fosse più personalizzabile rispetto a quella prodotta attraverso un framework CSS, svincolando il design da regole specifiche, mantenendo l'interfaccia leggera e il meno dipendente possibile da fonti esterne, oltre a quella di jQWidgets utilizzata per la realizzazione dei componenti. Per un progetto articolato come DocuDipity, l'impiego di un framework CSS avrebbe implicato l'utilizzo massiccio di elementi di markup puramente strutturali inseriti direttamente nell'HTML, finendo per sporcare il codice.

3.3.2 CSSGrid e Flexbox

Come supporto per il responsive design è stato utilizzata una combinazione tra CSSGrid e Flexbox.

Il CSSGrid layout nasce per far fronte ad un utilizzo eccessivo dei framework, e ha il vantaggio di funzionare da supporto per la creazione di layout bidimensionali, a differenza di Flexbox che permette di gestire strutture monodimensionali. Con CSSGrid è stato definito il layout generale della pagina web mentre con Flexbox si è delineato il modo in cui gli elementi si comportano all'interno di ogni area della griglia. Di seguito vengono mostrati i template dell'interfaccia realizzati con CSSGrid.

In Figura 3.1 ed in Figura 3.2 è mostrato un layout desktop ed un tablet per la gestione dell'area di lavoro all'interno di un progetto.

Partendo da sinistra troviamo un primo container chiamato "Sidebar" in cui è stata implementata la Project View, per visualizzare e gestire tutti gli articoli presenti nel progetto, procedendo verso la parte centrale troviamo un altro container chiamato Collection View in cui sono state implementate le viste per gestire il sottoinsieme di articoli che l'utente seleziona dalla Project View, ed infine nel container in basso chiamato Document View, sono stati implementati i componenti che si occupano della visualizzazione concreta dei singoli documenti aperti dall'utente nella Collection View.

