Un mediafilter LaTeX per DSpace

Emmanuele Somma

Sommario

L'articolo presenta un semplice programma di filtraggio (*mediafilter*) per permettere la trasformazione in PDF di pubblicazioni LaTeX depositate in un repository DSpace. La soluzione presentata permette l'integrazione nella pubblicazione anche dei metadati registrati nella scheda archivistica, che oltretutto possono essere utilizzati come caratteristiche dell'impaginazione o per guidare la compilazione.

Abstract

This article shows a simple filtering program (*mediafilter*) to turn LaTeX sources stored in a DSpace repository into PDF. The solution also allows to use metadata, stored in the archive cards, into the publication itself, as parameters for the publishing layout or the compilation process.

1 Introduzione

DSpace è una piattaforma di archiviazione e gestione degli oggetti digitali (articoli, libri, tesi, dataset, ecc.) usata come repository istituzionale o come servizio archivistico da enti e organizzazioni, in particolare in campo universitario o governativo (De Robbio (2002)).

DSpace è un programma server-side Java che interagisce con gli utenti attraverso un'interfaccia web e offre agli utenti le tipiche operazioni di navigazione nel contenuto e ricerca delle schede archivistiche. È programmabile, molto configurabile ed estendibile aggiungendo moduli scritti in linguaggio Java o Jython. Chi vuole contribuire alla collezioni documentali può registrare in autonomia nuove schede e gli amministratori possono gestire flussi di lavorazione, anche complessi, attraverso l'interfaccia amministrativa del repository, delle collezioni e delle comunità (collezioni di collezioni). Esiste anche un insieme di programmi accessibili sulla linea di comando del server che permette agli amministratori di compiere le operazioni più sofisticate come importazioni ed esportazioni, reindicizzazioni, migrazioni e molte altre.

Le schede archivistiche registrate in DSpace si riferiscono agli *elementi* (in inglese *item*) e collezionano un esteso insieme di metadati in formato Dublin Core (e altri schemi aggiuntivi). Ad ogni elemento sono collegati i file da conservare (denominati *bitstream* per sottolineare l'indipendenza dal formato).

DSpace non prevede che i file siano conservati con uno specifico formato (ad esempio PDF o DOC), dunque la scelta del tipo dei file dipende dalla politica degli amministratori del repository in merito all'uniformità del contenuto, economia gestionale e fruibilità dell'archivio. Si pone quindi il problema di fare delle scelte, di natura essenzialmente editoriale, su quali file conservare nell'insieme di quelli utili.

Solitamente sono conservati i file definitivi pubblicati (è il caso delle tesi, ad esempio), oppure i preprint (nel caso delle pubblicazioni su riviste esterne) o addirittura scansioni (nel caso di libri storici). Negli ultimi tempi i repository sono usati anche per conservare i dataset usati nella ricerca e persino il software realizzato.

L'articolo presenta il caso ipotetico di una collezione documentale per cui i gestori hanno deciso di offrire agli utenti finali l'accesso alle pubblicazioni in un formato comodo per la fruizione (ad esempio PDF), ma di conservare come fonte primaria del documento il sorgente LaTeX. È compito del server DSpace trasformare questo sorgente in un documento PDF.

Questa trasformazione *automatica* è gestita attraverso la realizzazione di un breve programma Java messo all'interno dell'insieme dei cosiddetti *mediafilter* di DSpace (Donohue (2016)). Viene prima proposta una soluzione minimale e successivamente considerate alcune semplici estensioni.

2 I *mediafilter* di DSpace

Un *mediafilter* è un *oggetto* dell'applicazione DSpace, derivato dalla classe MediaFilter, che agisce su un dato formato di *bitstream* in ingresso (che nel caso in esame è un file LaTeX con estensione .tex) e produce un bitstream in uscita (qui un file .pdf prodotto dalla compilazione con pdflatex).

