Alma Mater Studiorum - Università di Bologna

Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea in Informatica

UN AMBIENTE WEB PER LA CONVERSIONE DI DOCUMENTI ELETTRONICI: PROTOTIPAZIONE E INTEGRAZIONE CON PANDOC

Tesi di Laurea

Presentata da:
Stefano Liverani

Relatore:
Chiar.mo Prof.
Angelo Di Iorio

Sessione III
Anno Accademico 2020-2021

Ringraziamenti..

Prima di procedere, vorrei dedicare alcune righe a tutti coloro che mi hanno accompagnato in questo percorso.
Un sentito grazie al mio relatore, Angelo Di Iorio, per avermi dato questa possibilità ed avermi sostenuto e aiutato, offrendo la massima disponibilità in questi mesi.
Grazie ai miei genitori e ai miei fratelli per avermi sostenuto e incoraggiato in questo lungo e difficile percorso, per avermi appoggiato nelle mie scelte ed aiutato quando potevano.
Un grandissimo grazie va alla mia ragazza, Laura, che hai sempre creduto in me, anche nei momenti di maggiore sconforto, riuscendo a strapparmi un sorriso e dandomi fiducia.
All'università sapevo che avrei trovato colleghi e compagni con cui condividere le giornate sui libri. Grazie Lorenzo, perché in te ho trovato anche un amico.
Grazie Sabrina per la tua guida, mi hai aiutato in tutti questi anni consigliandomi e aiutandomi nei momenti di difficoltà.
Ringrazio infine tutti i miei amici per i momenti di spensieratezza e per avermi appoggiato e incoraggiato sempre nelle mie scelte.

Indice

Capitolo 1

Introduzione

Alla base di questo progetto vi è la necessità di creare un'applicazione in grado di svolgere molteplici operazioni su gruppi di documenti di testo facenti parte di una collezione, una raccolta di documenti con caratteristiche comuni che quindi necessitano di essere raggruppate ed analizzate insieme.
Tra le necessità principali vi è infatti quella di convertire questi file ed estrarre informazioni utili a creare un documento che riesca a descrivere a pieno la collezione analizzata.
Questa esigenza è nata al fine di arricchire le funzionalità di Docudipity[9], un ambiente finalizzato a supportare l'esplorazione e l'analisi di articoli scientifici.
Docudipity si occupa infatti di analizzare collezioni di articoli ed elaborarne il contenuto, permettendone una visualizzazione alternativa alla classica ipertestuale.
Per l'elaborazione delle collezioni, Docudipity espone una API che richiede un manifest, un documento in formato JSON contenete alcuni dati relativi ai documenti e alcune informazioni aggiuntive atte a caratterizzare la collezione; occorre inoltre che i documenti in input siano in formato RASH [8], un sottoinsieme di HTML, progettato principalmente per articoli scientifici, che riduce il numero di elementi disponibili al fine di semplificarne la struttura. Questa piattaforma ha quindi lo scopo di adeguare i formati in input, in modo da ottenere una visualizzazione omogenea (a prescindere dal formato originale) all'interno di Docudipity.
É stata creata Leshy, un'applicazione in grado di elaborare documenti in formato Docbook e Docx e convertirli prima in formato HTML e poi RASH, estraendo al tempo stesso dati utili all'analisi dei file.
Uno degli obiettivi della tesi è quello di creare un'architettura modulare per la conversione, permettendo con poche modifiche di aggiungere nuovi formati in input o anche testare altri convertitori. Il nome stesso "Leshy" è stato scelto in quanto, nella mitologia slava, Leshy è uno spirito della foresta in grado di mutare forma e prendere sembianze umane e animali.

L'altro aspetto che la tesi si pone come obiettivo è quello di testare il convertitore Pandoc[5]. Poiché esistono moltissimi convertitori, si è resa necessaria l'analisi di uno in particolare per testare se il grado di flessibilità e di personalizzazione fosse tale da poter offrire al progetto un solido strumento di conversione. In questa tesi verrà quindi mostrato tutto il processo di conversione ed estrapolazione dei dati da parte di Leshy e la sua architettura, oltre ad evidenziare le criticità emerse dall'analisi del convertitore scelto.

L'applicazione, per come è stata studiata, permette all'utente di compilare un form contenente informazioni atte a caratterizzare la collezione, le quali verranno salvate in un apposito manifest.
In seguito sarà possibile caricare gruppi di file dai quali Leshy estrarrà alcuni dati (Titolo, Descrizione, Anno, Autore). Sarà poi possibile modificare le informazioni o integrarle a piacimento tramite un form, prima di procedere all'effettiva conversione.
A conversione avvenuta, verrà mostrato un riepilogo degli esiti della conversione e sarà possibile scaricare un file zip contenente tutti i file correttamente convertiti e il manifest.
Nei seguenti capitoli, vengono mostrati nel dettaglio tutti i passaggi precedentemente descritti.
In una prima fase saranno menzionati DocuDipity, RASH e il loro funzionamento, verranno poi spiegate le necessità di un convertitore e di un manifest e infine analizzati alcuni convertitori esistenti, mettendo in luce i vari pregi e difetti riscontrati.
Successivamente verrà mostrato nel dettaglio quello scelto per il progetto, ovvero Pandoc. Verranno illustrate la sua architettura, le personalizzazioni possibili e le criticità emerse dal suo utilizzo.
Nel terzo capitolo sarà possibile comprendere a pieno il funzionamento dell'infrastruttura: saranno mostrati nel dettaglio tutti gli step da seguire per l'utilizzo di Leshy e i passaggi che vengono svolti per la conversione e la creazione del manifest.
Seguirà una parte in cui si illustrerà nel dettaglio come è stata implementata l'applicazione e quali strumenti sono stati utilizzati per la sua realizzazione.
Verranno mostrate nel quinto capitolo una serie di valutazioni effettuate su gruppi di documenti, riportando i risultati dei vari processi ed eventuali problematiche riscontrate.
Infine saranno tratte le conclusioni riportando anche possibili sviluppi futuri.

Capitolo 2

Da DocuDipity a Leshy

2.1 Cos'è DocuDipity

Docudipity è una web app sviluppata nel 2015 dal DASPLab (Digital and Semantic Publishing Lab), un gruppo di ricerca dell'Università di Bologna, e ne è stata recentemente rilasciata una nuova versione.
Docudipity si occupa di analizzare articoli scientifici fornendone rappresentazioni alternative a quella ipertestuale. L'obiettivo di DocuDipity è quello di voler fornire all'utente una visualizzazione personalizzabile, in grado quindi di focalizzare le informazioni a seconda delle esigenze (zoom and filter). L'utente è infatti in grado, a partire dalla collezione di articoli, di poterne selezionare alcuni e spostarli in un'area di lavoro che permette l'applicazione di filtri. Una volta elaborati sarà possibile aprire i singoli file e metterli a confronto secondo una specifica visualizzazione. Questo è stato reso possibile sfruttando varie tecniche tra cui SunBurst [10] che visualizza i documenti (specialmente gli XML) in maniera gerarchica (mediante uno schema a corone circolari) [Fig: 1].

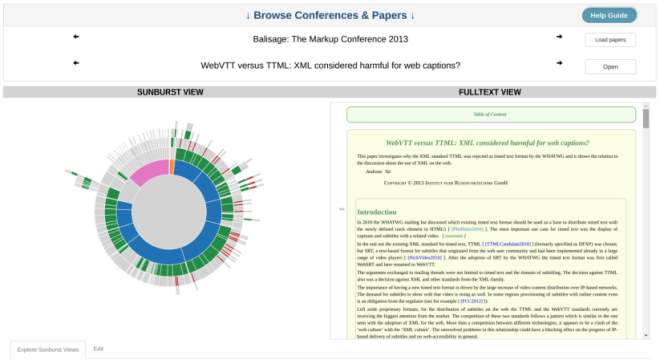

Figura 1: Home page di Docudipity

É in fase di sviluppo una nuova versione di Docudipity che prevede di ricevere in input un gruppo di documenti in formato RASH e un documento in formato JSON contenente alcune informazioni relative ai file, detto manifest. Nella prossima sezione verrà illustrato nel dettaglio il formato RASH e il motivo della sua scelta.

2.2 RASH

RASH[8] (Research Articles in Simplified HTML) è un formato Web-first per la scrittura di articoli accademici in HTML, con lo scopo di semplificarne la struttura, riducendo il numero di elementi a 32. Si pone come obiettivo quello di aiutare gli utenti a focalizzarsi su specifiche caratteristiche dei documenti: per un ricercatore può focalizzare l'attenzione sul contenuto dell'articolo, per un lettore può far sperimentare nuove modalità di navigazione. É inoltre in grado di estrapolare e riutilizzare dati che compaiono sotto forma di annotazioni.
La struttura di RASH si basa quindi su un modello di dati a pattern semplificato: il numero di elementi è stato notevolmente ridotto per permettere all'autore di concentrarsi solo sul contenuto dell'articolo, evitando di focalizzarsi troppo sulla scelta degli elementi da utilizzare. L'utilizzo di questi pattern aiutano alla realizzazione di documenti non ambigui, ben strutturati e li rendono facilmente riutilizzabili in differenti linguaggi.
L'idea alla base di RASH è anche quella di permettere agli autori di continuare a scrivere i loro documenti con il formato a loro più familiare e fornire un framework[2] in grado di convertire questi documenti in RASH. Attualmente il framework è in grado di convertire in tale formato OpenOffice e Microsoft Office.
In questa tesi si cercherà anche di verificare la possibilità di generare un file RASH a partire dai formati scelti in input, ovvero Docbook e Docx, e testare le potenzialità e i limiti del convertitore adottato. Sarà necessario infatti che il convertitore rispetti i vincoli imposti da RASH, ma come vedremo nella sezione 2.5.4 sono stati riscontrati molti limiti nella personalizzazione degli output di Pandoc, il che ha reso necessaria una fase di pre e post-processing.

2.3 Convertitori analizzati

La prima fase è stata l'analisi dei convertitori disponibili e degli strumenti necessari alla loro integrazione all'interno del progetto, ponendo l'attenzione alla capacità di elaborare molteplici documenti e di restituire HTML ben formati.
Di seguito analizziamo alcuni di questi convertitori.

