HOCHSCHULE TRIER
Trier University of Applied Sciences
Informatik - Computer Science

Pugo: Convert Google Docs using XSLT

Tobias Klevenz

Projektarbeit

Betreuer: Prof. Dr.-Ing. Georg J. Schneider

Almere NL, 30.6.2016

Abstract

This project aims to provide a way to convert documents created with Google Docs to any text based format. Google Docs has a limited number of integrated export formats, which can be extended by providing an interface that utilizes XSLT, which allows the creation of XML, HTML or other text based formats. To provide a suitable source for the XSL transformation, Google Docs' HTML export is converted to a wellformed XHTML document which then can be transformed to the new export format using XSLT. Two example implementations of the application were created, from which one is already actively used to transform Google Docs documents into a markdown inspired format for a Content Management System.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Introduction

1.1 Motivation

Cloud Productivity Suites are no longer the exception, Software as a Service solutions like Office 365 and Google Apps are used by millions of businesses and individuals worldwide. Being constantly connected to the internet has led to a shift from editing documents offline in dedicated office application to editing documents online in a browser. Although Cloud Productivity Suites still lack a lot of features compared to their offline counterparts, they shine through the features that come naturally by being online, like not having to worry about backing up files or working collaboratively on documents. Under the name Google Apps, Google has published a growing number of applications that all live in the cloud and can be accessed from any browser or device. Next to the communication specific tools like Google Mail and Google Calendar, Google Apps also includes a set of office tools to edit documents, spreadsheets, forms and sheets. Documents are created and edited using Google Docs.

After registering with a Google Account, users can create documents with Google Docs which are then automatically stored on Google’s servers. Users can share access to a created document with other collaborators, which depending on the access level can view, comment or edit the document. Collaborators do not necessarily need to have a Google Account, however all edits and comments made by people not logged into a Google Account will be made by “Anonymous” and won’t be distinguishable from other anonymous collaborators.

Google Docs creates and stores documents in a flat HTML inspired data format, without any semantic markup. It supports a fixed set of file types to which a document can be exported however all of those lack the same semantic markup. Existing solutions extending Google Docs to additional file formats are all single purpose solutions that export a document to one specific file type. To provide a general solution that will extend Google Docs to a much larger set of file formats I created Pugo, an application that uses XSLT to convert a document created with Google Docs to a much larger number of file types. This paper describes how the application was implemented, and how it can be used in a real world example. The example implementations provided demonstrate how the application can be configured to export documents to the EPUB format that can be read with ebook readers, as well as a markdown inspired format used by the Content Management System Kirby.

1.2 Goals

This project wants to extend Google Docs to support a wide range of formats, so it can be used as a multi purpose solution instead of simply extending Google Docs to support a single additional format.

As Google Docs itself does not have any semantic markup at all, it should be possible to define a set of rules using XSLT that will create a semantic structure from the flat html data structure that Google Docs provides.

Furthermore it should be determined if and how Google Docs can be used as a content creation platform beyond its designed purpose, as well as if it is feasible to use Google Docs as a backend for a content management system.

2 Related Work

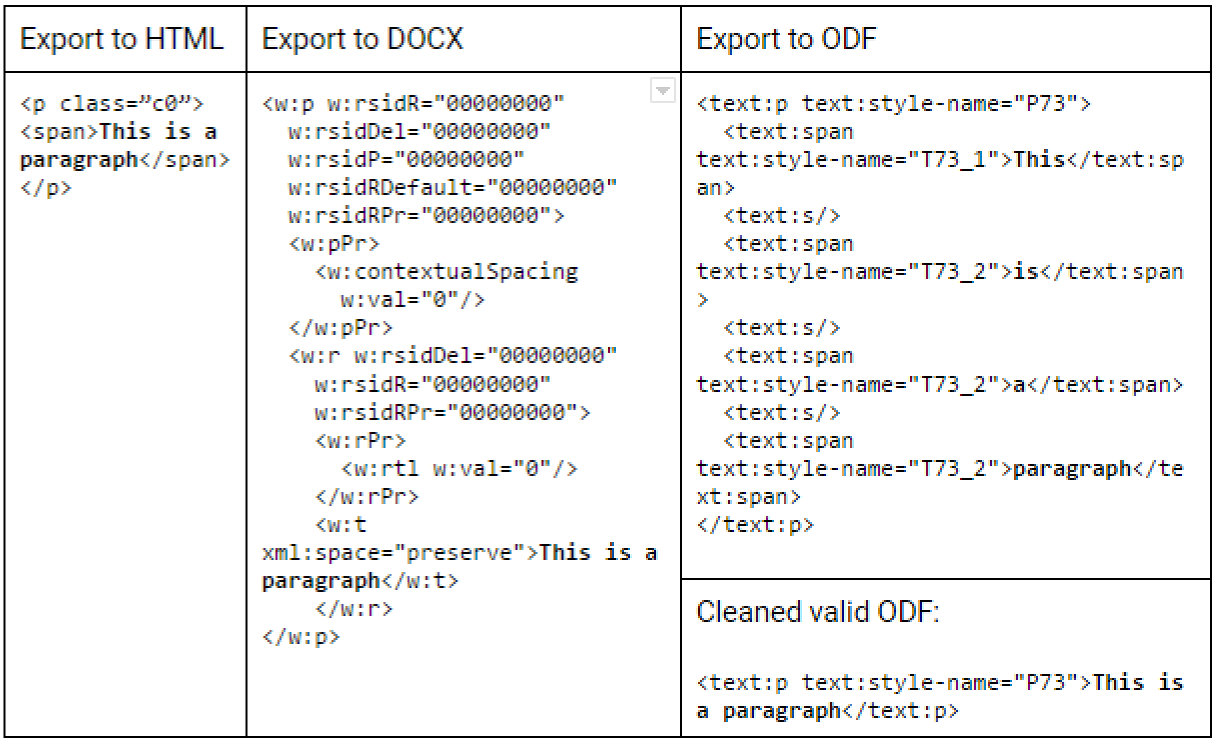
This section takes a look at Google Docs' own supported export formats as well as other more specific solutions that exist or have existed for the purpose of extending Google Docs to support additional file formats.

2.1 Google Docs integrated export formats

As of the writing of this document, Google Docs supports seven export formats, which are Microsoft's **Open Office XML (docx)**, Open Office's **Open Document Format (odt)**, **Rich Text Format (rtf)**, **Plain Text**, **PDF**, **HTML** and **EPUB**. Plain Text and PDF can be ignored for the purpose of this paper, the former will simply dump all text of a document without any markup into a text file and the latter will provide a "digitally" printed version of the document that can not be edited any further. Google Docs provides an API to export documents to those formats. EPUB, its latest addition to the formats, however is not currently supported to be exported using the API.

Through the office based formats docx and odt, Google Docs implements compatibility with its main competition, which are offline Office Suites. Both formats provide a decent enough document that can be edited with Microsoft Word, Open Office or other Office Solutions that support those formats, however the complexity of those formats makes them pretty much useless outside of those applications. Both formats are a set of zipped XML and binary files separating document content, images, style and meta information. The XML structure of the content documents is very verbose as it splits up paragraphs into text blocks with redundant style information. In the case of Open Document Format, a paragraph in the exported file will be split up into span elements for each word as you can see in **Figure 1**. This will result in an XML document that is barely human-readable, which was always one of the main goals of XML. The exact same thing happens with the rtf export, another format that aimed to be human-readable but fails to be so in this case.