All'interno del codice sorgente di un *mediafilter* devono essere definiti i metodi dell'oggetto derivato dalla classe MediaFilter che realizzano la trasformazione del bitstream dall'ingresso all'uscita; in questo senso si può dire che il programma sia un *filtro* da un *media* all'altro (*mediafilter* appunto). Se l'operazione è eseguita senza errori, il bitstream prodotto è registrato nella scheda archivistica insieme a quello d'ingresso con un nome, una descrizione e un tipo adeguato.

È possibile mettere in esecuzione un *mediafilter* manualmente con il comando :

```
1 [dspace]/bin/dspace filter -media \
```

```
2                                   -i <element>
```

oppure attraverso un processo temporizzato (cron).

Se tra i bitstream presenti in un *elemento* non si trova quello nel formato prodotto dal *mediafilter* in esame (o se è usata sulla linea di comando l'opzione `-f` o `-force`) allora il *mediafilter* viene messo in esecuzione e viene tentata la generazione automatica.

La creazione di un *mediafilter* avviene attraverso l'estensione della classe `MediaFilter` presente nel package `org.dspace.app.mediafilter` con la definizione obbligatoria di alcuni metodi:

```
1  public class Tex2PdfFilter extends
       Mediafilter {
2    public String getFilteredName(String
       oFileName) {
3      // Definisce il nome del file di
         output
4      return  oldFilename.replaceFirst(".
         tex$",".pdf");
5    }
6    public String getBundleName() {
7      // Definisce il nome dell'insieme dei
         file che devono contenere il file
         di output
8      return "ORIGINAL";
9    }
10   public String getFormatString() {
11     // Definisce la stringa di formato del
         file di output
12     return "PDF";
13   }
14   public String getDescription() {
15     // Definisce la stringa di descrizione
         del file di output
16     return "Converted␣PDF";
17   }
18   public InputStream getDestinationStream(
       InputStream source)
19         throws Exception {
20     // Crea il file di output
21     ...
22   }
23   public void postProcessBitstream(Context
       c, Item item, Bitstream
       generatedBitstream) throws Exception
       {
24     // Completa l'operazione
25     ...
26   }
27 }
```

È obbligatorio implementare questi metodi per definire completamente il *mediafilter*.

3 Implementazione dei metodi base

I due metodi principali della classe creata sono `getDestinationStream()` e `postProcessBitstream()` e la loro implementazione non è riportata nel precedente listato. Tutti gli altri metodi sono molto semplici, sono stati completamente espressi e risultano essere autoesplicativi.

L'implementazione di `getDestinationStream()` include tutto ciò che è necessario mettere in atto per realizzare il compito più semplice: compilare il codice sorgente `.tex` presente come *bitstream* nella scheda di archiviazione.

L'implementazione è la seguente:

```
1  public InputStream getDestinationStream(
       InputStream source) throws Exception {
2
3      // Crea il file tex da compilare
4      File tmpfile = createTempLaTeXFile(
           source);
5      String inputFilename = tmpfile.toPath
           ().toString();
6
7      // Esegui PDFlatex sul file .tex
8      String cmd = "pdflatex␣-output-format=
           pdf␣" + inputFilename ;
9      try {
10       String line;
11       Runtime rt = Runtime.getRuntime();
12       Process pr = rt.exec(cmd);
13     }
14     catch (Exception e) {
15       System.out.println("Exception␣in␣
             latex␣compilation");
16     }
17
18     // Raccogli il risultato
19     String output = inputFilename.
           replaceFirst(".tex\$",".pdf");
20     File fhi = new File(output);
21     byte[] textBytes = Files.readAllBytes(
           fhi.toPath());
22     ByteArrayInputStream destination = new
           ByteArrayInputStream(textBytes);
23
24     return destination;
25 }
```

Il metodo è logicamente diviso in tre parti:

1. Estrazione del file latex e salvataggio su un file temporaneo (linee 4 e 5);
2. Compilazione LaTeX (linee dalla 8 alla 16);
3. Inserimento del file risultato PDF e nel bitstream di output (linee dalla 19 alla 23).

L'unica funzione esterna usata da questo pezzo di codice, che non è presente nelle API del linguaggio Java, è `createTempLaTeXFile()`, la cui implementazione è riportata in appendice all'articolo.