Convertitore	Formati supportati	API	Costo
Zamzar	100+	Si	Pagamento: crediti di conversione
CloudConvert	200+	Si	Pagamento: minuti di conversione
AllDocs	40+	No	Gratuito
Pandoc	40+	Si	Gratuito

Figura 2: Convertitori analizzati

2.3.1 *Zamzar*

Zamzar[14] è un convertitore online che supporta più di 100 formati, in grado di convertire sia file locali che file accessibili via HTTP, FTP, SFTP. Per la personalizzazione delle conversioni, Zamzar fornisce una API.
Per poter usufruire del servizio è necessario ottenere una API key: una volta registrato l'account verrà assegnata una chiave che sarà necessaria per effettuare ogni chiamata al server.
Il processo di conversione si suddivide in tre step. La prima fase prevede la richiesta di conversione a Zamzar: per fare ciò è necessario specificare il formato che si vuole ottenere e inserire il/i file da convertire; mostriamo di seguito un esempio in JS [Fig: 3].

```
var request = require('request'),
    fs = require('fs'),
    apiKey = 'GiVUYsF4A8ssq93FR48H',
    formData = {
        target_format: 'png',
        source_file: fs.createReadStream('/tmp/portrait.gif')
    };

request.post({url:'https://sandbox.zamzar.com/v1/jobs/', formData: formData}, function (err, respon
se, body) {
    if (err) {
        console.error('Unable to start conversion job', err);
    } else {
        console.log('SUCCESS! Conversion job started:', JSON.parse(body));
    }
}).auth(apiKey, '', true);
```

Figura 3: Esempio di conversione

Successivamente sarà necessario effettuare una chiamata GET per verificare l'avvenuta conversione e Zamzar risponderà con un file JSON contenente tutti i dettagli relativi alla conversione, come in questo esempio:


```
{
    "id" : 15,
    "key" : "GiVUYsF4A8ssq93FR48H",
    "status" : "successful",
    "sandbox" : true,
    "created_at" : "2013-10-27T13:41:00Z",
    "finished_at" : "2013-10-27T13:41:13Z",
    "source_file" : {"id":2,"name":"portrait.gif","size":90571},
    "target_files" : [{"id":3,"name":"portrait.png","size":15311}],
    "target_format" : "png",
    "credit_cost" : 1
}
```


Figura 4: Esito della conversione

Infine si potrà procedere ad un'ultima chiamata GET per poter scaricare il/i file mediante l'id assegnatogli precedentemente nel file JSON. Per poter verificare la possibilità di effettuare conversioni e/o ottenere la lista dei formati disponibili, è possibile effettuare questo tipo di chiamata:

```
var request = require('request'),
    apiKey = 'GiVUYsF4A8ssq93FR48H';

request.get('https://api.zamzar.com/v1/formats/', function (err, response, body) {
    if (err) {
        console.error('Unable to get formats', err);
    } else {
        console.log('SUCCESS! Supported Formats:', JSON.parse(body));
    }
}).auth(apiKey, '', true);
```

Figura 5: Esempio di richiesta dei formati disponibili

Per poter usufruire della API di Zamzar è necessario registrarsi sul sito ed aderire ad uno dei piani disponibili.
Il sistema è basato su crediti di conversione, ogni file richiede almeno 1 credito per essere convertito; le conversioni più complesse possono richiedere più crediti. Il servizio è gratuito fino a 100 crediti al mese, dopodiché si passa alla versione a pagamento.

2.3.2 *CloudConvert*

CloudConvert[3] è un convertitore di file online. Supporta non solo formati testo ma anche audio, video e immagini ed è in grado di convertire più di 200 formati. Oltre ad offrire un servizio di conversione online direttamente sul portale, è possibile integrare il convertitore all'interno delle proprie applicazioni mediante una API.
La conversione si basa su "jobs", ovvero l'insieme delle operazioni necessarie a portare a termine un processo di conversione. Ogni conversione può essere costituita da più "tasks". Di seguito viene mostrato un esempio di come è strutturata una "job request":

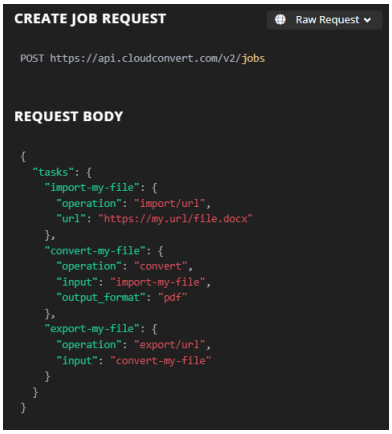

Figura 6: Esempio di Job

Terminata la conversione sarà possibile effettuare una GET e richiedere l'esito della conversione, ottenendo un dettaglio del risultato di ogni task.

Figura 7: Risposta da parte del Server

Nello stesso file sarà presente anche l'URL per il download del file, da richiedere entro 24h dalla conversione mediante una nuova GET.

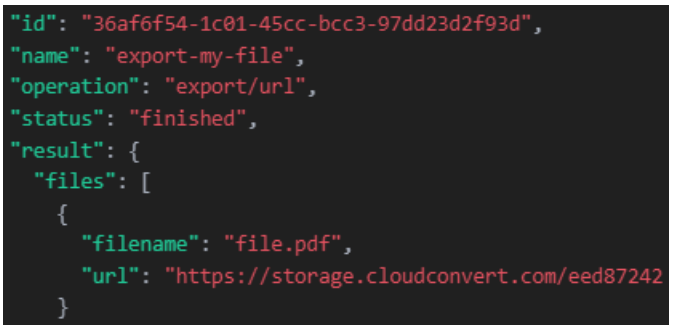

Figura 8: Url per il download del file

Inoltre, è possibile effettuare una GET inserendo dei filtri per verificare che i formati siano compatibili tra loro e supportati da CloudConverter. Di seguito un esempio di questa chiamata:

```
$ curl -g "https://api.cloudconvert.com/v2/convert/formats?
filter[input_format]=pdf&include=options"
```

Figura 9: Comando per la richiesta dei formati disponibili

Anche questo convertitore prevede un servizio gratuito che permette l'utilizzo illimitato della API testabile su una whitelist di file fornita dal produttore.
I pacchetti a pagamento invece, a differenza di Zamzar, utilizzano un sistema a "minuti di conversione". I pacchetti sono personalizzabili: all'aumentare dei minuti acquistati si riduce il costo al minuto.

2.3.3 *Alldocs*

Alldocs[12] è un convertitore online di file di testo completamente gratuito e open-source. Supporta più di 1400 combinazioni di conversione ed è in grado di convertire file fino a 10MB.
Alldocs utilizza Pandoc, un convertitore offline. A differenza dei precedenti convertitori, Alldocs non fornisce una API per integrarsi all'interno di altre applicazioni; risulta pertanto molto limitato per i nostri scopi. É stato comunque utile per verificare in prima fase la compatibilità di formati scelti per il progetto, in quanto ha permesso di testare Pandoc prima di procedere alla sua integrazione nell'applicazione.

2.3.4 *Pandoc*

Pandoc[5] è un convertitore offline disponibile per la maggior parte dei sistemi operativi. Dal sito è possibile scaricare l'installer e in seguito utilizzare il terminale per eseguire la conversione mediante linea di comando.
Questo software è totalmente gratuito e supporta più di 40 formati in input e in output. Nelle varie fasi di conversione è possibile inoltre personalizzare il convertitore stesso, aggiungendo parametri che ne determinano la conversione.
Pandoc fornisce anche una API che sfrutta una libreria Haskell contenente funzioni che simulano i comandi del terminale.
Per poter usufruire invece dello strumento a linea di comando, è necessario un tool che faccia da "ponte" tra Pandoc e la web app, come ad esempio NodeJS.
Nella sezione 2.5 verrà approfondito questo convertitore.

2.4 Scelta del convertitore

Convertitore	Formati supportati	API	Costo
Zamzar	100+	Si	Pagamento: crediti di conversione
CloudConvert	200+	Si	Pagamento: minuti di conversione
AllDocs	40+	No	Gratuito
Pandoc	40+	Si	Gratuito

Figura 10: Convertitori analizzati

Riprendendo la tabella iniziale [Fig: 10], possiamo subito osservare come Zamzar e CloudConvert supportino un numero di file molto maggiore rispetto a Alldocs e Pandoc, a discapito del fatto che entrambi richiedano una sottoscrizione a pagamento per poter usufruire liberamente di tutte le funzionalità e non dispongano di un sistema di personalizzazione della conversione.
Inoltre, essendo le sottoscrizioni a crediti/tempo di conversione non è possibile prevedere un presunto quantitativo di utilizzo, in quanto i file possono variare molto in dimensione e/o complessità di conversione. Stabilire a priori un limite in questo senso avrebbe ridotto fortemente le potenzialità dell'infrastruttura.
Alldocs non fornisce una API in grado di integrarsi con altre applicazioni. D'altro canto, è un sito opensource che permette la conversione di un numero illimitato di documenti sul portale.
Studiando quindi il funzionamento di Alldocs è stato possibile scoprire che la piattaforma sfrutta Pandoc per effettuare le conversioni.
Arriviamo infine a Pandoc che, pur supportando meno formati rispetto a Zamzar e CloudConvert, permette di integrare le proprie funzioni all'interno di web app sfruttando una API in Haskell o alcuni tool.
É stato scelto di testare NodeJS come tool piuttosto che la libreria Haskell in quanto risultava più semplice l'implementazione e l'integrazione del convertitore nell'ambiente. Inoltre, la familiarità con NodeJS ha portato a preferire questo approccio e al contempo a ridurre il numero di linguaggi utilizzati.
Nella prossima sezione verrà mostrato più nel dettaglio il funzionamento di Pandoc.

2.5 Approfondimento su Pandoc

Di seguito analizzeremo nello specifico tutti gli aspetti che caratterizzano questo convertitore.

2.5.1 Configurazione

Come precedentemente citato, è stato scelto di utilizzare Pandoc sfruttando NodeJS. Per fare ciò è necessario scaricare l'installer dal sito di Pandoc dove sono disponibili vari pacchetti per i principali sistemi operativi.
Dopo aver installato il pacchetto, sarà possibile fin da subito effettuare conversioni utilizzando il terminale; l'elenco delle combinazioni di formati input-output è disponibile sul sito di Pandoc.
Per integrarlo all'interno della web app è necessario installare un tool NodeJS: nello specifico è stato utilizzato il pacchetto *node-pandoc*.