Using the HTML export option, a user can download the document as a zip archive, containing the text content in a single HTML file and all images used in the document in a separate folder. The HTML uses CSS classes that are referenced in the p and span elements and defined in an internal stylesheet. Emphasized text does not use the corresponding html tags like strong or em, instead it will set its font-style or -weight via a CSS class applied to the span element. When using the API, Google Docs also has the option to export to a single HTML file that links to images stored on Google's Servers.



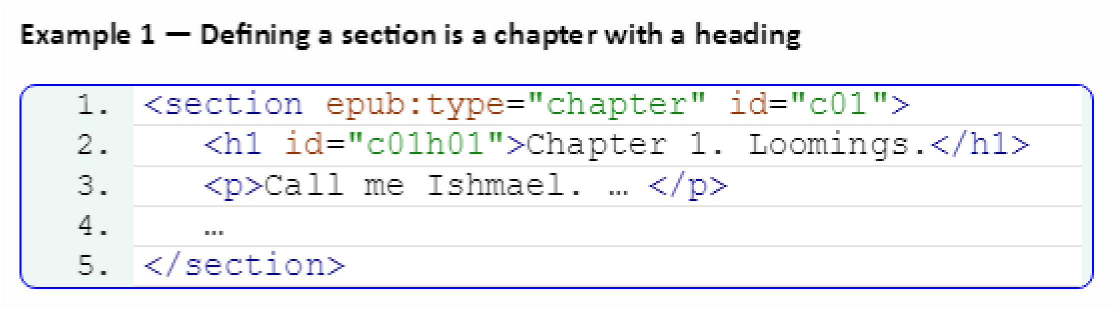
Export to HTML	Export to DOCX	Export to ODF

`<p class="c0"><span>This is a paragraph</span></p>`	`<w:p w:rsidR="00000000" w:rsidDel="00000000" w:rsidP="00000000" w:rsidRDefault="00000000" w:rsidRPr="00000000">`
	`<w:pPr>`
	`<w:contextualSpacing w:val="0"/>`
	`</w:pPr>`
	`<w:r w:rsidDel="00000000" w:rsidR="00000000" w:rsidRPr="00000000">`
	`<w:rPr>`
	`<w:rtl w:val="0"/>`
	`</w:rPr>`
	`<w:t xml:space="preserve">This is a paragraph</w:t>`
	`</w:r>`
	`</w:p>`
		`<text:p text:style-name="P73">`
		`<text:span text:style-name="T73_1">This</text:span>`
		`<text:s/>`
		`<text:span text:style-name="T73_2">is</text:span>`
		`<text:s/>`
		`<text:span text:style-name="T73_2">a</text:span>`
		`<text:s/>`
		`<text:span text:style-name="T73_2">paragraph</text:span>`
		`</text:p>`

		Cleaned valid ODF: `<text:p text:style-name="P73">This is a paragraph</text:p>`

Figure 1: Comparing office export formats. The exported office formats produce complicated large chunks of XML for a simple paragraph. The ODF is especially badly exported, ODF does not require every single word to be in a span element, as shown by the cleaned ODF example.

With EPUB docs can export a document as an eBook, a format that can be consumed by eBook readers. The exported document is a valid EPUB 3.0 document, which is the latest version of the format. EPUB uses XHTML to markup its content. As shown in **Figure 2**, EPUB allows for the addition of semantics using section elements with an attribute epub:type, that can be used to define all common sections found in publications, like chapter, preface or appendix. An EPUB document exported from Google Docs however will not containing any semantic markup at all, instead of the EPUB's table of content linking to individual chapters, it will merely link to all the headlines defined in the document. In the examples section this paper will demonstrate how using a set of XSLT rules it will be possible to create an EPUB document containing semantic markup.



Example 1 — Defining a section is a chapter with a heading

```
<section epub:type="chapter" id="c01">
    <h1 id="c01h01">Chapter 1. Loomings.</h1>
    <p>Call me Ishmael. … </p>
    …
</section>
```

Figure 2: Defining a chapter using a section element in EPUB 3.0 as described in [EPUBGuide]

2.2 Liberio

Liberio was a proprietary service that allowed users to convert documents stored in cloud storage solutions like Google Drive, Dropbox or Microsoft's OneDrive to EPUB. The service shutdown in November 2015. The team behind Liberio announced that they would open source the technology behind their service but as of the writing of this document did not do so. The service only allowed to export to a single format and did not offer extensive configuration options for this format.

2.3 gdocs2md

gdocs2md is a script written in Google Apps Script that will convert a document created with Google Docs to a markdown file. When run on a document created by Google Docs the script will create a markdown file and send it to the email address of the Google account of the user executing the script. All images contained in the source document will be attached to the email sent to the user. The script is also available packaged as a Google Docs Add-On which can be installed and run from within Google Docs.

3 Materials and methods

This section provides an overview over all tools that were involved in creating this project and explains why each specific tool was chosen.

3.1 Google Docs

Google Docs is available as part of Google Drive for no cost to anybody who has a google account, as well as part of Google Apps for Education and the paid subscription service Google Apps for Work. Its significant user base as well as its extensive well documented APIs were the primary reason behind choosing it for the project. As the project is build around open technologies and uses a simple html exported document as a source, extending the application to other similar cloud office solution like Microsoft's Office 365 is always an option but for the purpose of this paper not further considered.

3.2 XSLT

XSLT is a language designed to transform XML documents into other XML or text based formats. It takes an input document which has to be well formed XML and applies so called templates to the nodes of the document that define rules on how to transform those nodes in the resulting output. The source document will be treated as "read only" and one or more new documents will be created.

XSLT is currently available in three versions:

XSLT 1.0 was finished in 1999 but is still widely used on the web today, because it is the only version that is natively supported by most browsers.

XSLT 2.0 was released in 2007 and added a lot of features that made XSLT a more powerful tool to create more complicated structures. The addition of `for-each-group` for example, which is basically a loop that can process results in groups, very similar to SQL's GROUP BY statement, can be very helpful to build a hierarchical structure from a flat source such as a html document. Another addition is `result-document` which allows the creation of multiple files from one transformation. Both of those are relevant when looking at the example transformation to EPUB later on.

XSLT 3.0 is currently in a draft status, the changes it brings compared to 2.0 are mainly geared towards streaming large XML documents, a process where the transformation will not parse the complete DOM tree of the source document before the transformation and rather will process the document element by element.

All examples shown in this paper will use XSLT 2.0, the additional features compared to 1.0 are essential to the process. However the implementation of additional XSLT scripts does not have to be restricted to 2.0, both XSLT 1.0 and XSLT 3.0 are supported by the processor used in the project.

3.3 Java

The main part of the project was implemented in Java and can be compiled with Java 7 and up. Java as described in its Specification [JavaSpec p.1] is a *“general-purpose, concurrent, class-based, object-oriented programming language”*. Java offered many advantages for the implementation, as it has the biggest selection of available XSLT processors, a wide range of build tools and dependency management solutions like Maven, as well as wide ranging options when it comes to deploy the application in the cloud or on a hosted server.