Il resto del codice sorgente è molto semplice: una volta ottenuto il nome del file temporaneo creato dalla funzione `createTempLaTeXFile` nella variabile `inputFilename` lo si usa per creare la linea di comando `cmd` che è messa in esecuzione attraverso l'oggetto `Runtime` di Java (Oracle) all'interno di una sezione `try...catch`. In caso di errore del `Runtime`, questo programma prototipale si limita semplicisticamente a stampare un messaggio sullo schermo ed uscire senza creare il file.

Se invece l'operazione va a buon fine, nell'ultima parte del metodo, una volta ottenuto il nome del file di output sostituendo all'estensione `.tex` quella `.pdf`, si legge l'intero contenuto in un array di bytes che, depositati in un `ByteArrayInputStream`, sono restituiti dalla funzione.

Per concludere l'operazione è necessario compiere un *commit* nel contesto di esecuzione dell'applicazione. Questo viene eseguito nel metodo `postProcessBitstream` che ha accesso all'oggetto `Context` (Diggory (2014)):

```
public void postProcessBitstream(Context
    c, Item item, Bitstream
    generatedBitstream) throws Exception
    {
  c.commit()
}
```

Pur nella sua semplicità il programma risolve completamente il problema posto e permette la realizzazione del file PDF a partire da un file LATEX contenuto nell'archivio. Le istruzioni necessarie alla compilazione e all'installazione del *mediafilter* sono riportate nella seconda appendice al programma.

A partire da questa base è possibile migliorare il programma in vari modi. Innanzitutto sarebbe necessario rendere più generico il programma attraverso l'uso di variabili configurabili invece che con l'utilizzo di stringhe fisse, eventualmente leggendo le variabili dal file di configurazione di `DSpace`. È possibile anche definire processi di compilazione più sofisticati che includano passi per l'inclusione delle bibliografie ad esempio o, sempre attraverso apposita configurazione, permettere l'uso di differenti processori oltre a `pdflatex`.

In parte queste alternative possono effettivamente essere espresse dal file di configurazione di `DSpace`, ma in parte forse maggiore dovrebbero adattarsi alla pubblicazione stessa connessa alla scheda. Quindi la configurazione del processo di compilazione o alcune caratteristiche dell'impaginazione potrebbero essere riportate in modo opportuno dalla scheda archivistica stessa, usando i campi appositi dello standard Dublin Core oppure adottando uno schema ad hoc proprio per questo.

È dunque pensabile integrare le informazioni della scheda archivistica (metadati):

1. per guidare la compilazione,
2. o all'interno della pubblicazione stessa.

4 Estrazione dei metadati

Per accedere ai metadati della scheda archivistica di un *elemento* è necessario interrogare l'oggetto `Item` con il metodo `getMetadata()`, i cui argomenti sono:

- nome dello schema (es. `dc`)
- nome dell'elemento (es. `contributor`)
- nome del qualificatore (es. `author`)
- la lingua dell'elemento (es. `it_IT`)

Ciascuno di questi argomenti può essere sostituito da `Item.ANY` o, limitatamente al qualificatore e alla lingua, dall'equivalente valore `null`.

Dall'esecuzione della funzione:

```
Metadatum[] values = item.getMetadata(
    schema, element, qualifer, lang);
```

si ottiene un array di metadati (classe `Metadatum[]`) che si possono usare all'interno del programma, sia per guidare la compilazione attraverso le opzioni sulla linea di comando, che per far parte del contenuto stesso (Diggory (2014)).

```
String schema = DCValue.schema
String element = DCValue.element
String qualifier = DCValue.qualifier
String lang = DCValue.language
String value = DCValue.value
```

Per comporre i valori ottenuti dai metadati all'interno del file LATEX sono possibili due approcci alternativi:

1. Scrivere un file aggiuntivo nella directory temporanea con le necessarie informazioni, e questo implica che nel file LATEX sorgente originale si sia introdotta un'istruzione di `\input` con il nome del file così creato (nell'esempio questo nome è `metadata.tex`).