2.5.2 Modalità di conversione

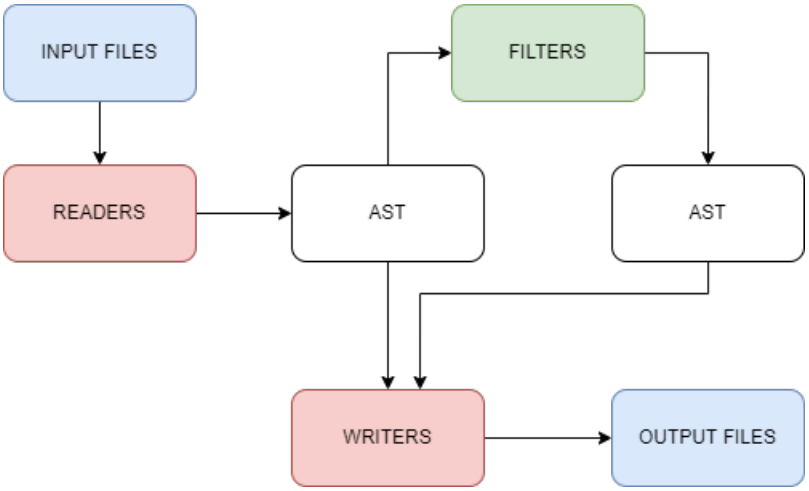

Figura 11: Flusso di conversione di Pandoc

Il sistema di conversione che Pandoc utilizza è basato su un design modulare [Fig:11]: consiste in un insieme di lettori (readers), che si occupano di analizzare il testo in input e di costruire un albero di sintassi astratta detto AST (abstract syntax tree), e di un insieme di scrittori (writers), che si occupano di analizzare l'AST e convertirne il contenuto nel formato richiesto. Nel comando infatti sarà

necessario specificare quale reader e quale writer utilizzare per la conversione.
Di seguito un esempio di visualizzazione dell'AST:

```
[OrderedList (1,Decimal,Period)
 [[Plain [Emph [Str "foo"]]]
 ,[Plain [Str "bar"]]]]
```

Figura 12: Esempio di AST

Oltre a queste due entità fondamentali, Pandoc prevede anche una terza entità che è possibile inserire tra il reader e il writer nel flusso di conversione per aggiungere particolari elaborazioni. Questa entità prende il nome di filter.
Il filter si inserisce nel processo di conversione subito dopo la generazione dell'AST da parte del reader e si occupa di modificare parte di esso a seconda dei filtri inseriti. Una volta conclusa la modifica viene generato un nuovo AST che viene passato al writer, il quale termina il processo di conversione restituendo in output il file convertito.
Pandoc supporta due tipi di filtri: Lua filters e JSON filters.
I primi sono scritti in linguaggio Lua e consistono in tabelle con i nomi di elementi come chiavi e funzioni che agiscono su questi elementi come valori.
I filtri JSON invece creano una visualizzazione JSON dell'AST, lo modificano e poi lo riconvertono in AST prima di passarlo al writer.
I filtri Lua permettono una maggiore personalizzazione in quanto l'interprete Lua è integrato in Pandoc; non necessitano quindi di un software esterno per l'interpretazione del linguaggio e generalmente sono più veloci dei filtri JSON in fase di elaborazione. D'altro canto i filtri JSON possono essere scritti quasi in qualsiasi linguaggio pertanto non occorre imparare una sintassi specifica.

2.5.3 Comandi aggiuntivi

Oltre al filter, Pandoc prevede anche l'utilizzo di comandi aggiuntivi in grado di modificare alcuni tipi di output. Di seguito alcuni esempi:

- **auto_identifiers**: ad un header senza un identificatore specificato in modo esplicito verrà automaticamente assegnato un identificatore univoco in base al testo dell'header;
- **empty_paragraphs**: preserva eventuali paragrafi vuoti (di default vengono infatti omessi);

- **latex_macros**: quando questa estensione è abilitata, Pandoc analizzerà le definizioni delle macro LaTeX e applicherà le macro risultanti a tutta la matematica LaTeX e LaTeX grezza;
- **implicit_figures**: un'immagine con testo alternativo non vuoto, presente da sola in un paragrafo, verrà visualizzata come una figura con una didascalia. Il testo alternativo dell'immagine verrà utilizzato come didascalia;
- **hard_line_breaks**: tutte le nuove righe all'interno di un paragrafo vengono interpretate come interruzioni di riga anziché spazi;
- **extract-media**: estrae dal documento eventuali immagini e le salva in una cartella specifica.

2.5.4 Criticità riscontrate

In questa sottosezione analizzeremo a fondo le criticità riscontrate nell'utilizzo del convertitore Pandoc.
In fase di test infatti sono stati analizzati vari file, contenenti tipi di elementi differenti per verificare il comportamento del convertitore. Prendiamo in esempio un file Docbook e analizziamo i risultati dei primi test di conversione:

```
<info>
  <confgroup>
    <conftitle>Balisage: The Markup Conference 2008</conftitle>
   <confdates>August 12 - 15, 2008</confdates>
  </confgroup>
  <abstract>
    <para>Where should attribute constraints live? In an external schema? In the document's own
      metadata? In a separate file? Several possibilities are examined, raising lots of questions
      and offering a few answers.</para>
  </abstract>
  <author>
    <personname>
      <firstname>Syd</firstname>
      <surname>Bauman</surname>
    </personname>
    <personblurb>
      <para>Syd Bauman is the technical person at the Brown University Women Writers Project,
        where he has worked since 1990, designing and maintaining a significantly extended
        TEI-conformant schema for encoding early printed books. He has served as the North
```

Figura 13: Frammento di Docbook

```
<header id="title-block-header">
  <h1 class="title">Freedom to Constrain</h1>
  <p class="subtitle">where does attribute constraint come from,
    mommy?</p>
  <p class="author">Syd Bauman Syd Bauman is the technical person at the
    Brown University Women Writers Project, where he has worked since 1990,
    designing and maintaining a significantly extended TEI-conformant schema
    for encoding early printed books. He has served as the North American
    Editor of the Text Encoding Initiative Guidelines, has an AB from Brown
    University in political science, and has worked as an Emergency Medical
    Technician since 1983. Senior Programmer/Analyst Brown University Women
    Writers Project Syd_Bauman@Brown.edu
    http://www.stg.brown.edu/staff/syd.html</p>
</header>
```

Figura 14: Frammento di HTML

Nella prima immagine [Fig:13] possiamo osservare come si presenta il frammento di Docbook mentre nella seconda [Fig:14] la trasposizione in HTML. L'elemento *info* in Docbook non viene considerato in fase di conversione e vengono persi alcuni dati: non compare infatti l'abstract. Un altro problema è stato riscontrato con l'elemento *bibliomixed*.

```
<bibliography><!-- begin -->

  <title>Bibliography<footnote xml:id="bibnote">
      <para>Note: all Web references accessed as of 18 April 2008.</para>
    </footnote>
  </title>

    <bibliomixed xml:id="Z3919">
    "ANSI/NISO Z39.19-2005: Guidelines for the Construction, Format, and Management of Monolingual
    Controlled Vocabularies."
    <emphasis role="ital">U.S.A. National Information Standards Organization.</emphasis>
    25 July 2005.
    <link xlink:href="http://www.eric.ed.gov/ERICWebPortal/resources/html/help/Z39-19-2005.pdf"
    xlink:type="simple" xlink:show="new"
    xlink:actuate="onRequest">http://www.eric.ed.gov/ERICWebPortal/resources/html/help/Z39-19-2005.pdf</link>
    </bibliomixed>
```

Figura 15: Frammento di Docbook

```
<section class="level1">
  <h1>Bibliography<a href="#fn15" class="footnote-ref" id="fnref15" role="doc-noteref"><sup>15</sup></a></h1>
  <p>"ANSI/NISO Z39.19-2005: Guidelines for the Construction, Format, and
    Management of Monolingual Controlled Vocabularies." <em>U.S.A. National
      Information Standards Organization.</em> 25 July 2005.
      <a href="http://www.eric.ed.gov/ERICWebPortal/resources/html/help/Z39-19-2005.pdf">
      http://www.eric.ed.gov/ERICWebPortal/resources/html/help/Z39-19-2005.pdf</a>
  </p>
```

Figura 16: Frammento di HTML

Come si può osservare nella prima immagine [Fig:15], in formato Docbook, erano presenti molte più informazioni e la struttura stessa del documento era diversa. Mancano infatti nell'HTML [Fig:16] l'id dei singoli elementi facenti parte della bibliografia.
Si è quindi tentato di agire direttamente sul convertitore sfruttando gli strumenti disponibili. Alcuni dei problemi minori sono stati risolti con l'aggiunta di comandi che verranno illustrati nella sezione 4.3.2, altri invece sono stati riscontrati ad un livello nel quale non è stato possibile agire.
Come precedentemente spiegato, nella prima fase Pandoc genera un AST, il quale contiene tutti i dati estratti dal file in input. Analizzando questo AST è stato riscontrato che alcuni elementi o parti di testo venivano già omessi in questa fase; pertanto, non era possibile agire con un filtro.
Per risolvere le problematiche riscontrate, sono state necessarie due fasi di elaborazione ulteriori a quella di conversione che hanno reso l'utilizzo di questo convertitore non ottimale. Queste fasi verranno mostrate nella sezione 3.2.4.

Capitolo 3

Leshy: un ambiente Web per la conversione di documenti elettronici

Prima di procedere, è necessario introdurre alcuni concetti per comprendere il corretto funzionamento della struttura:

- **Collection**: è un insieme di documenti accomunati da alcune caratteristiche che li rendono idonei ad una analisi complessiva;
- **Metadati**: insieme di elementi che caratterizzano un determinato file; tra i metadati principali troviamo titolo, descrizione, data creazione, autore;
- **RASH**: è il formato dati con il quale Docudipity analizza i documenti, è quindi necessario che Leshy converta i documenti in input in questo formato;
- **Manifest**: è l'elemento necessario a Docudipity per comprendere la collezione che dovrà analizzare. É un documento in formato JSON contenente tutte le informazioni che caratterizzano la collezione analizzata e l'insieme di alcuni metadati dei documenti che ne fanno parte. Di seguito un esempio di come si presenta il manifest.

```
{
    "author": "Stefano Liverani",
    "collection_name": "Collection",
    "date": "2022-02-09",
    "description": "Articoli",
    "notes": "Contiene file Docbook",
    "docs": [
        {
            "name": "InformationOntologyDesign",
            "title": "Informal Ontology Design",
            "desc": "A Wiki-Based Assertion Framework",
            "year": "2008",
            "authors": "Murray Altheim ",
            "path": ""
        }
    ]
}
```


Figura 17: Esempio di Manifest

Parliamo ora del funzionamento generale dell'applicazione: in questo capitolo verranno illustrati tutti i passaggi da seguire per il corretto utilizzo di Leshy e, più nel dettaglio, i passaggi di conversione dei documenti e la compilazione del manifest.