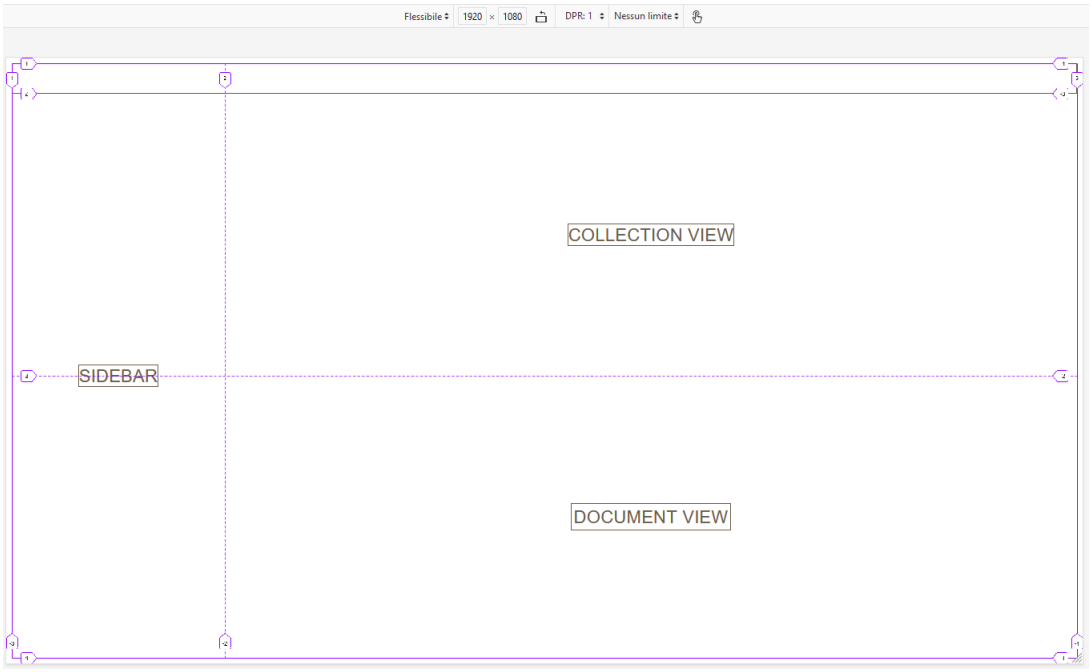


Figura 3.1: Template GUI DocuDipity versione Desktop

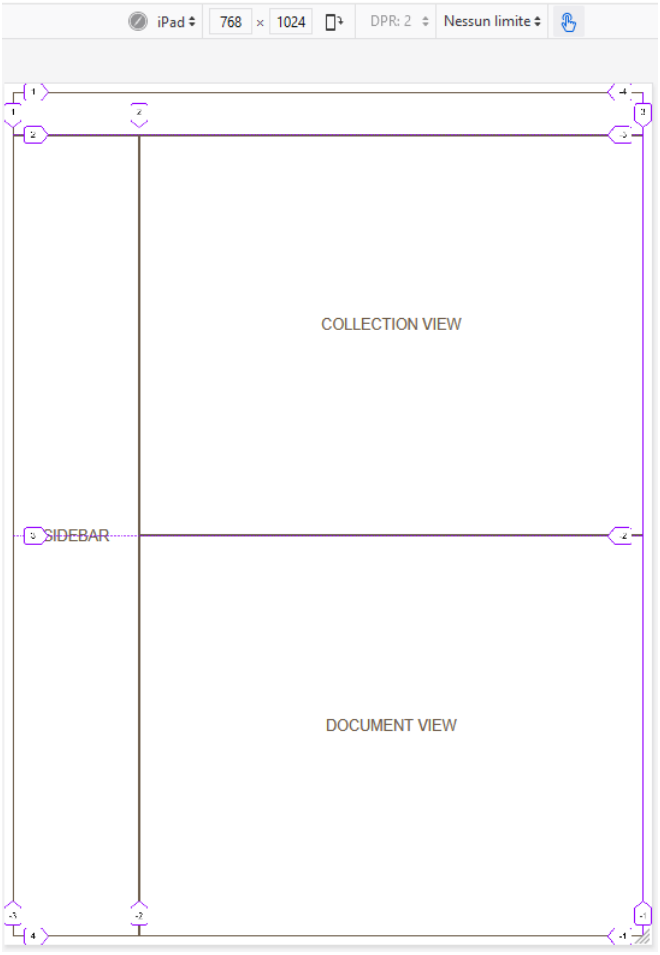


Figura 3.2: Template GUI DocuDipity versione Tablet

Capitolo 4

Progettazione e sviluppo

Al fine di favorire lo sviluppo di un software che fosse leggibile, estendibile e manutenibile è stato utilizzato il linguaggio di programmazione TypeScript, sfruttando i vantaggi derivanti dalla programmazione ad oggetti per la realizzazione del software.

Ogni componente è stato implementato in classi separate, ciascuna delle quali si occupa di gestire la logica di creazione dei singoli componenti e degli eventi ad essi connessi. Lo sviluppatore, attraverso una classe di Factory, potrà chiamare le singole funzioni responsabili della creazione dei componenti senza preoccuparsi della logica utilizza per l'implementazione o dell'architettura con cui sono stati costruiti.

L'utilizzo di una architettura basata su widgets, permette di lavorare su un singolo componente alla volta, facilitando lo sviluppo e garantendo una base solida e flessibile per la futura integrazione di nuovi componenti dell'interfaccia, senza dover andare a compromettere il sistema delle classi creato, introducendo possibili fragilità.

4.1 I widget jQWidgets

Per la creazione dell'interfaccia sono stati utilizzati numerosi widgets appartenenti al framework jQWidgets.

Tra i componenti utilizzati troviamo: jqxTree, jqxGrid, jqxTabs, jqxComboBox, jqxMenu, jqxButton.

Nei capitoli 4.1.1 e 4.1.2 verranno analizzati e spiegati con maggiore attenzione i due widget principali jQWidgets utilizzati in DocuDipity.

4.1.1 Tree

Il Tree è uno dei componenti principali utilizzati in DocuDipity, che ha richiesto insieme alla griglia il maggior sforzo implementativo.

Questo componente funziona come un explorer, ha come obiettivo quello di visualizzare in maniera gerarchica gli articoli del progetto che vengono caricati dal server. L'utente può espandere i vari livelli dell'albero e attraverso un click aprire il menu contestuale (realizzato attraverso l'integrazione con il widget jqxMenu), che offre l'opzione di aggiungere la selezione di articoli alla view corrente o ad una nuova Collection View che verrà aperta in nuovo tab, con una view tabellare in cui saranno presenti gli elementi selezionati.

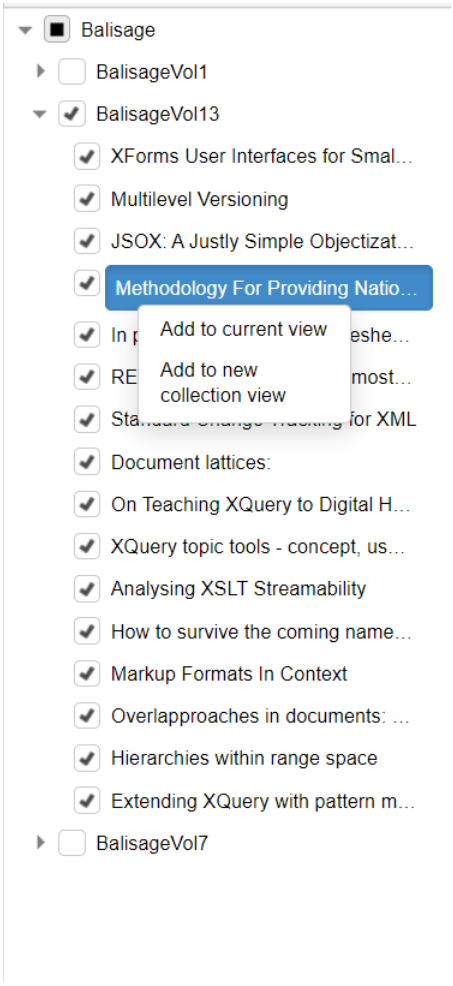


Figura 4.1: Visualizzazione del Tree all'interno dell'interfaccia DocuDipity

Nel Listing 4.1 è descritta una parte dell'implementazione del Tree, in particolare viene mostrato il codice del costruttore della suddetta classe. La classe Tree ha un costruttore che prende come parametro un oggetto complesso formato dall'id dell'elemento HTML in cui verrà disegnato il componente e da un oggetto di configurazione della classe TreeOptions, in cui sono definite tutte le proprietà necessarie alla costruzione del Tree. La classe TreeOptions permette al client di costruirsi un componente su misura senza essere minimamente esposto ai dettagli implementativi.

Nel Listing 4.2 è invece illustrato un esempio di chiamata alla funzione di factory che si occupa di creare un'istanza del Tree. Possiamo notare che come primo parametro della funzione viene passato l'id che si vuole assegnare all'elemento HTML, come secondo l'oggetto TreeOptions, sulla base del quale viene creata l'istanza del Tree, ed infine il container a cui verrà appeso l'elemento HTML creato.