3.3.1 Apache Maven

Apache Maven[1] is a build automation tool that can be configured using a so called Project Object Model (POM) that is stored using a single xml file named pom.xml. The POM contains settings for building the application, like needed dependencies or environment settings as well as general project information like name, url and developers involved in the project. Maven will run, test, compile, package and resolve the dependencies of the project. Needed libraries can be added as dependencies to the POM file and by default will be downloaded from the central repository[2], if needed additional repositories can be added. For the project the following libraries have been used:

Saxon

Saxon is an XSLT and XQuery processor developed in Java by Michael Kay, who was in a large part responsible for the creation of XSLT 2.0. It is in active development and is maintained by his company Saxonica[3].

Saxon is currently available in a free and open source version, the so called Home Edition as well as a Professional and Enterprise Edition. The Home Edition offers the complete feature set of XSLT 2.0 as well as XSLT 1.0 and is therefore sufficient for the purpose of this project.

JTidy

JTidy[4] is used to transform the from Google Docs exported HTML to well formed **X**HTML that can be processed using XSLT. JTidy is a Java implementation of the popular HtmlTidy application that is recommended by the W3C.

Apache Commons

Apache Commons[5] is a project that provides reusable Java components, that are designed to simplify common programming solutions. For this project IOUtils which is part of the Commons IO collection[6] is used to simplify the handling of Input- and OutputStreams. The encoder to Base64 which is available through Commons Codec[7] is used to encode images in the document.

[1] **https://maven.apache.org/**
[2] **http://repo.maven.apache.org/maven2/**
[3] **http://www.saxonica.com/**
[4] **http://jtidy.sourceforge.net/**
[5] **https://commons.apache.org/**
[6] **https://commons.apache.org/proper/commons-io/**
[7] **https://commons.apache.org/proper/commons-codec/**

Apache Tika

The Apache Tika[8] toolkit can extract metadata from different file types, in the project it is used to detect the Mimetype of the Images in the document, this was necessary because Google Docs will serve the images used in a document by URLs without file extension.

Apache Tomcat

For the development of the project Tomcat was used as Application Server. Strictly speaking this is not a dependency used in the POM but a plugin in the build settings. To run tomcat from in the development environment, the command `mvn tomcat7:run` can be executed which will then launch a Tomcat instance on port 8080 of localhost, which is very convenient for quick tests while developing.

The only Tomcat plugin available for maven is Tomcat 7, so that is the version that has been used for development however Tomcat 8 has been tested and works as well.

[8] https://tika.apache.org/

4 Application Design

4.1 Java Servlet

The main part of the application is implemented as a headless Java Servlet. As described in its documentation [JavaTutorial], a *“servlet is is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model”*, which is an ideal choice for the implementation of this project. A request is sent to a server, the server processes the data received with the request and sends back a response. An alternative to this would be to let the client process the data, which in the case of this project would be a browser, however the tools needed for processing the document inside a browser are severely limited due to the lack of XSLT and Java support in modern browsers. Describing the servlet as headless means that no frontend is needed to run the main application, the Servlet only has to handle HTTP Requests and process the received data. In handling those requests, the Servlet should support both GET and POST, so it is up to the frontend to decide how to connect to the Servlet, for security reasons one might prefer a POST where no sensitive data is sent in clear text with the URL of the request, but that should not be decided by the Servlet itself.

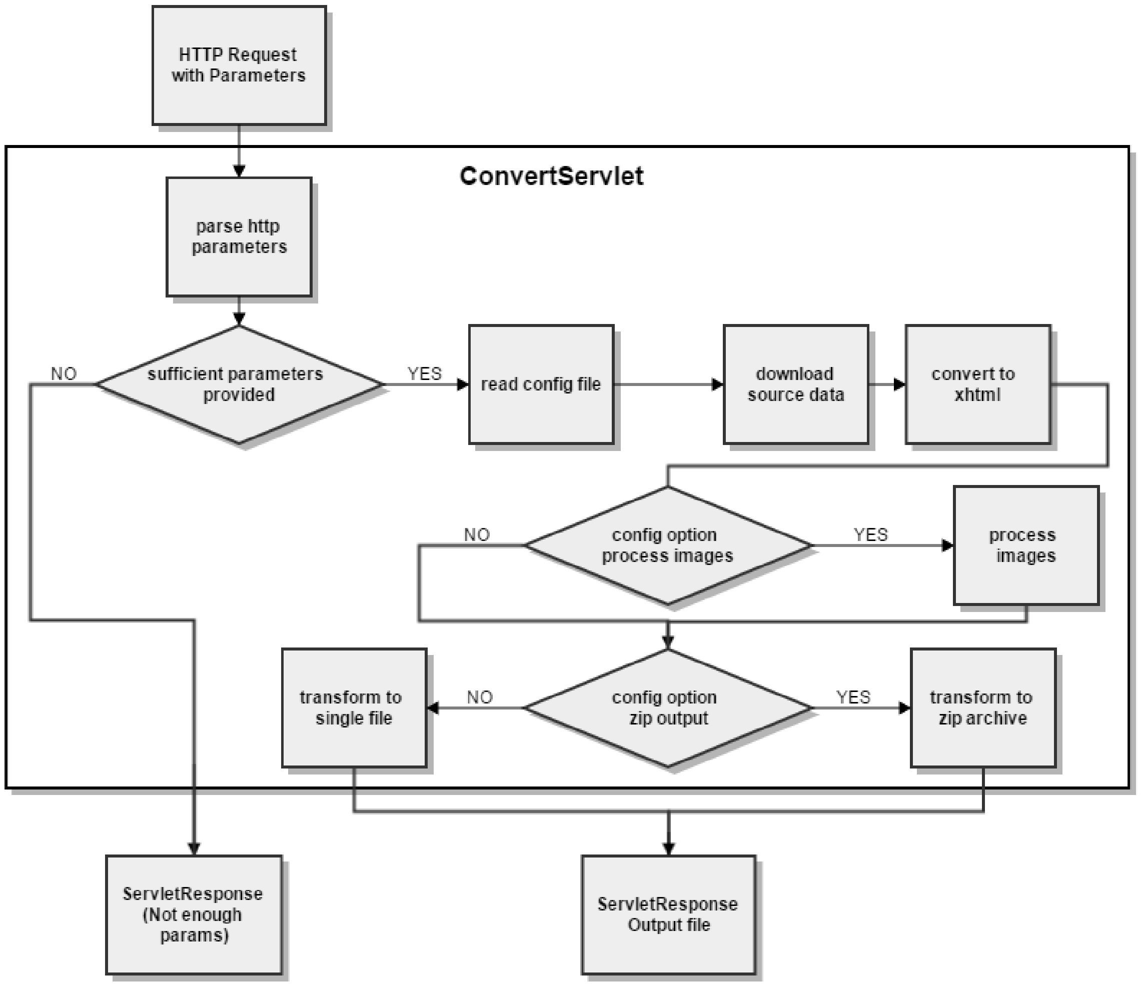


Figure 3: Flowchart outlining the application design.

As outlined in **Figure 3** an HTTP request with a set of parameters has to be sent to the servlet to start the process. After parsing the required parameters and reading the configuration file, the source document has to be downloaded and processed. Processing will include converting the document to XHTML, optionally download and process images included in the document and finally transform the document to either a single file or a zip archive containing multiple files.

4.2 XSL Transformation Scenarios

For the implementation of the XSL Transformation there are two different scenarios that have to be considered:

1. The transformation will transform the source into a single output file which is defined with the declaration of `xsl:output`
2. The transformation will transform the source into multiple documents which are defined using `xsl:result-document`

Scenario 1 is the standard implementation of an XSL transformation, while scenario 2 has to be implemented using a so called OutputURIResolver, which is a class that defines how the documents generated through `xsl:result-document` are handled. In the case of this project they should be written to a single Zip archive that can be send as a ServletResponse.