3.1 Funzionamento generale

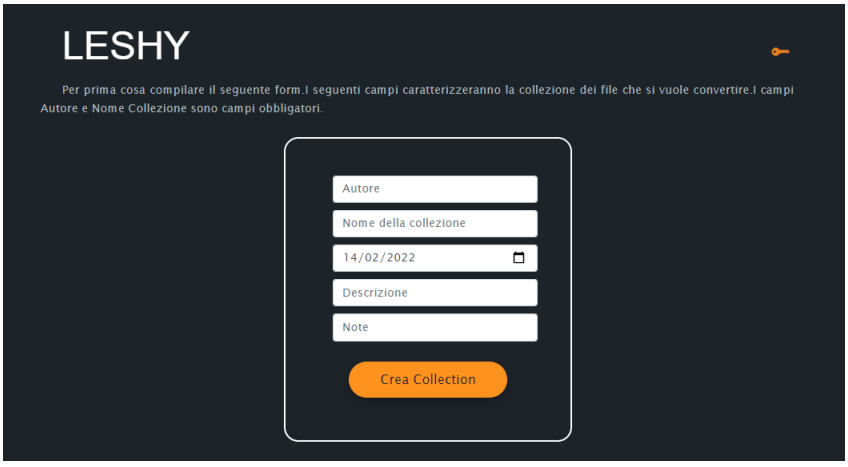

Figura 18: Home Page di Leshy

Prima di entrare nel dettaglio della struttura possiamo osservare nel seguente diagramma il funzionamento generale dell'app.

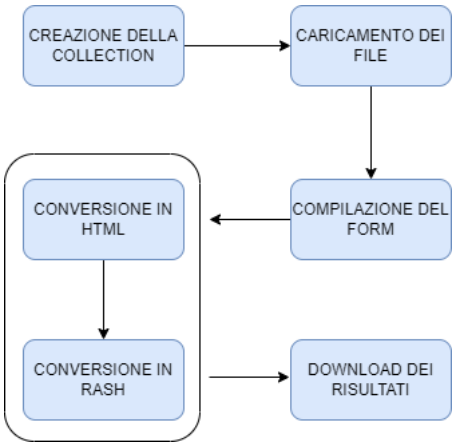

Figura 19: Flusso generale dell'applicazione

Come è possibile osservare dal diagramma [Fig: 19] il primo step consiste nella creazione di una Collection atta a caratterizzare il gruppo di documenti che saranno caricati nella fase successiva.
Dai tali documenti, saranno estratti dei metadati che andranno a popolare un form modificabile dall'utente. Prima della conversione questi dati verranno aggiunti al manifest.
Fatto ciò, Leshy procederà alla conversione prima in HTML, poi in RASH. In seguito, sarà possibile scaricare un pacchetto contenente i file convertiti e il manifest con tutte le informazioni raccolte durante le fasi precedenti.

3.2 Funzionamento nel dettaglio

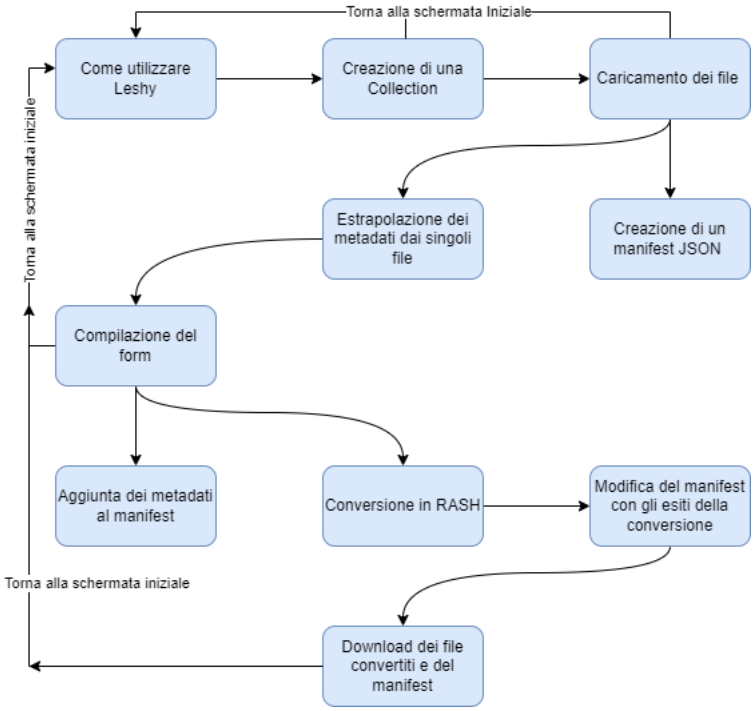

Figura 20: Dettaglio del flusso di Leshy

Entrando più nel dettaglio possiamo analizzare in questo diagramma tutti gli step che l'app prevede per svolgere le operazioni sopra citate.

3.2.1 Creazione di una Collection

In una prima fase sarà possibile inserire alcuni dati che caratterizzeranno la Collection [Fig:18].
Tra i dati disponibili l'autore e il nome della collezione sono campi obbligatori. La data è impostata al giorno corrente ma è possibile modificarla; descrizione e note aggiungono invece informazioni al manifest che verrà generato.

3.2.2 Caricamento dei file

In seguito alla compilazione del form verrà assegnato all'utente un numero che, unito al campo autore, caratterizzerà la propria collezione. D'ora in avanti l'utente verrà riconosciuto mediante questa combinazione.

Viene quindi generato un manifest in formato JSON che servirà a contenere tutte le informazioni utili relative alla collezione e ai file convertiti.

Figura 21: Esempio di dati estratti dal form iniziale

In questa fase vengono aggiunti i dati ottenuti dal form iniziale, ove presenti. Viene inoltre creato un campo denominato "docs" che conterrà la lista dei file caricati e alcuni metadati.

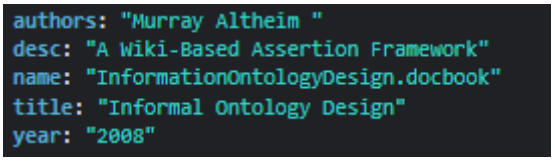

Figura 22: Metadati relativi ad un file

Ora è possibile caricare i file che si vuole convertire, facendo attenzione a caricare file dello stesso tipo e rispettando i formati disponibili. La dimensione massima dei singoli file è stata impostata a 5 MB, non vi è invece un limite di file per singola conversione. Nel caso in cui queste condizioni non venissero rispettate il sistema avviserà che non è possibile procedere alla conversione.
Durante la fase di caricamento dei file vengono anche analizzati e memorizzati alcuni metadati. Nello specifico per ogni documento vengono letti autore/i, descrizione, nome, titolo e anno di pubblicazione.

Figura 23: Schermata per il caricamento dei file

3.2.3 Compilazione del form

In questa schermata è possibile visualizzare l'elenco dei file correttamente caricati e un form precompilato contenente i dati ottenuti da ogni file. É possibile in questa fase modificare le informazioni del form, aggiungendo, modificando o rimuovendo alcuni dati.

Figura 24: Form per arricchire il manifest

Tutte le informazioni presenti nel form vengono aggiunte al manifest nella lista "docs". L'unico campo a non essere compilato è il "path" che indica il percorso del file convertito: questo dato verrà aggiornato in seguito alla conversione.

3.2.4 Conversione in RASH

Questa fase può essere suddivisa in vari step che vengono mostrati nella seguente figura [Fig:25]

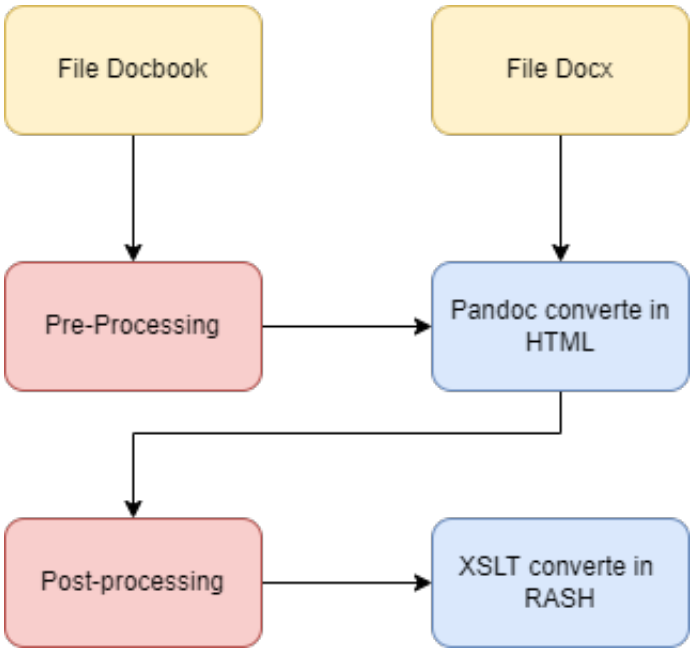

Figura 25: Step di conversione

Dopo il caricamento dei file e la compilazione del form viene avviato il processo di conversione. Come menzionato nei capitoli precedenti sono state necessarie alcune fasi intermedie tra le conversioni per permettere ai convertitori di generare file ben formati. Nello specifico si sono rese necessarie una fase di processamento prima della conversione in HTML, per risolvere le problematiche elencate nella sezione 2.5.4, e una fase prima della conversione in RASH.