Tra le varie proprietà che vengono passate all'interno della funzione di factory di questo esempio di chiamata troviamo:

- *height*: l'altezza desiderata per il componente
- *fields*: array di proprietà utilizzate dalla classe per creare il datasource del Tree
- *dataSourceFieldId*: l'id che verrà assegnato ad ogni singolo item
- *records*: un oggetto con una serie di proprietà di cui il widget ha bisogno per costruire la struttura gerarchica (proprietà id degli item, proprietà father per identificare gli elementi padre e proprietà text per il testo da inserire all'interno dei singoli item)
- *contextMenu*: un booleano che determina la presenza del menu contestuale sul Tree
- *onClick*: una funzione di callback dell'evento onClick() relativo al menu contestuale. È stata utilizzata per specificare le azioni da eseguire nel caso di click su "Add to current view" oppure "Add to new collection view", per aggiungere rispettivamente gli articoli selezionati alla Collection View corrente (tab selezionato), o ad un nuova Collection View (nuovo tab)

Le funzionalità del jqxTree messo a disposizione dalla libreria jQWidgets, sono state estese attraverso l'integrazione con il menù contestuale ed un tooltip visibile su ogni item del Tree quando il titolo dell'articolo risulta essere eccessivamente lungo.

```
1 export class Tree {
2       public _options: TreeOptions;
3       private _jqxTree: jqwidgets.jqxTree;
4       private _dataSourceAdapter: any;
5       private _contextMenu: jqwidgets.jqxMenu;
```

```
6       public element: HTMLElement;
7       public elementId: string;
8
9       constructor(structure: any) {
10        this._options = new TreeOptions(structure.settings);
11        this.element = document.getElementById(structure.elementId)!;
12        this.elementId = structure.elementId;
13
14        if (this._options.fields == null) this._options.fields = [];
15
16        if (this._options.dataSourceFieldId == null)
17          this._options.dataSourceFieldId = "id";
18
19        if (this._options.contextMenu == null) this._options.
              contextMenu = false;
20
21        let dataSource: any = {};
22        dataSource.id = this._options.dataSourceFieldId;
23        dataSource.datatype = "local";
24        dataSource.datafields = this._options.fields;
25        this._dataSourceAdapter = new $.jqx.dataAdapter(dataSource);
26
27        if (this._options.records == null)
28          this._options.records = {} as TreeRecordItem;
29
30        let treeOptions: jqwidgets.TreeOptions =
31        {
32          checkboxes: true,
33          height: this._options.height,
34          hasThreeStates: true,
35          theme: "bootstrap",
36          animationShowDuration: 0,
37          toggleMode: 'click'
38        };
39
40        this._jqxTree = jqwidgets.createInstance(
41        "#" + this.elementId,
42          "jqxTree",
```

```
43          treeOptions
44        ) as jqwidgets.jqxTree;
```

Listing 4.1: Costruttore della classe Tree

```
1     this._explorerDocuments = createTree("
          projectsView_explorerDocuments",
2       {
3         contextMenu: true,
4         height: "calc(100% - 36px)",
5         fields:
6         [
7           { name: "id" },
8           { name: "authors" },
9           { name: "year" },
10          { name: "father" },
11          { name: "name" },
12        ],
13        dataSourceFieldId: "id",
14        records:
15        {
16          id: "id",
17          father: "father",
18          text: "name",
19        },
20        onContextMenuClick: (e) => { ... }
21      },$("#sidebar")[0]);
```

Listing 4.2: Chiamata alla funzione di factory per la creazione di un'istanza del Tree

Nei Listing 4.3 e 4.4 vengono illustrati rispettivamente il metodo dataSource della classe Tree che si occupa di costruire il datasource in base ai dati ricevuti dall'API ed un esempio di chiamata a questa funzione.

Quando è stata fatta la chiamata all'API per ottenere tutti gli articoli scientifici da utilizzare nel componente, non è stato necessario scrivere alcuna riga di codice nel client, se non la chiamata asincrona per ottenere i dati richiesti, e la chiamata al metodo datasource del Tree.

```
1         dataSource(items?: any[]) {
2         if (items != null) {
3           this._dataSourceAdapter._source.localdata = items;
4           this._dataSourceAdapter.dataBind();
5
6           let records = this._dataSourceAdapter.getRecordsHierarchy(
7             this._options.records!.id,
8             this._options.records!.father,
9             "items",
10            [{ name: this._options.records!.text, map: "label" }]
11          );
12
13          $("#" + this.elementId).jqxTree({ source: records });
14
15          this._jqxTree.getItems().filter(k => k.element as HTMLElement
                ).forEach(k => {
16            k.element.setAttribute("title", k.label);
17          });
18          //#region Create original Datasource
19          this._currentUnchangedItems = [];
20          for (let item of items)
21          {
22            let currentItem = jQuery.extend(true, {}, item);
23            this._currentUnchangedItems.push(currentItem);
24          }
25          //#endregion
26        }
27        return this._currentUnchangedItems;
28      }
```

Listing 4.3: Metodo dataSource della classe Tree per la costruzione del datasource

```
1         this.getExplorerSource<TreeItem[]>().then((res) => {this.
              _explorerDocuments.dataSource(res)});
```

Listing 4.4: Esempio di chiamata alla funzione datasource per popolare il Tree

4.1.2 Grid

Questo widget è stato utilizzato per la creazione di un'area di lavoro che permette all'utente di lavorare sugli articoli che gli suscitano maggiore interesse tra quelli presenti all'interno del progetto, organizzarli attraverso l'applicazione di filtri, aprirli con una visualizzazione ipertestuale, ed eventualmente rimuoverli dall'area di lavoro una volta terminata l'analisi.

La griglia è formata da una toolbar in cui sono stati implementati due bottoni, il primo permette di aprire gli articoli selezionati nella Document View, il secondo di rimuoverli dalla griglia.