There is a possible third scenario that could be created using XSLT, if the output defined through `xsl:output` as well as the outputs defined through `xsl:result-document` are produced, however for the project this is not immediately relevant so a choice between either scenario 1 or 2 has to be made.

4.3 Configuration and Extensibility

To accomplish its task the application has to provide a way to configure different transformations to output formats without editing any Java code. To achieve this the application reads its configuration from one or more XML files that define the options needed for a transformation. An implementation of the application could either replace the default configuration or provide additional ones. Additional configurations should be selectable with a parameter that is sent with the request.

4.4 Frontend

The frontend that accesses the application should be independent and interchangeable as well as not directly tied to the Servlet. As a demonstration two examples will be provided, one is implemented as a website using standard HTML and JavaScript, the other is implemented as Google Apps Script which can be directly used as an add-on in Google docs. In general however any application that can make an HTTP Request should be able to serve as a client.

5 Implementation

This section describes the implementation of the application, which is primarily made up of a Java Servlet called `ConvertServlet`. As shown in the class diagram in **Figure 4**, additional classes have been created for the project. The classes `Parameters` and `Configuration` are used to store the parameters received from the http request as well as all information needed for the current configuration of the XSL transformation. The class `Transformation` is a wrapper class that will setup and run the transformation. If configured according to the second transformation scenario that is outlined in Section 4.2, the `ZipOutputURIResolver` is used to define how the documents created with `xsl:result-document` are written to the zip file. All objects of the defined classes are referenced and created from within the service() method of the servlet, encapsulating all relevant data for each request sent.

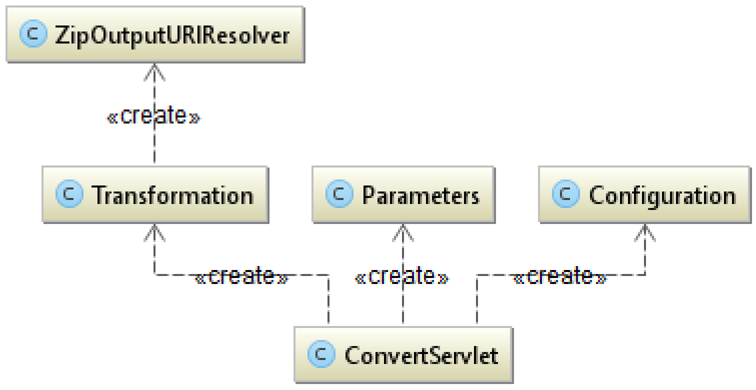


Figure 4: Class diagram of the package co.pugo.convert.

`ConvertServlet` is a generic Servlet implementation that extends `HttpServlet` and will be mapped to the address `http://servername/pugo-co/convert` in its default configuration. Requests sent to the servlet have to be sent to the URL including the necessary HTTP parameters. To perform tasks needed to answer an http request, the `HttpServlet` class provides the methods `doGet()` and `doPost()` to handle either a GET or POST request separately. Additionally the method `service()` can be used to handles both GET and POST requests. By default `service()` will dispatch the requests to the corresponding doXXX method, if overridden it will be used for both type of requests. As the application does not want to distinguish between GET and POST, the `service()` method is overridden and builds the centerpiece of the application from where all code is executed. This is a recommended pattern according to the Java documentation [JavaDevGuide].

5.1 Preparation

This section describes all the actions that have to be completed before the source document can be accessed and processed.

5.1.1 Parsing Parameters

The first thing the servlet will do upon receiving a request will be to create an object of the class Parameters by handing a map of all provided HTTP parameters to its constructor. The parameters are retrieved by calling `getParameterMap()` on the `HttpServletRequest` object that is given as a parameter to the `service()` method when the Servlet is accessed.

The applications minimum required parameters are **`source`** and **`fname`**, `source` has to be a link to the source document provided as a URL, `fname` is the name that should be given to the final output file that is returned by the servlet.

Optional parameters that can be provided are **`token`**, **`mode`** and any number of parameters in the form **`xslParam_<XSLT Parameter Name>`**.

Google's APIs use the OAuth 2.0 protocol for authorization and authentication. A client that wants to access a document from Google Docs via its API must obtain a so called access token before access to the document is given. The servlet can receive an optional parameter named **`token`** containing such an access token. This parameter is optional so a client sending the request could provide a direct link to the document after having the authentication already resolved. Additionally this potentially allows the servlet to receive a link to a document that hasn't been created with Google Docs at all, which opens the possibility of using the servlet for other services.

The parameter **`mode`** is used to select configuration files that are are not the default configuration, the project provides two examples, one for EPUB and one for Markdown.

The parameters of the form **`xslParam_<XSLT Parameter Name>`** are used to pass parameters to the XSL transformation. For example a parameter that is defined in the XSLT stylesheet as `<xsl:param name="`**`pub-language`**`"/>` could be provided as an http parameter named `xslParam_pub-language`.

After successful creation of the Parameters object the servlet will check if all required Parameters are given by calling the helper method `hasRequiredParameters(),` which will send usage instructions to the servlet response if not enough parameters are provided.

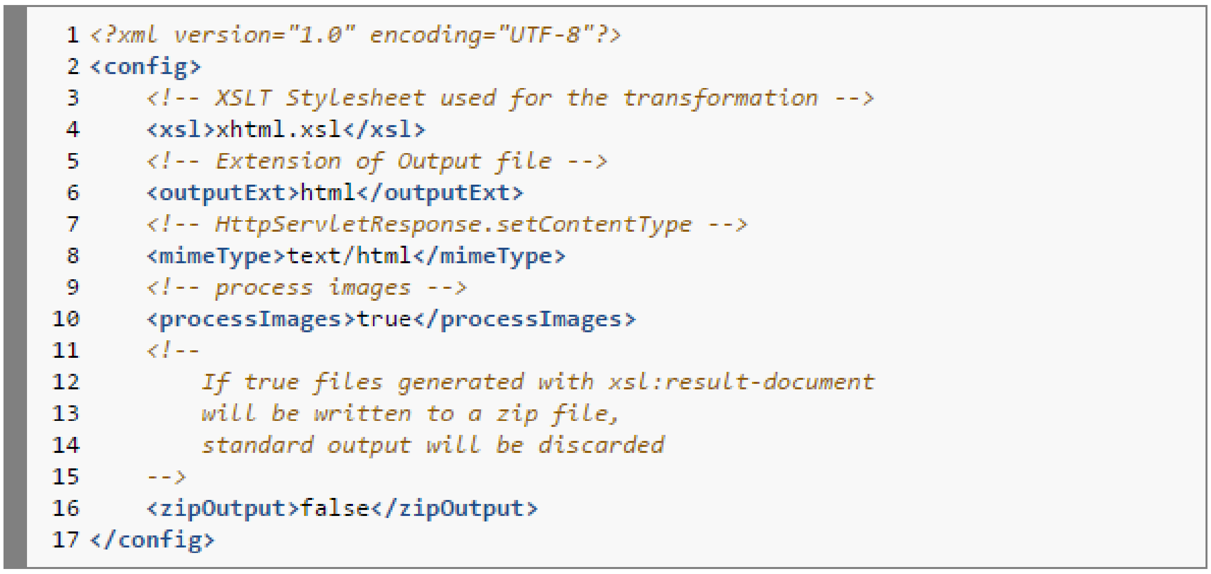
5.1.2 Reading Configuration file

To configure one or more transformation scenarios, the application makes use of XML configuration files. The configuration files are stored in **WEB-INF/CONFIG**, this directory is not part of the public document tree of the application [ServletSpec p70], which means they can not be accessed from outside the servlet context and this is an ideal choice to store a set of configuration files. The application has a default configuration stored in config.xml as well as optionally more additional files named in the form **config_<mode>.xml**. Each mode defines a different transformation scenario. To trigger the use of one of those scenarios the mode parameter has to be supplied with the request sent to the servlet. As examples the project ships with three example configuration files.