- **Pre-Processing**: durante il caricamento dei file sul server vi è una fase in cui vengono estratti alcuni metadati. In quel momento, anche per ridurre il numero di letture del file, sono state inserite alcune modifiche per permetterne una visualizzazione più coerente in HTML. Sono stati modificati

il tag *info* e il tag *bibliomixed* rispettivamente in *chapter* e *section*, maggiori dettagli sull'implementazione di queste modifiche saranno mostrate nella sezione 4.3.2. Inoltre è stato osservato che nei casi in cui le immagini non fossero estraibili o non raggiungibili, il convertitore prevede la sostituzione dell'immagine con una *section* contente il testo dell'immagine. Per mantenere una più coerente struttura è stato quindi inserito uno step di verifica dell'immagine: in caso di errore il percorso viene sostituito con quello di una immagine di default presente sul Server. In questo modo la struttura di destinazione rimarrà invariata mantenendo intatti tutti i suoi elementi. I risultati sono migliorati notevolmente, come si può osservare dagli estratti dell'HTML sotto.

```
<body>
  <header id="title-block-header">
    <h1 class="title">Freedom to Constrain</h1>
    <p class="subtitle">where does attribute constraint come from,
      mommy?</p>
  </header>
  <section class="level1">
    <h1></h1>
    Balisage: The Markup Conference 2008
    August 12 - 15, 2008
    <blockquote>
      <p>Where should attribute constraints live? In an external schema? In
        the document's own metadata? In a separate file? Several possibilities
        are examined, raising lots of questions and offering a few answers.</p>
    </blockquote>
    Syd
    Bauman
    <p>Syd Bauman is the technical person at the Brown University Women
      Writers Project, where he has worked since 1990, designing and
      maintaining a significantly extended TEI-conformant schema for encoding
```

Figura 26: Conversione del Tag Info in HTML

```
<section class="level1">
  <h1>Bibliography<a href="#fn15" class="footnote-ref" id="fnref15" role="doc-noteref"><sup>15</sup></a></h1>
  <section id="Z3919" class="level2">
    <h2></h2>
    "ANSI/NISO Z39.19-2005: Guidelines for the Construction, Format, and
    Management of Monolingual Controlled Vocabularies."
    U.S.A. National Information Standards Organization.
    25 July 2005.
    http://www.eric.ed.gov/ERICWebPortal/resources/html/help/Z39-19-2005.pdf
  </section>
  <section id="BernersLee2001" class="level2">
    <h2></h2>
    Berners-Lee, Tim, James Hendler, and Ora Lassila. "The Semantic Web."
    Scientific American.
    2001.
    http://www.sciam.com/article.cfm?id=the-semantic-web
  </section>
```

Figura 27: Conversione del Tag Bibliomixed in HTML

- **Conversione in HTML**: ogni file viene convertito in HTML sfruttando Pandoc e il risultato viene memorizzato in una apposita cartella sul server; vengono anche estratte le eventuali immagini;

- **Post-Processing**: altre modifiche sono state invece svolte direttamente sull'HTML prodotto: si è reso necessario rimuovere alcuni elementi aggiunti dal convertitore e modificarne altri.
  Nello specifico sono stati rimossi il tag *style*, alcuni attributi del tag *html* e alcuni commenti. É stato inoltre necessario modificare la *src* delle immagini in quanto, in fase di estrazione e successivo salvataggio sul Server, veniva assegnato al tag *img* una *src* non coerente con l'effettiva posizione dell'immagine risultando non visibile sull'HTML prodotto. L'immagine sotto mostra un esempio:

```
<h1>Images</h1>
<p>Images can be of three main types. Inline images are images that are
  part of the normal text flow, like this image of a green dot <img
    src="495860Stefanoconverted/test3/media/image2.png" style="width:0.15in;height:0.15in" alt="dot_green.png" />.
  Inline images
  do not cause breaks in the text and are usually small in size.<img
    src="495860Stefanoconverted/test3/media/image3.png" style="width:1.06667in;height:1.06667in" /> The next
  category of image
```

Figura 28: Esempio di src da modificare

Per rendere coerente il percorso è stato necessario rimuovere dalla *src* tutto il percorso prima di *media*. Le immagini vengono infatti salvate in questa posizione rispetto al file convertito:

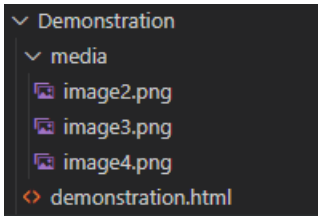

Figura 29: Esempio di una cartella di output

Alcune immagini invece, come quelle proveniente da un URL, vengono salvate nella stessa cartella del file; pertanto, sono state spostate nella cartella media in modo da rendere coerente la struttura.
Per terminare la conversione in RASH si è resa necessaria l'aggiunta di un ulteriore step: il file HTML modificato è stato elaborato insieme ad un file xslt contenente ulteriori passaggi di conversione, che verranno mostrati più nel dettaglio nel capitolo 4 alla sezione 4.3.2.

- **Conversione in RASH**: ogni file viene convertito in RASH sfruttando un pacchetto NodeJS denominato *xslt-processor*;

Al termine di questo processo, viene mostrato l'elenco dei file caricati e il loro stato di conversione.
L'esito positivo della conversione comporta anche l'aggiunta del path al manifest. In caso di esito negativo invece il file viene rimosso dal manifest.

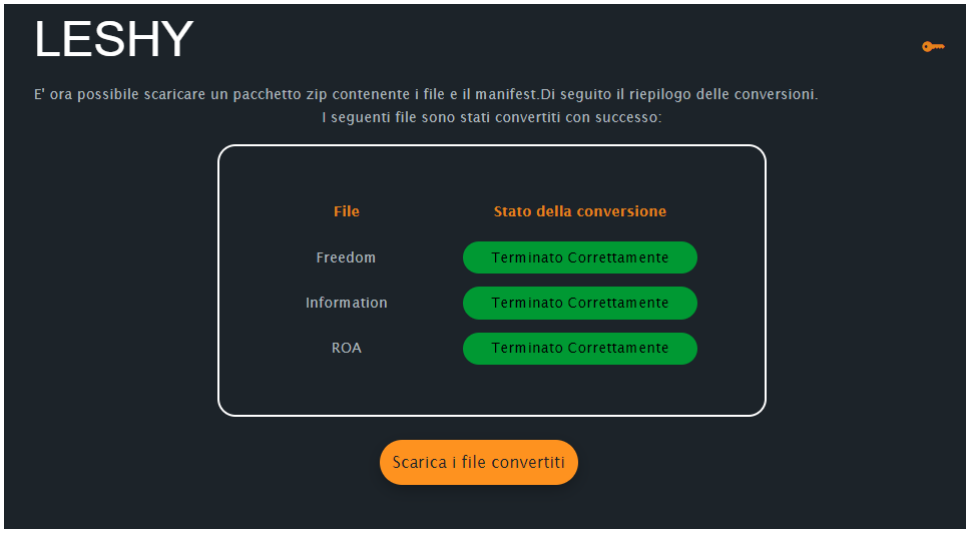

Figura 30: Schermata contenente l'esito delle conversione

3.2.5 Composizione del manifest

In questa sezione viene illustrata nel dettaglio la composizione del manifest che è stato generato dalle fasi precedenti [Fig:31]:

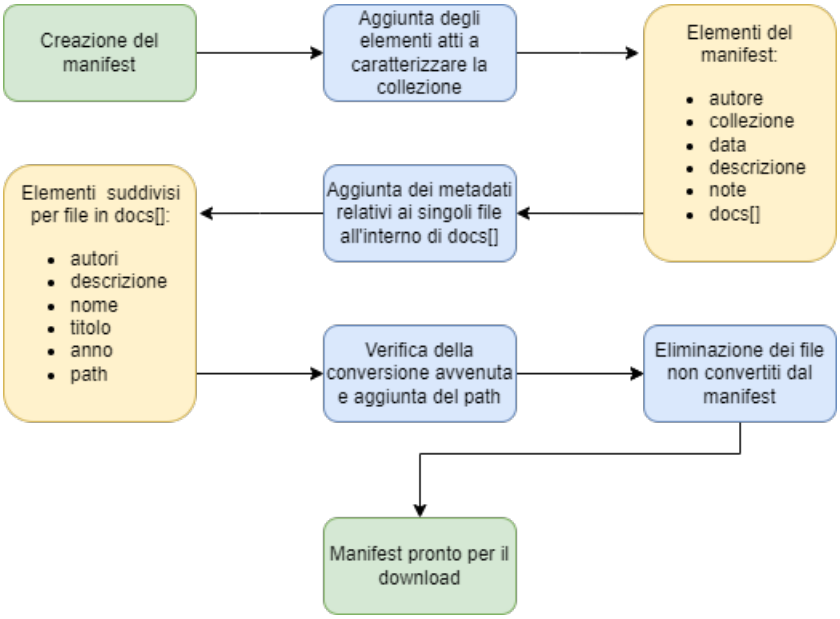

Figura 31: Composizione del manifest

La struttura del manifest alla fine del processo si presenta quindi come nella seguente immagine [Fig:32]:

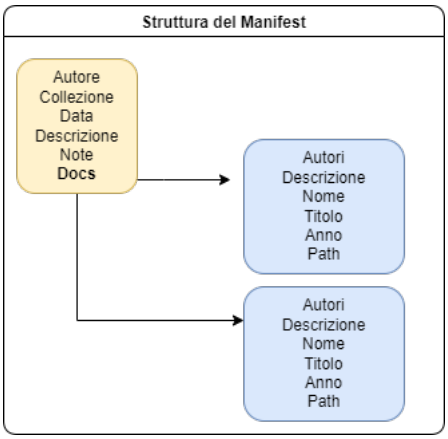

Figura 32: Struttura del manifest

3.2.6 Download dei file e del manifest

Questa è l'ultima fase: al momento del download tutti i file correttamente convertiti vengono aggiunti ad un file zip insieme al manifest e scaricati dall'utente. Sempre in questa fase vengono cancellati dal server tutti i file relativi a questa operazione.

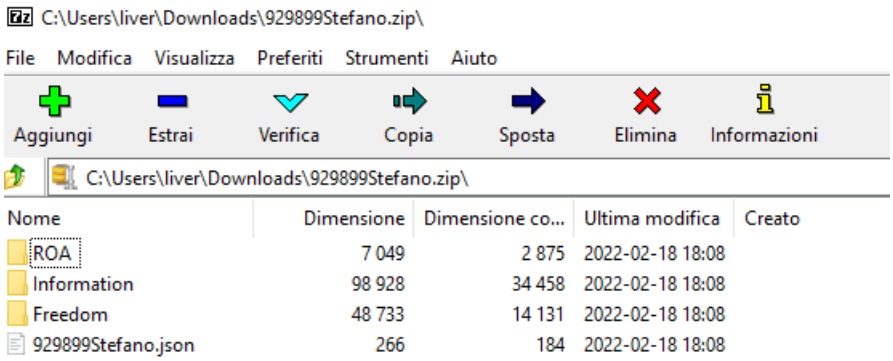

Figura 33: Esempio di pacchetto zip

Capitolo 4

Implementazione

In questo capitolo verrà illustrata nel dettaglio l'implementazione di Leshy. Saranno mostrati i linguaggi utilizzati, i pacchetti Node e tutte le funzioni principali di Client e Server.