Le funzioni di ricerca, ordinamento e applicazione di filtri sugli item della griglia sono implementate direttamente all'interno delle singole colonne, con il vantaggio di non dover sprecare ulteriore spazio sopra la griglia per l'implementazione di filtri attraverso TextBox, ComboBox, ecc.

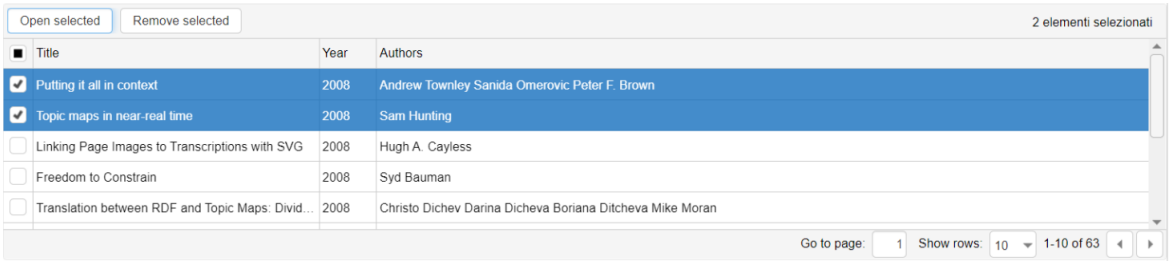


Figura 4.2: Visualizzazione della Grid all'interno dell'interfaccia DocuDipity

Nel Listing 4.5 vediamo un esempio di chiamata alla funzione di factory per la creazione di una griglia. Anche in questo caso, come accadeva per il Tree, viene passato alla funzione un oggetto di configurazione per creare la griglia in base alle esigenze del client. Tra i parametri degni di nota mostrati in questo esempio, troviamo due array, toolbar e columns, entrambi creati per facilitare lato client la creazione della griglia, ed esporre il meno possibile lo sviluppatore ai dettagli implementativi.

Vediamo nel dettaglio la toolbar, creata a partire da un array di oggetti ToolbarItem (Listing 4.6), formati dalle seguenti proprietà:

- *type*: enumerativo in base al quale la griglia renderizza un bottone con un comportamente specifico piuttosto che un altro (ad esempio nel Listing 4.5, viene passato

l'enum ToolbarItemType.Delete, necessario alla griglia per creare un bottone che si occupa di eliminare le righe selezionate, evitando di aggiungere righe di codice relative alla logica dell'eliminazione direttamente nel client)

- *value*: id che viene assegnato al bottone
- *text*: testo che viene scritto nel bottone
- *visible*: visibilità del bottone
- *click*: funzione di callback relativa al click del bottone
- *delete*: funzione di callback relativa all'evento di eliminazione di un item della griglia

Analogamente alla toolbar, anche columns si comporta pressoché nello stesso modo, richiedendo un oggetto formato da:

- *title*: titolo della colonna
- *field*: nome della proprietà dell'oggetto che viene utilizzato per gli item di quella colonna
- *type*: tipo della colonna
- *width*: larghezza della colonna

All'interno della funzione factory createGrid, vengono inoltre passate le callback relative agli eventi di riga selezionata (onSelectRow) e riga deselezionata (rowUnselect).

```
1     let grid = createGrid(divGrid.id,
2       {
3         width: "100% - 32px",
4         height: "calc(50% - 64px)",
5         toolbar: [
6           {
7             type: ToolbarItemType.Custom,
8             value: "btnOpenSelected" + divGrid.id,
9             text: "Open selected",
10            visible: false,
11            click: () => {...}
12          },
13          {
14            type: ToolbarItemType.Delete,
```

```
            value: "btnDeleteSelected" + divGrid.id,
            text: "Remove selected",
            visible: false,
            delete: (e) => {...}
          }
        ],
        columns: [
          {
            title: "Title",
            field: "name",
            type: ColumnTypeEnum.String,
            width: "25%",
          },
          {
            title: "Year",
            field: "year",
            type: ColumnTypeEnum.Number,
            // fitSpace: true, //has to be recalculated each time
            width: "5%",
          },
          {
            title: "Authors",
            field: "authors",
            type: ColumnTypeEnum.String,
            width: "67.7%",
          },
        ],
        dataSourceFieldId: "id",
        onSelectRow: () => {...}
        },
        rowUnselect: () => {...},
      }, true);
```

Listing 4.5: Chiamata alla funzione di factory per la creazione di un'istanza della Grid

```
export class ToolbarItem {
  type: ToolbarItemType;
  text?: string;
  click?: (e: ToolbarClickEvent) => void;
```

```
5    delete?: (e: ToolbarClickEvent) => void;
6    visible?: boolean;
7    value?: string;
8  }
```

Listing 4.6: Classe ToolbarItem utilizzata per la creazione della toolbar della griglia

4.2 I componenti custom

Oltre ai widgets della libreria jQWidgets sono stati creati altri componenti sfruttando la tecnologia dei Web Components. Attraverso questa tecnologia è stato possibile creare elementi HTML riutilizzabili in maniera del tutto indipendentemente dal framework.

In particolare sono stati sviluppati tre componenti, i primi due, ossia DocuRender e ButtonBar, sono stati integrati insieme per formare un componente unico in grado di visualizzare gli articoli in base ad una determinata View, mentre il terzo, ossia DocuCardList, che è stato sviluppato ma non adottato in questa soluzione, rappresenta un prototipo alternativo al componente precedente.