config.xml is the default configuration which will be used when no mode parameter is provided. The transformation scenario it provides will provide an XHTML version of the converted document.

config_epub.xml configures a transformation scenario that will return an EPUB file. This configuration will make use of the zip archive functionality of the application.

config_md.xsl configures a transformation scenario that will return a markdown inspired format used by Kirby CMS[9], which is meant to be used together with the Google Docs Addon example.



```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <!-- XSLT Stylesheet used for the transformation -->
    <xsl>xhtml.xsl</xsl>
    <!-- Extension of Output file -->
    <outputExt>html</outputExt>
    <!-- HttpServletResponse.setContentType -->
    <mimeType>text/html</mimeType>
    <!-- process images -->
    <processImages>true</processImages>
    <!--
        If true files generated with xsl:result-document
        will be written to a zip file,
        standard output will be discarded
    -->
    <zipOutput>false</zipOutput>
</config>
```

Figure 5: Default configuration file. This is the default configuration file of the application which is used to configure the XHTML transformation scenario.

As shown in **Figure 5** the following options can be configured in the configuration files:

xsl: name of the stylesheet to use for the transformation, the stylesheets referenced here are stored in the same directory as the configuration file.

mimeType: defines the content type of the response that is sent by the servlet. This should represent the file type of the file created from the transformation.

processImages: flag to enable or disable processing of images, if set to true this will trigger the application to download images found in html img tags and replace the links of the src attributes with base64 encoded versions of the images.

zipOutput: if true all files generated by xsl:result-document will be written to a zip file, while the standard output is discarded.

ConvertServlet defines a helper method getConfigFile() that will try to open an InputStream of the configuration file specified by the mode parameter. If no mode parameter has been provided or a configuration file for the mode can not be found it will return the default configuration instead. The returned InputStream is passed to the constructor of the Configuration class, which will provide a single object holding all the needed configuration information extracted from the config file.

5.1.3 Response Properties

The response sent by the Servlet needs to have certain properties so the receiver knows how to handle the received content. The method `setResponseProperties()` will set the ContentType to the mimetype that was read from the configuration file and set the charset of the

[9] **https://getkirby.com/**

received content to UTF-8 so special characters are preserved. Furthermore the property `Content-Disposition` of the header is set to `attachment; filename=<Filename>` which in case the receiver is a browser will tell it to download the file instead of directly opening it.

5.1.4 Connect to the source document

To access the source document the method `getSourceUrlConnection()` will open and return a `URLConnection` object by calling `openConnection()` on the URL that has been provided with the source parameter. If an authorization token was provided the method will also set the request property Authorization of the `URLConnection` object so the document can be accessed.

5.2 Convert HTML to XHTML

As mentioned previously, the source document for an XSL transformation has to be a so called well-formed XML document, which means it adheres to certain syntactical rules that are outlined in the XML 1.0 specification. To achieve that without an actual transformation of the data format is to convert the HTML of the source document to XHTML, which basically is an HTML version that conforms to the XML syntax. Since writing a parser that does this conversion would go beyond the scope of this paper, a library called JTidy was used to do the conversion. JTidy is a Java port of the popular HTML TIDY tool which is recommended by the W3C for syntax checking HTML [Tidy]. Although Tidy was designed to fix mistakes made by editing HTML files, its ability to produce well formed XHTML as well of its ease of implementation make it a perfect fit for this project.

The method `tidyHtml()` requires two parameters, the HTML document as an `InputStream`, and an `OutputStream` that the parsed XHTML document will be written to. It will create an instance of Tidy set its option to create XHTML, and execute its `parse()` method that will handle the conversion.

5.3 Processing Images

When exporting a document from Google Docs to HTML via its API it can either be exported as a zip archive that contains an HTML file and all included images or as a single HTML document that links to images stored on Google’s Servers. By using the single HTML document export as source the servlet saves itself the additional step of extracting a zip archive before being able to process the document, instead the links to the images are extracted from the document and downloaded if the used configuration has the `processImages` option set. As documents could also not contain images at all, or a desired output format would want to keep the links to the version on Google’s Servers, this proved to be a more versatile solution than using the zipped export.

If the `processImage` option in the configuration file is set, the application will process the images in three steps:

First all image links of the document are extracted and stored as a Set of Strings. To do that the method `extractImageLinks()` uses a `Scanner` object that searches through the content of the document for `img` tags and extracts the URL found in their `src` attribute.

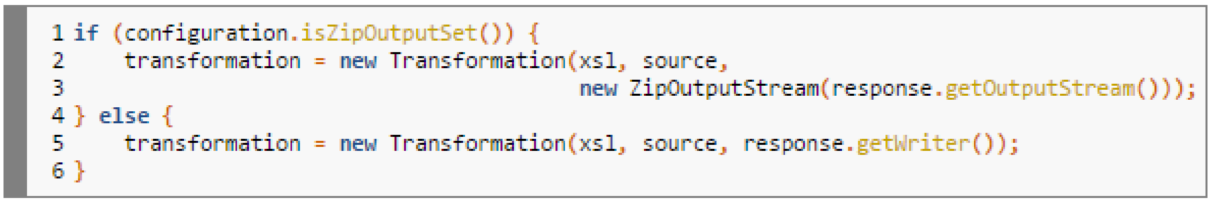
The Set generated through `extractImageLinks()` is than used as an argument for the method `downloadImageData()` which will return a key value store, with the image URLs as keys and the data of the images as respective values. The image data is stored as a Base64 encoded String, so it can be used as a value inside a `src` attribute of an `img` element.

The final step processing the images is replacing the URL in the `img` element's `src` attribute with the downloaded image data. The data that has to be placed in the src attribute is a String of the form "**data:image/<image type>;base64,<base64 encoded image>**". Since the image URL provided by Google Docs gives no indication of the type of the served image file, the application makes use of Apache Tika, a content analysis tool, that can read binary files and detect their file type, to identify the type of the downloaded image. The method `replaceImgSrcWithBase64()` loops over the key value store generated in the previous step and replaces the original image link (key) with the corresponding image data (value) in the described format.

The image processing described provides the XSL transformation with a single source document where all images are embedded inline, which is advantageous for further processing of the document, compared to having all files stored separately which would complicate the transformation process because XSL itself can not work with external binary data.