4.1 Linguaggi, librerie e framework utilizzati

Per iniziare si introducono i linguaggi, le librerie e i framework utilizzati per la realizzazione dell'infrastruttura.

- **HTML**: utilizzato per la realizzazione della struttura generale della pagina web e per la sua impaginazione;
- **CSS**(Cascading Style Sheets): utilizzato per la formattazione degli elementi senza intaccare il file HTML;
- **JavaScript**: su Leshy è stato utilizzato insieme a NodeJS per la realizzazione dell'applicazione lato Server, per la modifica in tempo reale dell'aspetto della pagina lato Client e per la gestione dei file e dei metadati sul Server;
- **JQuery**: è una libreria JS progettata per semplificare la navigazione del DOM. Può essere utilizzato per trovare, selezionare, modificare gli elementi oppure assegnare eventi a determinate azioni. In questo progetto JQuery è stato utilizzato lato Client per la creazione di chiamate asincrone (AJAX) per l'invio o la ricezione di dati con il Server;
- **Bootstrap**: è un framework CSS che contiene una raccolta di strumenti per l'impaginazione, la grafica e lo stile che rendono veloce e altamente personalizzabile l'aspetto delle pagine web.
  Nell'infrastruttura è stato utilizzato per facilitare l'impaginazione e la realizzazione di alcuni aspetti della pagina.

4.2 NodeJS

NodeJS è un ambiente di runtime JavaScript open source. Nasce nel 2009 da V8, un motore sviluppato da Google per il browser Chrome.
Una delle sue peculiarità è quella di permettere a JavaScript di essere utilizzato non solo lato Client ma anche per la realizzazione di componenti Server.
Le enormi potenzialità di NodeJS sono dovute all'elevato numero di librerie disponibili, integrabili nell'applicazione attraverso il package manager fornito da Node, ovvero *npm*.
Mediante *npm* è possibile scaricare i pacchetti necessari ad una specifica esigenza mediante il comando *npm install* seguito dal nome del pacchetto. Verrà generata una cartella all'interno del proprio progetto, solitamente nominata *node_modules* contenente la libreria richiesta.
Di seguito sono illustrati nel dettaglio tutti i pacchetti utilizzati e il loro funzionamento:

- **Express**: uno dei più famosi pacchetti di Node. Permette di creare un server in pochissimi passaggi. É stato appunto utilizzato per la creazione della struttura base del server;

- **Express-fileupload**: è un middleware di Express per gestire l'upload dei file sul Server.
  Nello specifico è stato utilizzato per la gestione dell'upload dei file da parte degli utenti, sfrutta un sistema di cartelle temporanee prima dell'inserimento dei file nella cartella dedicata all'utente;

- **Path**: è utilizzato per lavorare con percorsi di file o cartelle presenti sul Server;

- **Node-Pandoc**: questo pacchetto rende possibile l'utilizzo di Pandoc nelle applicazioni web. Infatti, è in grado di eseguire le operazioni di Pandoc per la conversione dei documenti senza doverle effettuare da terminale.
  La sua funzione è quella di preparare gli elementi necessari a Pandoc per effettuare le conversioni. Pandoc, nella sua versione offline da terminale, necessita di ricevere come input il percorso di un file o una stringa (*src*) e le istruzioni che ne determinano il formato in ingresso, eventuali filtri e il formato in output (*args*). Questo pacchetto si occupa quindi di passare le due variabili (*src*) e (*args*) e permettere la conversione offline. Maggiori dettagli sul funzionamento del convertitore offline sono disponibili sul sito di Pandoc alla sezione (*Getting started with Pandoc*) [4]. Nel progetto Node-Pandoc è stato utilizzato per la realizzazione di un convertitore online;

- **Fs**: questo modulo fornisce un elevato numero di funzioni per accedere ed interagire con il file system. Non è necessario installare il pacchetto essendo già incluso nel core di NodeJS.
  Ha lo scopo di elaborare, modificare, leggere e scrivere file sul Server.

- **Zip-local**: permette di gestire file zip, estrarne il contenuto o comprimere file o gruppi di file.
  Nello specifico è stato utilizzato per la creazione del file zip da scaricare al termine delle conversioni contenente tutti i documenti convertiti e il manifest;

- **Unzipper**: permette di decomprire file zip; in particolare questo pacchetto è anche in grado di estrarre singoli file presenti all'interno di un file zip senza doverlo decomprimere tutto.
  Su Leshy è stato utilizzato per estrarre il file core.xml contenente i metadati dei file docx;

- **Del**: ha la funzione di cancellare file o cartelle. Permette inoltre di cancellare anche cartelle non vuote, cancellando ricorsivamente il contenuto delle cartelle prima di eliminare la cartella principale;

- **Xpath**: questo pacchetto permette di sfruttare le query xpath per la selezione di elementi all'interno di un file xml.
  Nello specifico è stato utilizzato per estrarre i metadati dai file Docbook;

- **node-fetch**: serve a verificare il funzionamento di un URL; nel progetto è utilizzato per verificare se le immagini presenti nei file Docbook hanno URL validi;

- **Xmldom**: è un DOMParser. Nel progetto è stato utilizzato per la visualizzazione dei file docbook, docx e HTML in DOM; in questo modo è stato possibile leggere, modificare, eliminare gli elementi per adattarli all'occorrenza;

- **Stream-promise**: utilizzato per concatenare output e risolvere la promise solo al termine di tutte le funzioni annidate.
  Nell'app è stato utilizzato per l'estrazione del core.xml dai file Docx.

- **Xslt-processor**: è in grado di processare file xslt. Nel progetto permette di convertire i file HTML in RASH sfruttando appunto un file xslt per le modifiche.

4.3 Organizzazione del progetto

Questa sezione ha lo scopo di illustrare i file creati, il loro ruolo e la disposizione delle cartelle all'interno del progetto.

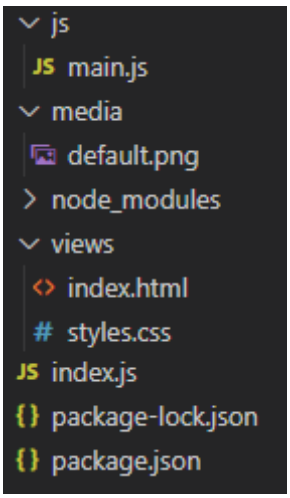

Figura 34: Struttura generale del progetto

Di seguito una breve illustrazione degli elementi principali:

- **index.js**: è il file che serve all'avviamento del Server; contiene tutte le funzioni che il server esegue su richiesta del Client;
- **main.js**: racchiude tutte le funzioni utili al Client per l'impaginazione e le richieste da effettuare al Server;
- **index.html**: è il file che viene utilizzato per la visualizzazione della pagina; questo file viene modificato in maniera dinamica da JavaScript;
- **package_lock.json**: contiene dati relativi ai pacchetti node utilizzati per il progetto, indicando informazioni utili al loro funzionamento;
- **node_modules**: è la cartella che contiene tutti i pacchetti Node che sono stati scaricati per la realizzazione dell'applicazione.

Mostriamo ora i dettagli implementativi del progetto.

4.3.1 Client Side

In questa sezione verranno illustrate le chiamate effettuate dal Client per ottenere/fornire informazioni al/dal Server.

- **createUser**: avvia la funzione che permette al Server di generare la collection e assegnarla all'utente;
- **uploadToFS**: mediante questa funzione avvengono due chiamate al server: la prima, uploadToFS, si occupa di inviare i dati ottenuti dal form che permette il caricamento dei file da convertire; la seconda, createJSON, invia tutti i dati inseriti nel form atti a caratterizzare la collezione. Di seguito viene mostrata la chiamata createJSON:

```
let author = document.getElementById("author").value;
let name = document.getElementById("name").value;
let today = document.getElementById("currentDate").value;
let description = document.getElementById("description").value;
let notes = document.getElementById("notes").value;
let formAutore = {
    author: author,
    name: name,
    date: today,
    description: description,
    notes: notes,
    globalCollectionName: GlobalCollectionName
};
$.ajax({
    url: '/createJSON',
    type: 'POST',
    data: formAutore,
    dataType: 'json',
    contentType: 'application/x-www-form-urlencoded',
    success: function () {
        console.log("Json creato");
    },
    error: function(){
        console.log("Errore durante la creazione del Json");
    }
});
```

Figura 35: Chiamata AJAX createJSON

- **uploadFolder**: questa chiamata è utilizzata ogni qual volta è necessario controllare i file presenti nella cartella di upload;
- **sendDocbookData/sendDocxData**: queste due chiamate richiedono al server di estrarre i metadati dai documenti e fornirli al Client che si occuperà, mediante la funzione populateForm(), di visualizzare i dati ottenuti:

```
$.ajax({
    url: '/sendDocbookData',
    type: 'POST',
    data: gc,
    processData: false,
    contentType: false,
    success: function (result) {
        console.log("Filelist è stata ricevuta:", result);
        populateForm(result);
        document.getElementById("fileUploadList").style.display = "inline-block";
    }
});
```

Figura 36: Chiamata AJAX sendDocbookData

- **toEndPage**: questa chiamata permette di ottenere l'elenco dei file da convertire e crea la cartella che conterrà i file convertiti; nella *success* di questa chiamata verrà creata la tabella che conterrà il riepilogo dei file per i quali sarà avviata la conversione:

```
let resultTable = document.getElementById("resultTable");
for (let index = 0; index < data.length; index++) {
    if (data[index].split('.')[1] != 'xml') {
        let row = document.createElement("tr");
        row.setAttribute("class", "fileRow");
        let fileUp = document.createElement("td");
        fileUp.setAttribute("class", "fileUp");
        fileUp.innerHTML = data[index].split('.')[0];
        let fileStat = document.createElement("td");
        fileStat.setAttribute("class", "fileStat");
        fileStat.setAttribute("id", data[index].split('.')[0]);
        fileStat.style.backgroundColor="#e6b800";
        fileStat.innerHTML = "Conversione in corso..";
        row.appendChild(fileUp);
        row.appendChild(fileStat);
        resultTable.appendChild(row);
    }
}
```

Figura 37: Crezione della tabella dei risultati

In seguito verranno effettuate singole chiamate per ogni file che aggiorneranno questa tabella a seconda dell'esito della conversione;