2. Introdurre stringhe specifiche (ad esempio `$dc@contributor@author$`) direttamente all'interno del sorgente LATEX e poi usare una libreria Java di text-templating (come Apache Velocity o FreeMarker) o i servizi delle API Java standard (come `java.text.MessageFormat` o `StringTemplate` se si vuole fare a meno di librerie aggiuntive).

Un esempio della prima linea d'azione può essere implementato con tre funzioni: una che estrae tutti i metadati dalla scheda archivistica, la seconda che prepara il contenuto del file di supporto LATEX nel formato adeguato e la terza che scrive il file temporaneo.

```
public Map<String, List<String>>
    getMetadataHash(Item item) {
Map<String, List<String> > md = new
    HashMap<>();
Metadatum[] dcValues = item.getMetadata(
    Item.ANY, Item.ANY,
                    Item.ANY, Item.ANY);
Set<String> schemas = new HashSet<String
    >();
for (Metadatum dcValue : dcValues) {
        schemas.add(dcValue.schema);
    }
for (String schema : schemas) {
    Metadatum[] dcorevalues = item.
        getMetadata(schema,
                        Item.ANY, Item.ANY
                            , Item.ANY);
    String dateIssued = null;
    String dateAccessioned = null;
    for (Metadatum dcv : dcorevalues) {
    String qualifier = dcv.qualifier;
    if (qualifier == null) {
        qualifier = "";
    } else {
        qualifier = "@" + qualifier;
```

```
    }
    String key = schema + "@" + dcv.
        element + qualifier;
    List<String> vlist = new ArrayList<
        String>();
    if ( md.containsKey(key) ) {
        vlist = md.get(key);
    }
    vlist.add(dcv.value);
    md.put(key, vlist);
    }
}
return md;
}
```

In questa prima funzione il nome delle variabili viene costruito con il carattere `@` come separatore, ovvero così:

```
\newcommand{
  \schema@elemento@qualificatore@X
  }{ <valore> }
```

Poiché per ogni nome di metadato in `DSpace` sono sempre possibili più valori si è introdotto anche un elemento finale, indicato come `X` nel precedente esempio, ovvero una lettera (ad iniziare da `A`) che si incrementa per ogni valore. Quindi, in effetti, il primo valore della chiave `dc.contributing.author` sarà utilizzabile effettivamente nel sorgente LaTeX come `\dc@contributing@author@A`.

La funzione `getMetadataHash` restituisce una lista di chiavi-valori (possibilmente multipli) che la successiva funzione `getMetadataAsString` trasforma in un blocco di testo:

```
public String getMetadataAsString(Map<
    String, List<String>> md) {
Iterator<String> keySetIterator = md.
    keySet().iterator();
List<String> lines = new ArrayList<String
    >();
while(keySetIterator.hasNext()) {
        String key = keySetIterator.next()
            ;
        List<String> values = md.get(key);
        int index = 0;
        for (String value : values) {
          String letter = (char) (65+index
              ++);
          lines.add("\\newcommand{" + key
              + "@" + letter + "}{");
          lines.add(latex_escape(value)+"}
              ");
        }
}
String block = "% LATEX BLOCK\n\\
    makeatletter\n";
for(char line: lines) {
    block = block + line + "\n";
}
block = block + "\\makeatother\n% END
    LATEX BLOCK\n\n\n";
return block;
}
```

In questa implementazione prototipale non si è tenuto conto che le stringhe contenute nei metadati potrebbero contenere caratteri non permessi all'interno del sorgente LaTeX. Per realizzare un programma effettivamente utilizzabile bisognerebbe prevedere la trasformazione di questi caratteri non accettabile in combinazioni di caratteri accettabili da LaTeX.

La funzione che scrive il file LaTeX nella directory temporanea è:

```
private void createTempFile(String dir,
    Item item) throws IOException {
    Map<String, List<String>> md =
        getMetadataHash(item);
    String metadataStr =
        getMetadataAsString(md);
    FileUtils.writeStringToFile(new File(
        dir + "metadata.tex"), metadataStr
        );
}
```

All'interno del file sorgente LaTeX iniziale è quindi necessario introdurre nel preambolo l'istruzione:

```
\input{metadata}
```

e poi utilizzare nel testo i comandi creati dalle variabili.