5.4 Transformation to output format

The final method that the servlet will call from within the service() method is called `setupAndRunXSLTransformation()`. As its name suggests it takes care of setting up and running the transformation to the output document. It will create an instance of the class `Transformation`, add any provided XSLT parameters and execute the transformation. The class `Transformation` serves as a wrapper around a `Transformer` object, that will handle the setup of the transformation and exposes a method that will run the final transformation. It has two constructors, one for either type of transformation scenario. Both constructors will receive the XSL stylesheet and the source document as `InputStream` as well as a third parameter that serves as an output channel for the XSL transformation to write too. The third parameter is either an instance of `ZipOutputStream` when the transformation is configured to generate a zip archive or an instance of `PrintWriter`. As shown in **Figure 6** the `ZipOutputStream` instance, created using the `ServletOutputStream` retrieved by calling `getOutputStream()` on the `ServletResponse` or the `PrintWriter` returned by calling `getWriter()` are used for either Constructor.



```
1 if (configuration.isZipOutputSet()) {
2     transformation = new Transformation(xsl, source,
3                                   new ZipOutputStream(response.getOutputStream()));
4 } else {
5     transformation = new Transformation(xsl, source, response.getWriter());
6 }
```

Figure 6: Selecting Transformation constructor. This piece of code is executed in setupAndRunXSLTransformation(), line 1 checks if zipping the output is enabled via the configuration, if so in line 2 and 3 the constructor is used that will configure the transformation to create a zip archive, otherwise in line 5 the constructor for the default transformation scenario is used.

By calling either `getWriter()` or `getServletOutputStream()` on the ServletResponse, the servlet opens an output channel for the data that is written to the body of the HTTP response that is sent back to the client. The former is used to send character data to the client, in the case of the application this is the text based document created using the default XSL transformation, the latter sends binary data which happens when a zip archive is returned.

5.5 Transformation class

As mentioned earlier `Transformation` is a wrapper class that takes care of setting up and handling the transformation, more specifically it is a wrapper around a `Transformer` object, which Java provides for processing XML documents. An instance of `Transformer` is obtained from a `TransformerFactory`. More specifically the `Transformation` class uses Saxon's version of the `TransformerFactory`, which is obtained by creating a new instance of `net.sf.saxon.TransformerFactoryImpl`. To create an instance of `Transformer` the factory's `newTransformer()` method is called with the XSLT stylesheet as a parameter.

`Transformation` exposes a method called `setParameters()` that when called will loop over the provided XSLT parameters and pass them to the method `setParameter()` of the Transformer object. `setParameter()` expects two parameters, the parameter name as a string and the parameter value. The parameter value is of the type object, which means any kind of object can be used as a value for the XSLT parameter, however the XSLT stylesheet receiving the parameters can only handle a certain subset of data types. For the scope of this project it is sufficient to only support three basic datatypes, which are `integer`, `boolean` and `string`. As all provided XSLT parameters have been passed to the Servlet as HTTP parameters, all values have been provided as a `string`. `setParameters()` will try to parse those strings to `integer` or `boolean` and default to pass a string if it fails to do so. Technically there is a larger number of datatypes that can be passed as parameters to an XSL stylesheet, however those are less relevant to the purpose of this project, especially since XSLT is capable of converting a string to other datatypes on its own.

As mentioned earlier, `Transformation` has to handle two output scenarios, for either a zipped output or a single text based document, which is decided by using the respective constructors. If the zipped output scenario is selected, the transformer will make use of a so called `OutputURIResolver` which decides where to write each generated file to. In the case of this project the `ZipOutputURIResolver` class which is explained in the next section implements the `OutputURIResolver` interface. The `OutputURIResolver` is set by setting the attribute `"http://saxon.sf.net/feature/outputURIResolver"` of the `TransformerFactory` to an instance of `ZipOutputURIResolver`, before the `Transformer` object is created.

Finally `Transformation` also exposes the method `transform()`, which when called will run the `transform()` method of the `Transformer` object, with the Source document and the corresponding output channel. If a zip archive is created, the output channel used here is a `NullOutputStream`, because the transformations default output is discarded and only the documents written using the `ZipOutputURIResolver` are relevant.

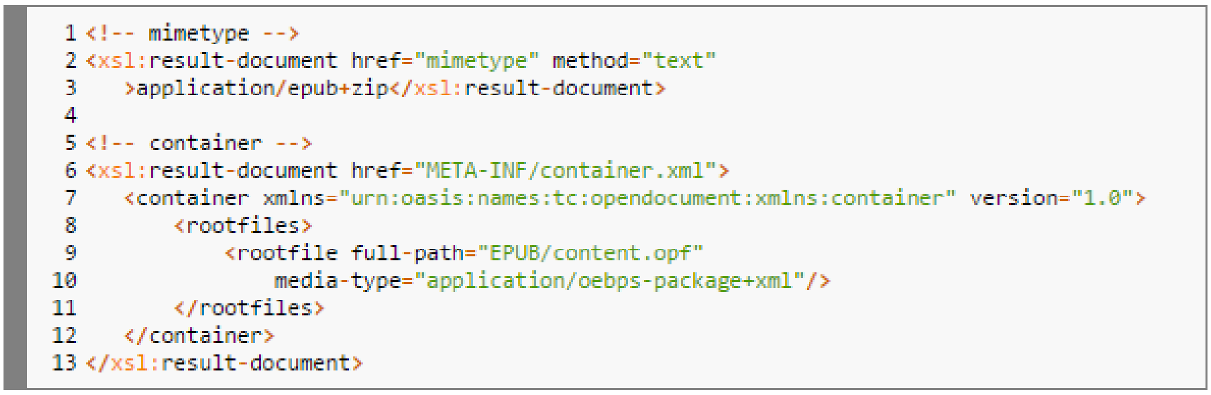
5.5.1 ZipOutputURIResolver

To provide a way for handling documents generated with `xsl:result-document` Saxon's `OutputURIResolver` interface has to be implemented, which as described by Saxon's reference [SaxonDoc] *"is used to map the URI of a secondary result document to a Result object which acts as the destination for the new document."* For the case of creating a zip archive containing the files, this is accomplished by mapping the given URI's to entries in the zip archive.

The constructor of `ZipOutputURIResolver` expects a `ZipOutputStream` as parameter, in the case of this project it receives the `ZipOutputStream` that has previously been created in the `setupAndRunXSLTransformation()` method.

When implementing the `OutputURIResolver` interface its methods **`resolve()`**, **`close()`** and **`newInstance()`** are required to be implemented, as described below.

`resolve()` will resolve the Output URI provided from the XSLT file to an entry in the zip archive. It receives the URI it has to write to via its `href` parameter, this parameter contains the address that has been provided by `xsl:result-document` in the XSL stylesheet. For each result document created `resolve()` will create a new entry in the zip archive by calling `putNextEntry(new ZipEntry(href))` on the `ZipOutputStream`. If for example the result-document calls shown in **Figure 7** are present in the used XSL transformation, this would result in creating two files in the zip archive, the first instruction will create a text file in the root directory of the zip archive which is called mimetype, while the second instruction will create a directory META-INF containing an XML file called container.xml.



```
 1 <!-- mimetype -->
 2 <xsl:result-document href="mimetype" method="text"
 3    >application/epub+zip</xsl:result-document>
 4
 5 <!-- container -->
 6 <xsl:result-document href="META-INF/container.xml">
 7    <container xmlns="urn:oasis:names:tc:opendocument:xmlns:container" version="1.0">
 8        <rootfiles>
 9            <rootfile full-path="EPUB/content.opf"
10                media-type="application/oebps-package+xml"/>
11        </rootfiles>
12    </container>
13 </xsl:result-document>
```

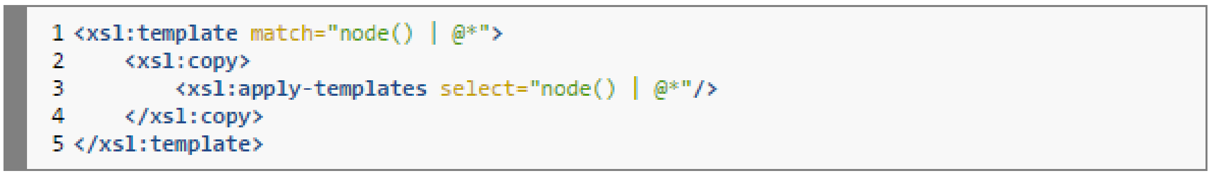
Figure 7: xsl:result-document example. This is an excerpt from the EPUB example implementation, shown are two xsl:result-document statements. The first statement in line 2 and 3 will simply create a text file in the root directory of the zip archive named **mimetype** with the content “application/epub+zip” the second statement in line 6 to 13 will create an XML file at the address **META-INF/container.xml** with lines 7 to 12 as content.