- **toHTML**: con questa chiamata viene avviata la funzione sul Server che converte il file in HTML sfruttando Pandoc; i dettagli di questa funzione verranno illustrati nella sezione 4.3.2;

- **toEditHTML**: avvia il processo di modifica dell'HTML ottenuto dalla chiamata precedente;
- **updateJSON**: si occupa di fornire al Server i metadati relativi ai singoli file, eventualmente modificati dall'utente nell'apposito form. Il Server si occuperà poi di aggiungere i file con i relativi metadati al manifest:

```
for (let index = 0; index < data.length; index++) {
    console.log("name:", data[index]);
    let name = data[index].split('.');
    console.log("name: ", name[0]);
    let obj = {
        name: name[0],
        title: document.getElementsByClassName(name[0] + "inputTitle")[0].value,
        desc: document.getElementsByClassName(name[0] + "inputDesc")[0].value,
        year: document.getElementsByClassName(name[0] + "inputYear")[0].value,
        authors: document.getElementsByClassName(name[0] + "inputAuthors")[0].value,
        path: "",
        collection: GlobalCollectionName
    }
    //singola aggiunta di ogni file
    $.ajax({
        url: '/updateJSON',
        type: 'POST',
        data: obj,
        dataType: 'json',
        contentType: 'application/x-www-form-urlencoded',
        success: function () {
            console.log("UpdateJson Done!");
        },
        error: function(err){
            console.log("Errore:", err);
        }
    })
}
```

Figura 38: Chiamata AJAX updateJSON

- **download**: questa chiamata avvia il processo di download dei risultati; nella *success* viene effettuato il download del file zip ottenuto e l'eliminazione di tutti i file residui presenti sul Server:

```
$.ajax({
    url: '/download',
    type: 'POST',
    data: gc,
    processData: false,
    contentType: false,
    success: function (data) {
        var element = document.createElement('a');
        element.setAttribute('href', data);
        element.setAttribute('download', '');
        element.style.display = 'none';
        document.body.appendChild(element);
        element.click();
        document.body.removeChild(element);
        deleteUpload();
        deleteConverted();
```

Figura 39: Chiamata AJAX download

4.3.2 Server Side

Entriamo ora nel dettaglio delle funzioni che rispondono alle chiamate del Client.

- **createJSON**: crea un file JSON, detto manifest, e inserisce le informazioni ottenute dal Client;
- **updateJSON**: aggiunge le informazioni relative ad un file alla lista docs del manifest;
- **uploadToFS**: carica i file sulla cartella del Server, sfrutta express-fileupload per spostare i file da una cartella temporanea a quella definitiva.

```
app.post('/uploadToFS', async function (req, res) {
  let targetFile = req.files.file;
  let fileName = targetFile.name;
  fileName = fileName.toString().split('.');
  fileName[0] = fileName[0].replace(/ /g, '');
  let directoryUpload = __dirname + "/" + req.body.collection;
  try {
    //verifica l'esistenza della cartella
    if (!fs.existsSync(directoryUpload)) {
      fs.mkdirSync(directoryUpload)
    }
    targetFile.mv(directoryUpload + "/" + fileName[0] + "." + fileName[1], err => {
      if (err)
        return res.status(500).send(err);
      res.send('File uploaded!');
    });
  } catch (err) {
    console.error(err)
  }
});
```

Figura 40: Funzione per il caricamento dei file sul server

- **uploadFolder**: restituisce al Client l'elenco dei file nella cartella di upload;
- **toEndPage**: restituisce al Client l'elenco dei file nella cartella di upload e crea la cartella della collezione per i file da convertire;
- **toHTML**: questa funzione si occupa di convertire i file presenti nella cartella di upload in HTML sfruttando il convertitore Pandoc. Per prima cosa la funzione crea una sotto cartella all'interno della collezione che conterrà il file convertito ed eventuali media estratti dal file. Successivamente vengono creati i due elementi necessari a Pandoc per effettuare la conversione, ovvero *src e args*.
  Src non è altro che il percorso del file da convertire, *args* contiene invece tutte le indicazioni su come convertire il file. Di seguito mostriamo la composizione di *args*:

```
var args = '-f docbook' + ' -t html -o ' + destination + filetype[0] + ".html" +
  ' --section-divs --standalone --extract-media=' + uploadFolder + 'converted/' + filetype[0];
```

Figura 41: Contiene i comandi da passare a Pandoc per la conversione

  - *-f*: indica il formato in input;
  - *-t*: indica in quale formato si vuole convertire;
  - *-o*: specifica il percorso e il nome del file che si vuole salvare;

  - - *-section-divs*: comando opzionale, aggiunto per permettere di mantenere la gerarchia dei documenti originali; con questo comando infatti vengono aggiunte section che indicano il livello di annidamento del testo;
  - - *-standalone*: permette di creare un output funzionante con tag di apertura html e metadati;
  - - *-extrac-media*: indica il percorso nel quale si vogliono salvare le eventuali immagini estratte dai documenti.

- **SendDocbookData/SendDocxData**: queste due funzioni si occupano di estrarre dai documenti i metadati necessari a completare il manifest.
  Per l'estrazione dei metadati da Docx si fa affidamento ad una funzione ausiliaria denominata extractFromCore in cui si decomprime il file e si estraggono i dati dal core.xml.
  Per DocBook invece si fa affidamento al pacchetto NodeJS *xpath*, di seguito l'estratto del codice:

```
const titleArray = xpath.select('//*[local-name() = \'title\']//text()', doc)
const descArray = xpath.select('//*[local-name() = \'subtitle\']//text()', doc)
const year = xpath.select('string(//*[local-name() = \'confdates\'])', doc)
const authors = xpath.select('//*[local-name() = \'personname\']//text()', doc)
```

Figura 42: Estrazione metadati da DocBook

  Inoltre le funzioni SendDocbookData e SendDocxData implementano anche le fasi di pre-processing. Di seguito viene mostrato un esempio di come è stato modificato il tag info dei file Docbook:

```
let info = doc.getElementsByTagName('info');
for (let i = 0; i < info.length; i++) {
  info[i].tagName = info[i].tagName.replace(/info/g, "chapter");
}
```

Figura 43: Modifica del Tag info

- **toEditHTML**: questa funzione si occupa invece della fase di post-processing dei file HTML ottenuti dalla conversione. Vengono rimossi alcuni tag e viene corretta la *src* delle immagini, mostriamo qui il codice per la correzione del percorso delle immagini:

```
for (let j = 0; j < images.length; j++) {
  let newSrc = images[j].getAttribute('src');
  newSrc = newSrc.split("converted/" + file + "/")[1];
  images[j].setAttribute('src', newSrc);
}
```

Figura 44: Correzione del percorso delle immagini

- **toRash**: questa funzione si occupa di effettuare l'ultimo step di conversione da HTML a RASH; in questa fase viene sfruttato il pacchetto NodeJS *xslt-processor*. Viene utilizzato un file xslt per effettuare le modifiche necessarie a rendere l'HTML un file RASH.
  Questa chiamata permetterà, in seguito alla conversione, di effettuare tutti gli step necessari all'aggiornamento del manifest, con l'aggiunta del percorso del file convertito e la restituzione all'utente dell'esito della conversione;

- **download**: si occupa di rimuovere dal manifest tutti i file che non sono stati convertiti correttamente e avvia il processo di compressione della cartella destinata all'utente.

4.3.3 Funzioni ausiliarie

Di seguito alcune funzioni ausiliarie:

- **deleteUpload(), deleteConverted(), deleteJSON()**: queste tre funzioni effettuano le chiamate al Server che permettono rispettivamente di eliminare i file caricati dall'utente, i file convertiti e il manifest;
- **manualDelete()**: questa funzione permette di avviare il processo di eliminazione di tutti i file generati da qualsiasi utente che presentano ancora file residui sul Server. Questa funzione è protetta da password, solo l'amministratore può quindi utilizzarla. É stata implementata per permettere facilmente all'amministratore di rimuovere eventuali residui dovuti a operazioni terminate in maniera anomala da parte dell'utente o del Server;
- **reloadPage(), reloadAndDelete(), reloadAndDeleteAll()**: queste tre funzioni permettono, in fase di interruzione dell'operazione, mediante ritorno alla pagina iniziale, di eliminare tutti i file residui sul Server generati da quell'utente;
- **zipFolder()**: funzione che permette di comprimere la cartella contenente tutti i file da scaricare:

```
function zipFolder(directoryConv, fileName) {
  return new Promise(resolve => {
    zipper.sync.zip(directoryConv).compress().save(fileName + ".zip")
    let fileToDownload = __dirname + "/" + fileName + ".zip";
    console.log("fileToDownload:", fileToDownload);
    resolve(urlFolder + fileName + ".zip");
  })
}
```

Figura 45: Creazione del file Zip da scaricare

Capitolo 5

Valutazione e Conclusioni

In questo capitolo ci occuperemo di analizzare alcuni test effettuati per verificare il funzionamento di Leshy, le criticità, i limiti e confrontare i risultati ottenuti. Infine verranno tratte le conclusioni e mostrati eventuali sviluppi futuri.

5.1 Valutazione

I test sono stati effettuati su un computer con processore i7-5500U 2.40 GHz, RAM 12GB, SSD 256GB. Il sistema operativo utilizzato è Windows 10 Home; il server è stato avviato dal terminale di Visual Studio Code e le prove sono state effettuate su Chrome e Edge.
Verranno analizzati separatamente nelle prossime sezioni il comportamento dell'applicazione con i file Docbook e Docx.

5.1.1 Analisi dei risultati: Docbook

Tutti i file testati in questa fase hanno dimensione massima di 100KB. Nella seguente tabella vengono visualizzati i risultati.

N° file/finestra	1 finestra		2 finestre		3 finestre	
	Tempo	File Convertiti	Tempo	File Convertiti	Tempo	File Convertiti
5	<1s	5	<2s	10	<5s	15
10	<2s	10	<6s	20	<8s	30
20	<5s	20	<13s	40	<15s	60
30	<8s	30	<16s	60	<25s	90

Figura 46: Esiti conversione: Docbook ->RASH

I test sono stati eseguiti a partire da una finestra di conversione fino ad arrivare a tre in parallelo. Osservando la tabella [Fig: 46] si può notare che l'aumento del tempo di conversione è pressoché proporzionale al numero di file da convertire; non vi sono pertanto da segnalare rallentamenti anomali. Ogni test è stato ripetuto 3 volte, ottenendo sempre lo stesso risultato, con la variazione di 1/2 secondi in base alla dimensione dei file scelti. Il processo è rallentato leggermente, impiegando qualche secondo in più per il completamento delle operazioni all'aumentare del numero dei file e delle finestre in parallelo. Sono stati registrati alcuni rallentamenti nella conversione all'aumentare della dimensione dei file e con presenza di hyperlink all'interno del testo. La presenza di immagini non ha invece modificato visibilmente il tempo di conversione. In ogni caso tutte le operazioni hanno portato a termine le fasi correttamente.