L'oggetto `Item` non è presente tra i parametri del metodo `getDestinationStream()`, quindi tutte le operazioni necessarie all'acquisizione dei metadati devono essere compiute nell'implementazione di un ulteriore metodo denominato `preProcessBitstream()` che, come dice il nome, viene eseguito prima della funzione di elaborazione del bitstream (cioè `getDestinationStream()`)

```
    public boolean preProcessBitstream(
        Context c, Item item, Bitstream
        source) throws Exception {
      // Metodo eseguito prima di
          realizzare la conversione, puo'
          servire ad estrarre i metadati
          utili
      tempPath = Files.
          createTempDirectory("latex");
      createTempFile(tempPath.toString(),
          item);
    }
```

Allo stesso modo, qualora sia necessario modificare il valore dei metadati o scriverne di nuovi (ad esempio per salvare informazioni sull'andamento della compilazione), è necessario interagire con l'oggetto `Context` di `DSpace` che non è presente nel metodo `getDestinationStrem()` (Diggory (2014)). Anche in questo caso può usarsi il metodo `preProcessBitstream` o, forse più adeguato, il metodo `postProcessBitstream` che viene messo in esecuzione solo se l'operazione di trasformazione è andata a buon fine.

5 Conclusioni

Pur nella sua estrema sintesi e semplificazione, quest'articolo presenta un esempio completo di realizzazione di un *mediafilter* per `DSpace`. Il programma è utile ad organizzare collezioni di documenti presenti nel repository direttamente in formato sorgente, lasciando a `DSpace` il compito di creare i file PDF come sottoprodotto.

Le possibilità di estensione di quest'esempio sono numerose e solo per citarne alcune è possibile considerare i seguenti scenari:

1. Spesso una pubblicazione è composta da più di un file `.tex`, quindi potrebbe essere necessario caricare nell'archivio differenti file sorgente. In questo caso le strategie possibili possono essere due.
   (a) Usare un unico file compresso (come un file `.tar` o `.zip`) contenente tutti i sorgenti necessari da scompattare prima della compilazione.
   (b) Caricare singolarmente tutti i file necessari nella scheda archivistica insieme ad un *file di produzione* che racchiuda i nomi dei bitstream da coinvolgere nella produzione del prodotto. Questo file di produzione potrebbe avere un'estensione ad hoc (ad esempio `task`) che permetterà di mettere in esecuzione il *mediafilter*.
2. Le pubblicazioni di natura scientifica o statistica potrebbero prevedere un aggiornamento automatico dei dati sottostanti. Per realizzare un tale automatismo si può registrare un processo periodico di cancellazione dei file PDF prodotti e adottare all'interno del file LATEX sistemi di acquisizione dei dati dalle fonti utilizzando estensioni del linguaggio LATEX come `pythontex` (Poore, 2015) o considerando l'introduzione di elaborazioni nei linguaggi di programmazione scientifica come passi propedeutici alla compilazione dei documenti.
3. Piuttosto che adottare LATEX come file sorgente è sempre possibile usare un formato più astratto, dal più semplice `Markdown` (Ovadia (2014)) al più sofisticato `Docbook` (Martínez-Ortiz *et al.* (2006)), che possa generare differenti tipi di file prodotto (PDF, HTML, ePub, ecc.). In questo caso si potrà utilizzare un processore generico come `pandoc` (Dominici (2014)) o `expand` (Krijnen *et al.* (2014)) per compiere il lavoro di compilazione.

Queste e altre simili estensioni possono far diventare un'istanza `DSpace` non solo un repository statico di documenti, ma un processore di flussi di lavoro il cui risultato, pronto per essere consultato dall'utente finale, viene conservato all'interno delle schede archivistiche.