The **`close()`** method of the `OutputURIResolver` is called every time a document has been successfully written. In the case of writing to the ZipOutputStream it is used to finalize the last created entry in the zip archive by calling `closeEntry()` on the ZipOutputStream. This will make sure the resulting file is not corrupt and a valid zip archive is produced.

The method **`newInstance()`** returns an instance of the ZipOutputURIResolver. It is required to be implemented but is not used by the application in its current state.

5.5.2 XSL Transformation

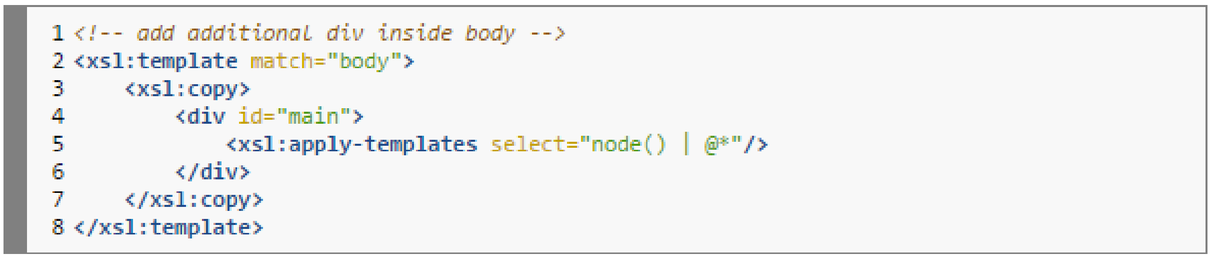
The default output configuration that was previously shown in **Figure 5** serves as a starting of point for developing XSL transformations to be used with the application. In its current state it will use the **xhtml.xsl** stylesheet, which will by default just copy the content of the source document without modification. This provides an example XHTML document representing the source for all possible XSL transformations.



```
1 <xsl:template match="node() | @*">
2     <xsl:copy>
3         <xsl:apply-templates select="node() | @*"/>
4     </xsl:copy>
5 </xsl:template>
```

Figure 8: The copy template in default transformation, will match every node in an XML document. Shown in line 1 `node()` will match all xml elements and comments and `@*` matches all attributes, the content of the template (line 2 to 4) will copy the matching node and recursively apply the same rule using the apply-templates statement in line 3.

As shown in **Figure 8**, xhtml.xsl contains a single template which will cycle through all elements of the document and copy each element including its content. This so called copy template matches on all nodes of the document as long as there is no template defined that provides a better match. XSLT works by defining templates that match certain rules defined by patterns written in the language XPATH, the XSLT processor will then find the most precisely defined template and apply its defined rules. In the case of the default xhtml.xsl configuration this will apply the same template that matches on all nodes recursively. To demonstrate how this can be extended some examples have are provided within xhtml.xsl. **Figure 9** shows a template matching the body element, which would be more precise than the copy template and would stop the recursive matching of the copy template and apply its content instead.



```
1 <!-- add additional div inside body -->
2 <xsl:template match="body">
3     <xsl:copy>
4         <div id="main">
5             <xsl:apply-templates select="node() | @*"/>
6         </div>
7     </xsl:copy>
8 </xsl:template>
```

Figure 9: Example template matching body. The template shown will match the body element of the document (line 2), copy the ele-ment itself (line 3 to 7) and add an additional div element (line 4 to 6) before applying all other templates that match all nodes inside of body (line 5).

The provided examples shows the power of XSLT, that with just a few lines of code can provide a way to modify an XML document. It shows how the application with an adjusted version of its default configuration could be used to create a modified HTML document that could be published on the web, however the flat structured nature of html documents and especially the one exported from Google Docs can prove to be challenging when the desired output format is a more complex nested or hierarchical structure, which will be examined in the following section.

6 Example front end implementations

To demonstrate the use of the application, two example front ends have been created. The first example represents the original vision of the project, to enable Google Docs to be used for publishing ebooks in the EPUB format which because of Google's own implementation of the EPUB format into docs will merely stand as a demonstration of how to use the application, also the EPUB produced using XSLT still offers certain advantages, while the second example has become an actual use case that is actively used to generate content for websites using the Kirby Content Management System.

A detailed description of the implementation of the examples would go beyond the scope of this paper, so only outlines of how they were created are shown and their use is demonstrated. In doing so the focus has been set on how to get the necessary information from Google Drive to build a URL that can be sent as a request to the converter.

6.1 EPUB Converter

The EPUB converter was the original idea behind this project, its goal is to provide a simple way of picking a document from Google Drive and convert it to EPUB with a click of a button.

6.1.1 EPUB format

At its basic an EPUB file is a zip archive containing html files representing chapters of a book as well as xml files that describe the content of the file. Additionally it can have binary resources like images, video and audio files, which for the purpose of this example however are not relevant. The structure used for the EPUB format is called Open Container Format [OCF]. The Open Container Format as well as EPUB itself are specified by the International Digital Publishing Foundation.

At a minimum an EPUB file must contain the following:

A **mimetype** file that identifies the content of the zip archive as an EPUB document. This is simply a text file with the content “application/epub+zip” and is used by reading devices to identify the file type of the document.

The root file of the document which is called a **Package Document**. This document specifies all other documents and resources used in the EPUB file. Every file included in the EPUB archive has to be listed here so EPUB reading systems are made aware of their existence. The EPUB specification allows for more than one root file which could for example be used to support multiple languages within the same file.

A file called **container.xml** which is used to point EPUB reading systems to the root file or files of the document.

A **Navigation Documen**t that describes the table of contents of the document in an html nav element. This is a simple html document that will have a nav element with links to the Content Documents and the chapters stored in them.

As well as one or more **Content Documents**, which are either XHTML or alternatively SVG. Apart from small additions, like section elements that define the content hierarchically, EPUB’s content documents are standard XHTML documents, that need no additional conversion from what is exported from Google Docs.

6.1.2 Basic Usage

The website for the EPUB converter was intentionally designed to be very simplistic with a UI as minimal as possible, to provide easy access for users. **Figure 10** shows the demo installation of the EPUB converter, which on launch features two buttons, one to pick a document from Google Drive and one to trigger the transformation that will convert the selected document to EPUB. The latter will be disabled until a document has successfully been selected.