L'utilizzo dei convertitori Pandoc e XSLT si sono rivelati una buona soluzione per questo formato, pur dovendo effettuare alcuni aggiustamenti ai file in input,come segnalato nella sezione 4.3.2.

5.1.2 Analisi dei risultati: Docx

Gli stessi test effettuati con Docbook sono stati eseguiti anche con il formato Docx. A differenza di Docbook, in questa fase sono state riscontrate diverse problematiche, probabilmente dovute alla complessità del formato stesso.
In un caso specifico Pandoc ha interrotto la conversione restituendo il seguente messaggio ad ogni tentativo di conversione del file in oggetto: "Error: couldn't parse docx file: DocxError". Un'ipotesi della causa di questo errore è la compromissione del file, sebbene questo sia visualizzabile su Office. Infatti, provando a copiare manualmente tutto il contenuto in un nuovo documento e procedendo alla conversione di quest'ultimo, l'errore non si presenta, portando a termine correttamente il processo.
In seguito sono stati effettuati ulteriori test, inserendo un maggior numero di documenti e aumentandone notevolmente la complessità in termini di lunghezza e differenza di elementi.
É stato riscontrato un eccessivo rallentamento, verificatosi soprattutto per file contenenti un numero di pagine elevato (80/100). Analizzando quindi i vari step di conversione è stato osservato che la maggior parte dei rallentamenti è avvenuta durante la fase finale, quella della conversione a RASH mediante il foglio XSLT. Anche il convertitore Pandoc ha presentato rallentamenti, seppur meno evidenti, passando da una conversione pressoché istantanea ad un tempo di circa dieci secondi per i file più complessi.
Si è pensato quindi di impostare un limite al numero di pagine/caratteri per ogni

documento da verificare prima della conversione. Questa operazione avrebbe comportato l'estrazione del numero di pagine/caratteri da un file xml ottenuto dal Docx mediante unzip. L'idea è stata scartata per evitare di aggiungere ulteriori step che avrebbero appesantito la struttura.
Pertanto, in questa fase di test è stato deciso di impostare il tempo di conversione di Pandoc a 5 secondi, dopo i quali il processo di conversione viene interrotto per evitare un ulteriore rallentamento durante la conversione a RASH. Tramite questa soluzione è stato possibile procedere con ulteriori prove.
Di seguito vengono mostrate due tabelle. I file relativi alla prima tabella hanno dimensione inferiore a 50kB, nella seconda tabella sono invece stati utilizzati file di dimensione compresa tra 200kB e 1300kB. Tutti i test sono stati svolti con il tempo di conversione limitato a 5 secondi su Pandoc.

N° file/finestra	1 finestra		2 finestre	
	Tempo	File Convertiti	Tempo	File Convertiti
5	<2s	5	<3s	10
10	<20s	10	<6s	20

Figura 47: Esiti conversione: Docx ->RASH (Dimensione file: <50kB)

Si può osservare nella figura [Fig: 47] un elevato aumento del tempo di conversione nel caso di caricamento di 10 file. Analizzando i documenti è stato osservato che la conversione di 9 file ha impiegato un tempo proporzionato al caso precedente mentre il decimo file ha impiegato più di 10 secondi nella fase di conversione HTML ->RASH. Il file in questione ha un numero di pagine superiore a 100 che, come evidenziato sopra, porta a un rallentamento notevole nella fase finale di conversione.
Analizziamo ora la conversione di file con dimensione compresa tra 200kB e 1300kB.

N° file/finestra	1 finestra		2 finestre	
	Tempo	File Convertiti	Tempo	File Convertiti
5	<30s	2	<35s	3

Figura 48: Esiti conversione: Docx ->RASH (Dimensione file: 200kB-1300kB)

Dal primo test possiamo osservare come su 5 file solo 2 siano stati correttamente convertiti. Gli altri 3 sono stati interrotti avendo impiegato più di 5 secondi per la conversione Docx ->HTML. Nello specifico i file convertiti avevano dimensione 210kB e 1350kB, quelli non convertiti 220kB, 370kB e 650kB. Analizzandoli si è

potuto osservare che i file convertiti presentavano un numero di pagine ridotto: la dimensione elevata è infatti dovuta alla presenza di immagini e tabelle. I file non convertiti invece contenevano un elevato numero di hyperlink e un numero di pagine superiore a 60. Ripetendo i test con 2 finestre parallele sono stati riscontrati risultati simili.
I file HTML ottenuti da Pandoc risultano ben formati. Analizzando invece tutti i RASH ottenuti in entrambe le fasi si è potuto osservare che alcuni di essi presentano errori nella struttura che ne impediscono una corretta visualizzazione da browser.

5.2 Conclusioni

L'obiettivo di questa tesi è quello di fornire un'applicazione in grado di testare il convertitore Pandoc su diversi formati e creare uno strumento utile alla gestione e preparazione dei documenti per l'utilizzo su Docudipity.

Il progetto è quindi iniziato dall'analisi di Docudipity e del formato richiesto, ovvero RASH. Una volta comprese le necessità di questi ultimi si è passati alla scelta del convertitore. É stato selezionato Pandoc in quanto permetteva di usufruire di tutte le sue funzioni in maniera gratuita e di integrarsi facilmente all'interno dell'applicazione, permettendo inoltre di personalizzare la fase di conversione. Analizzando i formati disponibili su Pandoc, si è scelto di testare l'applicazione con DocBook e Docx. Per come è stato progettato l'ambiente e per l'architettura stessa di Pandoc, l'applicazione è facilmente estensibile ad altri formati, rendendo Leshy molto versatile.
Pandoc d'altro canto ha evidenziato alcune criticità in fase di utilizzo. Per poter ottenere una buona conversione è stato necessario inserire alcune fasi di pre e post-processing, in modo da ottenere un documento ben formato e convertibile in RASH, rendendo l'utilizzo di questo convertitore non ottimale. Sarà necessario effettuare un test più approfondito sull'API di Pandoc e sul funzionamento con altri formati.
Successivamente è stata implementata l'interfaccia utente per permettere di caricare i file, creare il manifest ed effettuare tutto il processo di conversione. In questa fase è stata implementata anche l'estrazione dei metadati dai documenti per l'aggiunta di informazioni al manifest.
Infine è stato aggiunto il convertitore XSLT, per convertire gli HTML ottenuti da Pandoc in formato RASH. In questa fase è stata testata la conversione completa a partire dal formato Docbook, ottenendo buoni risultati, e la conversione in via preliminare dal formato Docx, per la quale saranno necessarie ulteriori verifiche. Sarà fondamentale approfondire quest'ultima tipologia di conversione per effettuare dei miglioramenti e rimuovere i limiti riscontrati in questa fase.

Tra gli sviluppi futuri per quest'applicazione, oltre a quelli già citati in precedenza, vi è sicuramente la possibilità di inserire all'interno di Leshy nuovi convertitori allo scopo di ampliare i formati disponibili e trovare alternative valide a Pandoc.
Si può inoltre aggiungere un sistema di login per permettere all'utente di personalizzare la configurazione di Leshy, ad esempio impostando un convertitore predefinito o precompilare i form per la popolazione del manifest.

L'obiettivo prefissato di creare un'applicazione in grado di elaborare collezioni di documenti è stato raggiunto, auspicando che una futura implementazione possa esterne le potenzialità con la sostituzione o l'aggiunta di convertitori e formati.

Bibliografia

[1] A. Di Iorio, A. Gonzalez-Beltran, F. Osborne, S. Peroni, F. Poggi, and F. Vitali. It rocs!: The rash online conversion service. https://dl.acm.org/doi/10.1145/2872518.2889408#sec-ref, 2016.

[2] A. Di Iorio, A.G. Nuzzolese, F. Osborne, S. Peroni, F. Poggi, M. Smith, F. Vitali, and J. Zhao. The rash framework: enabling html+ rdf submissions in scholarly venues. https://www.cnr.it//prodotto/i/366590, 2015.

[3] Lunaweb-GmbH. Cloudconvert. https://cloudconvert.com/docx-to-html, 2012.

[4] John MacFarlane. Getting started with pandoc. https://pandoc.org/getting-started.html, 2006.

[5] John MacFarlane. Pandoc. https://pandoc.org/, 2006.

[6] Kevin Maiani. Progettazione e sviluppo di un ambiente di lettura e confronto tra articoli scientifici. https://amslaurea.unibo.it/23292/, 2021.

[7] Stefano Notari. Progettazione e sviluppo di un'api rest per un ambiente di lettura e confronto tra articoli scientifici. https://amslaurea.unibo.it/23505/, 2021.

[8] S. Peroni, F. Osborne, A. Di Iorio, A.G. Nuzzolese, F. Poggi, F. Vitali, and E. Motta. Research articles in simplified html: a web-first format for html-based scholarly articles. peerj computer science 3:e132. https://doi.org/10.7717/peerj-cs.132, 2017.

[9] F. Poggi, P. Ciancarini, A. Di Iorio, S. Peroni, and F. Vitali. Exploiting coordinated views for scholarly reading and analysis. in 25th international distributed multimedia systems conference on visualization and visual languages, dmsviva 2019 (vol. 2019, pp. 113-124). knowledge systems institute graduate school, ksi research inc. https://iris.unimore.it/handle/11380/1200461, 2019.

[10] A. Ragaad and H. Shah Rukh. Visualizing software structures through enhanced interactive sunburst layout. in proceedings of the international working conference on advanced visual interfaces, avi '16, pages 288–289. https://dl.acm.org/doi/10.1145/2909132.2926066, 2016.

[11] Enric Shinn and Licensed under the MIT License. node-pandoc. https://www.npmjs.com/package/node-pandoc, 2016.

[12] Uberdosis. Alldocs. https://alldocs.app/, 2022.

[13] W3jar. Node js file upload using express-fileupload module. https://www.w3jar.com/node-file-upload-with-express-js/, 2019.

[14] Zamzar-Ltd. Zamzar. https://developers.zamzar.com/, 2006.