4.2.1 DocuRender

DocuRender è un Web Component che si occupa di fornire una visualizzazione concreta di un articolo specifico secondo una determinata vista. Come è possibile notare dalla Figura 4.3, il componente è formato dai seguenti elementi:

- Una toolbar formata da un altro Web Component (trattato in maniera più approfondita nel capitolo successivo), che oltre a mostrare il titolo dell'articolo aperto dalla Collection View, permette all'utente di selezionare l'articolo all'interno della griglia e scorrerlo verso il basso o verso l'alto attraverso le frecce posizionate ai lati del bottone, con conseguente cambiamento dell'ipertesto mostrato dentro il componente ogni volta che viene premuta una freccia.

  Procedendo verso destra, all'interno della toolbar è presente una ComboBox utilizzata per mostrare all'utente le viste disponibili per quell'articolo (in questa versione di Docudipity è stata implementata la vista ipertestuale), ed infine un icona per rimuovere dalla DocumentView il componente.

- Un body che si si occupa di mostrare il contenuto ipertestuale dell'articolo selezionato, effettuando una chiamata all'API GET che restituisce l'HTML relativo al documento aperto e di applicare un CSS per stilizzarlo come mostrato in Figura 4.3

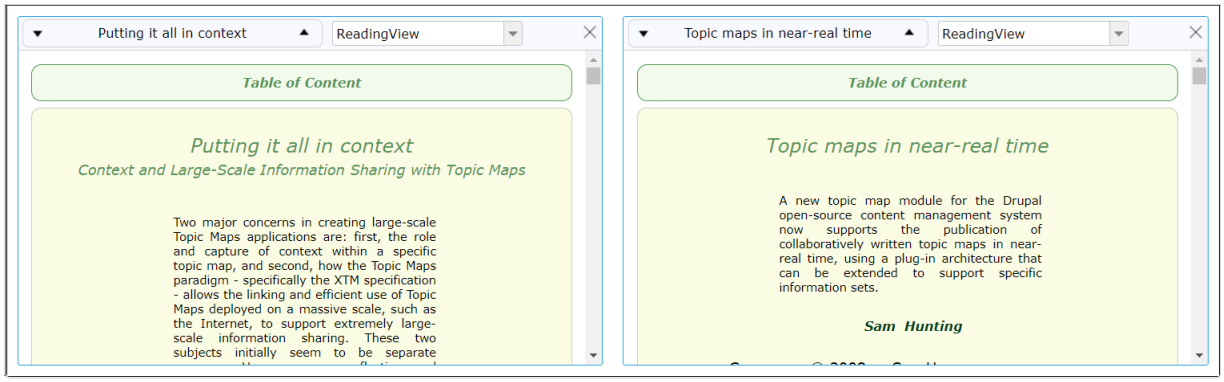


Figura 4.3: Visualizzazione ipertestuale di due articoli scientifici attraverso DocuRender

Vediamo anche in questo caso una parte dell'implementazione ed un esempio di chiamata alla funzione di factory che si occupa di renderizzare il componente.

Come viene mostrato in Figura 4.7, la classe DocuRender del nostro componente estende una classe DocuControl, che si occupa di applicare all'elemento HTML uno stile CSS generico (lo stile da applicare è determinato in base ad un enumerativo che identifica il tipo di componente, in questo caso ControlTypeEnum.DocuRender), e di appendere l'HTML definito nella proprietà template della classe all'elemento HTML del componente. La classe implementa inoltre un'interfaccia DocumentObserver, utilizzata per comunicare con un'altra classe che si occupa di gestire il meccanismo di registrazione di questi componenti all'evento di rimozione di un articolo dalla griglia (maggiore approfondimento nel capitolo 4.4).

Anche in questo caso, come nei precedenti widgets del framework jQWidgets, viene passato al costruttore un oggetto di configurazione, per lasciare all'utente la massima libertà di costruzione del componente senza dover essere esposto ad alcun dettaglio implementativo.

```
1  export class DocuRender extends DocuControl implements
       DocumentObserver {
2  private _options: DocuRenderOptions;
3  private _cmbViews: ComboBox;
4
5  constructor(element: HTMLElement, options?: DocuRenderOptions | null)
6  {
7    super(element, ControlTypeEnum.DocuRender);
8
9    if (options == null) options = new DocuRenderOptions();
10
```

```
11      this._options = options;
12
13      if (options.buttonText != null)
14        this._options.buttonText = options.buttonText;
15      if (options.value != null) this._options.value = options.value;
16
17      if (options.onButtonDownClick != null)
18        this._options.onButtonDownClick = options.onButtonDownClick;
19      if (options.onButtonUpClick != null)
20        this._options.onButtonUpClick = options.onButtonUpClick;
21
22      super.render(this.template);
23
24      this.createControls();
25
26      if(options.rebind != null)
27        this.doWebApiCall(options.rebind.parameters);
28    }
29
30    get template() {
31      return `
32              <div class='docu-render_toolbar' id='docu-render_toolbar${
                    this._element.id}'>
33                  <div class='docu-render_cmbViews' id='
                        docuRender_cmbViews'></div>
34              </div>
35              <div class='docu-render_view'>
36                  <div id='document'></div>`;
37    }
```

Listing 4.7: Costruttore della classe DocuRender

Nella Figura 4.8 vediamo un esempio di chiamata alla funzione di factory che crea il componente. Come primo parametro della funzione viene passato l'oggetto di configurazione con le seguenti proprietà:

- *buttonText*: stringa utilizzata come testo del bottone
- *value*: id del bottone
- *rebind*: utilizzata per passare i parametri con cui viene fatta la chiamata asincrona per ottenere la visualizzazione ipertestuale degli articoli
- *onButtonDownClick*: callback relativa all'evento di click sulla freccia in basso del bottone
- *onButtonUpClick*: callback relativa all'evento di click sulla freccia in alto del bottone
- *onClick*: callback relativa all'evento di click sul bottone stesso