Appendice 1: La funzione `createTempLaTeXFile()`

Per completezza espositiva si riporta in quest'appendice il codice sorgente del metodo `createTempLaTeXFile()` che peraltro non presenta caratteri di particolare complessità.

```
1  private File createTempLaTeXFile(
       InputStream source) throws IOException
        {
2      File f = File.createTempFile("document
           ", ".tex");
3      f.deleteOnExit();
4      FileOutputStream fos = new
           FileOutputStream(f);
5      byte[] buffer = new byte[1024];
6      int len = source.read(buffer);
7      while (len != -1) {
8          fos.write(buffer, 0, len);
9          len = source.read(buffer);
10     }
11     fos.close();
12     return f;
13 }
```

L'unica annotazione da fare su quest'implementazione è che nel caso del sorgente riportato a pag. 4 sarà necessario aggiungere alla creazione del file anche il parametro relativo alla directory temporanea in cui viene posto il file dei metadati. Quindi la riga 2 diventerebbe:

```
1      File f = File.createTempFile("document
           ", ".tex",tempPath);
```

Il codice sorgente dei programmi riportati nell'articolo è disponibile all'indirizzo `https://github.com/exedre/2018-arstexnica-latex-dspace`.

Appendice 2: Compilazione del *mediafilter*

Per integrare il *mediafilter* realizzato all'interno di una installazione di `DSpace` è necessario procedere alla sua compilazione all'interno del codice sorgente di `DSpace`.

Il file Java contenente la classe del nuovo *mediafilter* va posizionata all'interno della directory

```
1 cp mediafilter.java [dspace-src]/dspace-
      api/src/main/java/org/dspace/app/
      mediafilter
```

per poi eseguire il comando di compilazione:

```
1 cd [dspace-src] && mvn package
```

Al termine della compilazione si potrà installare il nuovo filtro con il comando:

```
1 cd [dspace-src]/dspace/target/dspace-
      installer/ && ant update
```

Una volta installato il nuovo *mediafilter* sarà necessario riportarlo nella configurazione aggiungendo un nome nella chiave di configurazione `filter.plugin`:

```
1 filter.plugins = <Altri nomi di filtri>,
      Nuovo Mediafilter
```

e aggiungendo la classe del filtro nell'elenco dei filtri di formato:

```
1 plugin.named.org.dspace.app.mediafilter.
      FormatFilter = \
2   org.dspace.app.mediafilter.
        nuovoMediafilter = Metadata Exporter
        , \
3   [...]
```

Infine è necessario indicare il tipo o i tipi dei file di input che dovranno mettere in esecuzione il filtro indicato:

```
filter.org.dspace.app.mediafilter.
    nuovoMediafilter.inputFormats =
    Markdown
```

Riferimenti bibliografici

De Robbio, A. (2002). «Relazione tecnica su dspace (mit)».

Diggory, M. (2014). «Business logic layer». URL https://wiki.duraspace.org/display/DSDOC5x/Business+Logic+Layer.

Dominici, M. (2014). «An overview of pandoc». *TUGboat*, **35** (1), pp. 44–50.

Donohue, T. (2016). «Mediafilters for transforming dspace content». URL https://wiki.duraspace.org/display/DSDOC5x/Mediafilters+for+Transforming+DSpace+Content.

Krijnen, J., Swierstra, D. e Viera, M. O. (2014). «Expand: Towards an extensible pandoc system». In *International Symposium on Practical Aspects of Declarative Languages*. Springer, pp. 200–215.

Martínez-Ortiz, I., Moreno-Ger, P., Sierra, J. L. e Fernández-Manjón, B. (2006). «Using docbook and xml technologies to create adaptive learning content in technical domains.» *IJCSA*, **3** (2), pp. 91–108.

Oracle. «Class runtime». URL https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html.

Ovadia, S. (2014). «Markdown for librarians and academics». *Behavioral & Social Sciences Librarian*, **33** (2), pp. 120–124.

Poore, G. M. (2015). «Pythontex: reproducible documents with latex, python, and more». *Computational Science & Discovery*, **8** (1), p. 014010.

▷ Emmanuele Somma
Biblioteca Paolo Baffi
Servizio Struttura Economica
Dipartimento Economia e Statistica
Banca d'Italia
emmanuele dot somma at bancaditalia dot it