Come secondo parametro della funzione viene passato il container a cui appendere il componente, ed infine l'id del componente stesso. Questa funzione viene richiamata per ogni articolo che l'utente decide di aprire dalla Collection View.

```
1        let docuRender = createDocuRender(
2        {
3          buttonText: item.name,
4          value: item.id,
5          rebind:
6          {
7            parameters:
8            {
9              expressionId: item.id,
10             schemaId: "60228f84483b5f04449c4703",
11           }
12         },
13         onButtonDownClick: (e) => {...},
14         onButtonUpClick: (e) => {...},
15         onClick: () => {...},
16       }, $("#documentviewer" + this._tab._jqxTabs.selectedItem)[0], "
           docudipity_docuRender" + item.id + this._docuRenderCounter
           ++);
```

Listing 4.8: Chiamata alla funzione di factory per la creazione di un'istanza di DocuRender

4.2.2 ButtonBar

ButtonBar è il componente che è stato realizzato per essere integrato all'interno del DocuRender.

Si tratta di un bottone che permette di scorrere gli articoli scientifici aperti nella Document View, all'interno della griglia.

Figura 4.4: Visualizzazione della ButtonBar all'interno dell'interfaccia DocuDipity

Nella Listato 4.9 è rappresentata la funzione createControls della classe DocuRender, in particolare viene mostrata la parte relativa alla funzione di factory che si occupa di creare il componente. Come primo parametro della funzione viene passato l'id da assegnare al componente, come secondo parametro l'oggetto di configurazione per creare il ButtonBar, ed infine il container a cui appendere l'elemento.

Tra i parametri passati all'interno dell'oggetto di configurazione troviamo:

- *onArrowDownClick*: callback relativa all'evento di click sulla freccia in basso (passata in fase di creazione del DocuRender)
- *onArrowUpClick*: callback relativa all'evento di click sulla freccia in alto (passata in fase di creazione del DocuRender)
- *onClick*: callback relativa all'evento di click sul bottone (passata in fase di creazione del DocuRender)
- *buttonValue*: testo del bottone (passato in fase di creazione del DocuRender)

```
1    let buttonBar = createButtonBar("btnBar_" + this._element.id,
2      {
3        onArrowDownClick: (e) => {
4          if (this._options.onButtonDownClick != null) {
5            let arrowDownClickEvent = new ArrowDownClickEvent();
6            let value = $(this._element).find( "#
                 docudipity_lblDocSelectionbtnBar_" + this._element.id)
                 [0];
7            arrowDownClickEvent.value = value;
8            this._options.onButtonDownClick(arrowDownClickEvent);
9          }
10       },
11       onArrowUpClick: (e) => {
12         if (this._options.onButtonUpClick != null) {
13           let arrowUpClickEvent = new ArrowDownClickEvent();
14           let value = $(this._element).find("#
                 docudipity_lblDocSelectionbtnBar_" + this._element.id)
                 [0];
15
16           arrowUpClickEvent.value = value;
17           this._options.onButtonUpClick(arrowUpClickEvent);
18         }
19       },
20       onClick: (e) => {
21         if (this._options.onClick != null)
22           this._options.onClick(e);
23       },
24       buttonValue: this._options.buttonText,
25     },
26     $("#docu-render_toolbar" + this._element.id)[0]);
```

Listing 4.9: Chiamata alla funzione di factory per la creazione di un'istanza del ButtonBar

4.2.3 Un prototipo alternativo: DocuCardList

Sono stati eseguiti diversi test prima di procedere alla creazione definitiva del DocuRender, uno tra questi vede la creazione di DocuCardList, un Web Component alternativo per visualizzare i documenti con alcuni metadati, ed un'icona corrispondente al formato del documento.

Si tratta di un prototipo di un componente non ancora ultimato e che pertanto non è stato utilizzato in questa versione di Docudipity, ma è stato proposto come possibile alternativa al componente DocuRender per la visualizzazione a mosaico dei documenti all'interno della Collection View.

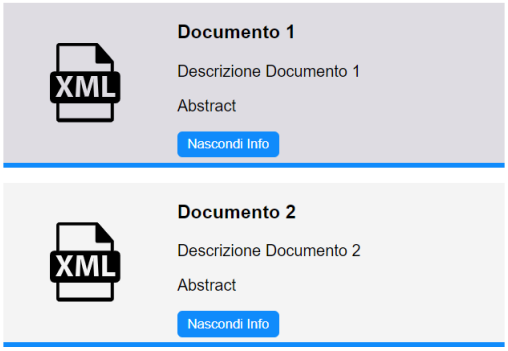


Figura 4.5: Prototipo Web Component DocuCardList

4.3 Factory

Il Factory method è un design pattern di tipo creazionale che fornisce una soluzione alla creazione di oggetti senza dover specificare l'esatta classe dell'oggetto che viene creato. Questo pattern è stato utilizzato per separare la logica di creazione dei vari componenti dell'interfaccia DocuDipity (widgets appartenenti al framework jQWidgets e Web Components) dal codice client che utilizza effettivamente questi componenti, in modo da favorire l'estensione futura dei widgets in maniera indipendente dal resto del codice. L'implementazione di ogni componente è stato incapsulata all'interno di classi concrete separate e la logica di creazione di tali oggetti è stata delegata ad una classe di factory (utilityManager.ts), che si occupa di creare le istanze degli oggetti, esponendo delle funzioni per creare ciascun componente. Il client attraverso la classe utilityManager può chiamare la funzione necessaria alla creazione del componente desiderato, passando come parametro un oggetto in cui specificherà tutte le opzioni di creazione dell'oggetto, l'id che vuole assegnare al componente, e il container a cui vuole appenderlo.

Vediamo un esempio di una funzione di Factory per la creazione di una ComboBox (Listing 4.10): la funzione createControls si occupa di costruire l'elemento HTML inse-

rendo il tag corretto in base ad un enumerativo, impostare all'elemento l'id che viene passato come primo parametro della funzione ed appendere l'elemento HTML al container passato come terzo parametro della funzione. Nella seconda parte della funzione viene invece creato il componente attraverso un plugin jQuery (Listing 4.11). I plugin realizzati rendono disponibili nuovi metodi sull'oggetto prototype di jQuery, in questo caso sono state create tante funzioni quanti sono i componenti che vengono utilizzati in Docudipity. Il metodo docuComboBox(options), ritorna un'istanza della classe ComboBox, in base all'oggetto 'options" che viene passato come parametro, e viene chiamato su un selettore jQuery.

Se un futuro programmatore avrà la necessità di estendere la struttura esistente, integrando un nuovo componente, sarà sufficiente aggiungere una nuova funzione nella classe utilityManager, ed una classe concreta all'interno della directory 'components' in cui dovrà essere incapsulata la logica di creazione dell'oggetto.

```
1 export function createComboBox(id: string, options?: ComboBoxOptions |
     null, container?: HTMLElement) : ComboBox
2 {
3     let element = createControls(ControlTypeEnum.ComboBox, container,
         id);
4     let control = $(element).docuComboBox(options);
5
6     return control;
7 }
```

Listing 4.10: Chiamata alla funzione di factory per la creazione di un'istanza di ComboBox

```
1  jQuery.fn.extend({
2      docuComboBox: function (options?: ComboBoxOptions | null)
3      {
4          let element = $(this)[0] as HTMLElement;
5          let structure : any = {
6              settings: options,
7              element: element,
8              elementId: element.id
9          };
10         return new ComboBox(structure);
11     }}
```

Listing 4.11: Plugin jQuery per la creazione di una ComboBox

4.4 Observer

L'observer è un design pattern di tipo comportamentale che attraverso un meccanismo di registrazione ad eventi permette di notificare oggetti multipli (Observers) quando uno o più eventi si verificano nell'oggetto che stanno osservando (Subject). Questo pattern è stato utilizzato nel momento in cui si è concretizzata la necessità di dover gestire l'evento per la rimozione di articoli dalla griglia della Collection View. Immaginiamo lo scenario in cui l'utente seleziona uno o più articoli dalla griglia e decide di aprirli per visualizzare la loro rappresentazione ipertestuale nella Document View per compiere delle analisi. Una volta terminata la fase di analisi sugli articoli, l'utente decide di rimuoverli dalla griglia, che ricordiamo essere l'area di lavoro, a questo punto, essendo l'articolo non più di interesse, andrà rimossa anche la sua visualizzazione all'interno della Document View. Questo risultato è stato ottenuto grazie all'applicazione del design pattern Observer. Ogni volta che l'utente decide di aprire e quindi visualizzare un nuovo articolo, l'oggetto creato viene registrato al verificarsi di un "cambiamento", in questo caso rappresentato dalla rimozione di un articolo. Per questa ragione è stata creata una classe per gestire i cambiamenti di stato degli articoli (Listing 4.12), che mantiene all'interno di un array tutte le referenze agli oggetti che si sono registrati, in modo che una volta che l'evento osservato si verifica, venga chiamato su tutti gli observer registrati un metodo deleteElement, che si occupa di rimuovere dal DOM l'elemento HTML relativo al visualizzatore ipertestuale. Ogni Observer implementa l'interfaccia DocumentObserver che espone il metodo deleteElement, implementato all'interno di ciascun componente.

```
1  export class DocumentState {
2    private observers: any[] = [];
3
4    constructor() {}
5
6    attach(observer: DocumentObserver)
7    {
8      this.observers.push(observer);
9    }
10
11   detach(observer: DocumentObserver)
12   {
13     const index = this.observers.indexOf(observer);
14     if (index > -1) {
15       this.observers.splice(index, 1);
16     }
17   }
```

```
18    notifyDelete(itemsId: any) {
19      for(let i = this.observers.length -1;  i >= 0; i--)
20      {
21        if(itemsId.indexOf(this.observers[i].value()) > -1)
22        {
23          this.observers[i].deleteElement();
24          this.detach(this.observers[i]);
25        }
26      }
27    }
28  }
```

Listing 4.12: Classe per la gestione del cambiamento di stato degli articoli scientifici

Capitolo 5

Conclusioni

L'obiettivo di questa tesi era quello di sviluppare un'interfaccia grafica del web tool preesistente DocuDipity. In primo luogo, per comprendere il dominio del problema, sono state analizzate le tecniche di visualizzazione di documenti, con particolare attenzione a quelle utilizzate all'interno di DocuDipity per la gestione documentale di articoli scientifici.

Lo sviluppo del progetto ha seguito un iter che non si ferma alla semplice implementazione, affiancando a questa fase, un percorso completo che inizia dal design dell'interfaccia realizzando prototipi e mockup, alla valutazione delle tecnologie e dei framework più adatti nel rispetto dei requisiti identificati per Docudipity.

Per la realizzazione dell'interfaccia è stato usato il framework jQWidgets, che mette a disposizione un'ampia libreria di widgets, in parte integrati con nuove funzionalità per far fronte alle esigenze del progetto.

Oltre all'utilizzo di jQWidgets, sono stati creati componenti HTML riutilizzabili e facilmente estendibili in modo che i prossimi programmatori che si occuperanno di sviluppi futuri del progetto troveranno una serie di standard utili per poter lavorare in maniera più agevole e ordinata. La parte di sviluppo dell'interfaccia di cui mi sono occupato è quella relativa all'area di lavoro del progetto, per la quale sono stati soddisfatti buona parte dei requisiti definiti in fase di analisi.

Uno dei requisiti requisiti relativi all'area di lavoro che non si è potuto soddisfare è la creazione di un modulo per l'import degli articoli scientifici all'interno del progetto; per ora tutti gli articoli che l'utente visualizza all'interno del progetto e su cui può eseguire operazioni sono caricati attraverso la chiamata ad un'API che restituisce tutti i documenti presenti sul server.

In successive versioni di Docudipity, sarà quindi necessario integrare all'interno del sistema delle classi esistente un nuovo componente, che dovrà essere utilizzato in più punti dell'interfaccia, sfruttando gli standard che sono stati utilizzati finora. Per creare il componente e mantenere intatta la struttura, evitando di introdurre fragilità all'interno

delle classi si consiglia di realizzarlo in una classe separata, sfruttando il widget jqxWIndow messo a disposizione da jQWidgets o sviluppando un componente custom come è stato già fatto per altri componenti utilizzati in DocuDipity. Sono destinate a future implementazioni la parte di gestione e creazione dei progetti e l'integrazione di nuove viste, come quella SunBurst, in aggiunta a quella ipertestuale. L'architettura utilizzata per la realizzazione della nuova interfaccia è completamente indipendente dalle viste, a differenza della precedente, che era invece stata pensata e progettata per funzionare esclusivamente con due sole viste.

Oltre alla visualizzazione attraverso SunBurst, già presente nella precedente versione, sono stati pensati anche altri tipi di visualizzatori di articoli scientifici da implementare successivamente nell'interfaccia, uno di questi è un visualizzatore in grado di mostrare alcuni dati statistici relativi agli articoli, come il numero di volte in cui è stato citato, l'anno di pubblicazione, ecc.

In prossime versioni di DocuDipity è prevista anche l'implementazione del meccanismo di coordinamento delle viste, che consentirà di sincronizzare i visualizzatori tra loro, combinando in questo modo i vantaggi di questi ultimi, ossia quello derivante dal SunBurst che consente di rappresentare in uno spazio ristretto l'organizzazione gerarchica di un documento e quello della visualizzazione ipertestuale che ne fornisce una visualizzazione in dettaglio.

Non essendo ancora stata implementata la parte relativa alla gestione dei progetti, l'API per il salvataggio è stata testata creando sul server un progetto di prova e salvando gli articoli scientifici aperti nell'area di lavoro all'interno di questo progetto.

La nuova versione realizzata porta notevoli miglioramenti rispetto alla precedente, risulta essere molto più flessibile ed intuitiva, meglio ingegnerizzata e con una parte client ben separata da quella server.

Bibliografia

[1] Ag-grid. `https://www.ag-grid.com/`.

[2] Balsamiq. `https://balsamiq.com/`.

[3] jqwidgets. `https://www.jqwidgets.com/`.

[4] Ng-bootstrap. `https://ng-bootstrap.github.io/#/home/`.

[5] W3techs. `https://w3techs.com/technologies/details/js-jquery`.

[6] Benjamin B Bederson. Photomesa: a zoomable image browser using quantum treemaps and bubblemaps. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 71–80, 2001.

[7] Mei C Chuah. Dynamic aggregation with circular visual designs. In *Proceedings IEEE Symposium on Information Visualization (Cat. No. 98TB100258)*, pages 35–43. IEEE, 1998.

[8] 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.

[9] Angelo Di Iorio, Silvio Peroni, Francesco Poggi, Fabio Vitali, and Paolo Ciancarini. Docudipity: disclosing habits in scholarly writing. *DMSVIVA2019*, 2016.

[10] 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.

[11] Stefano Notari (Tesi in corso in Informatica, Università di Bologna). Progettazione API REST per un ambiente di lettura e di confronto per articoli scientifici. 2021.

[12] Brian Scott Johnson. *Treemaps: Visualizing hierarchical and categorical data*. PhD thesis, 1993.

[13] Kelly Patricia Kingrey. Concepts of information seeking and their presence in the practical library literature. *Library philosophy and practice*, 4(2):1–14, 2002.

[14] 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.

[15] Michael J McGuffin and Jean-Marc Robert. Quantifying the space-efficiency of 2d graphical representations of trees. *Information Visualization*, 9(2):115–140, 2010.

[16] Francesco Poggi, Paolo Ciancarini, Angelo Di Iorio, Silvio Peroni, and Fabio Vitali. Exploiting coordinated views for scholarly reading and analysis. In *25th International Distributed Multimedia Systems Conference on Visualization and Visual Languages, DMSVIVA 2019*, volume 2019, pages 113–124. Knowledge Systems Institute Graduate School, KSI Research Inc., 2019.

[17] 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.

[18] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.

[19] 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.

[20] 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.

[21] Carol Tenopir, Donald W King, Sheri Edwards, and Lei Wu. Electronic journals and changes in scholarly article seeking and reading patterns. In *Aslib proceedings*. Emerald Group Publishing Limited, 2009.

[22] Carol Tenopir, Regina Mays, and Lei Wu. Journal article growth and reading patterns. *New Review of Information Networking*, 16(1):4–22, 2011.

[23] Barbara Tillett. What is FRBR? a conceptual model for the bibliographic universe. *The Australian Library Journal*, 54(1):24–30, 2005.