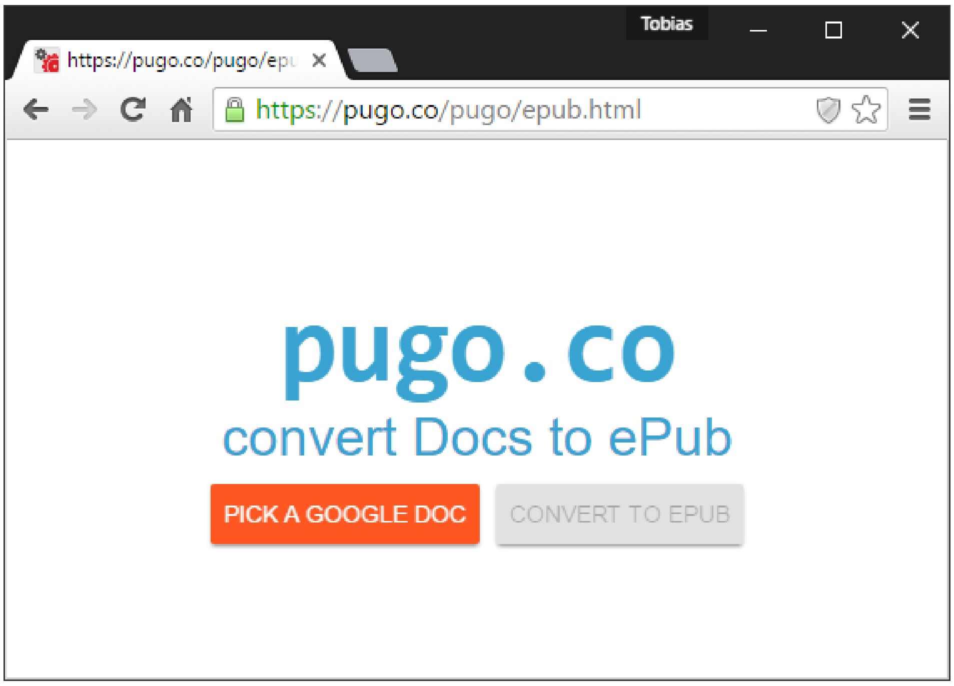


Figure 10: Demo installation of the EPUB converter

Upon pressing the button to pick a Google Doc, the application will request access to the user’s Google Drive. If the access is granted the Google File Picker, shown in Figure 11, will open and let the user select one of their Google Docs documents. The file picker can be configured only to show a certain subset of files stored in a user’s Google Drive, for the EPUB converter it has been configured to only show documents created with Google Docs.

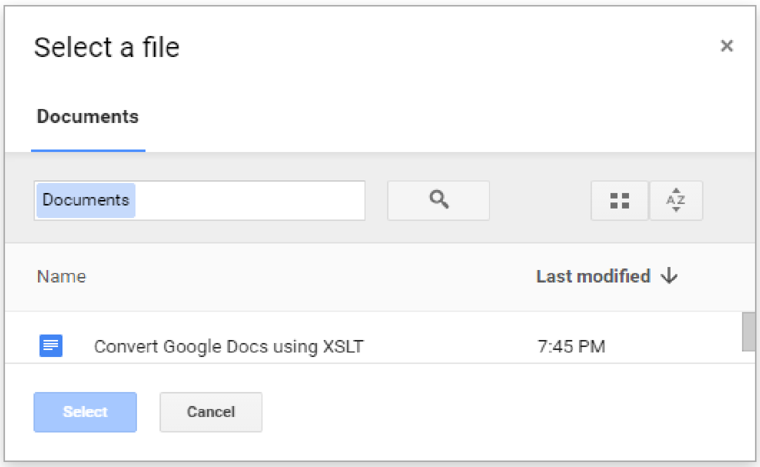
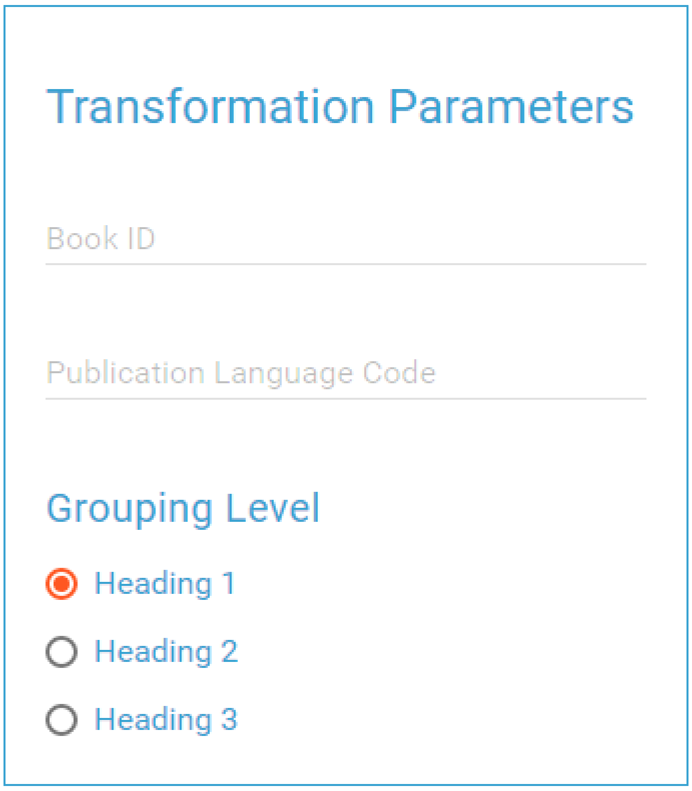


Figure 11: Google File Picker giving access to all Google Docs documents stored in the user’s Google Drive

After successfully selecting a file the “Convert To EPUB” button becomes available, and the dialog shown in **Figure 12** appears. Here the user can enter a set of parameters that have been defined in the XSL stylesheet used for the transformation to EPUB. When the convert button is pressed an HTTP POST request is sent to the converter including the parameters set using the dialog.



Transformation Parameters

Book ID

Publication Language Code

Grouping Level

[✓ Heading 1]

Heading 2

Heading 3

Figure 12: Transformation Parameters dialog, used to configure parameters that will be given to the XSL transformation

6.1.3 Implementation UI

The user interface of the EPUB converter is realized using an html form element that uses a set of input fields to temporarily store all relevant data needed for sending the request to the servlet. When a document is selected using the Google File Picker the fields shown in Figure 13 are populated with the values of the parameters required for the request. The fields are hidden from the UI but will be sent as parameters with the request upon submitting the form.
The Google File Picker returns a JSON object that contains information about the selected document, including its id and name, which are used for the needed parameters.

```
<input type="hidden" name="source" id="source">
<input type="hidden" name="token" id="token">
<input type="hidden" name="fname" id="fname">
<input type="hidden" name="mode" value="epub">
```

Figure 13: Hidden fields used to store parameters

The fields shown in **Figure 13** show the parameters that the request that is sent to the servlet must include to successfully convert the selected document to EPUB.

source will hold the html export link which is created by using the id that has been retrieved from the picker call. The export link is of the following form:
`https://docs.google.com/feeds/download/documents/export/Export?id=<ID>&exportFormat=html`

token will be filled with the OAuth access token, that has been granted when the Google File Picker was authorized.

fname is created by appending .epub to the document name retrieved from the picker call.

mode is set to epub, as explained in the servlet implementation section this will trigger the servlet to use the output configuration defined in config_epub.xml.

Upon submit the form will send a POST request to the servlet with the specified parameters. Additionally the parameters set in the transformation parameters dialog will be added to the request and passed to the transformation as XSLT parameters.

6.1.4 XSLT Stylesheet

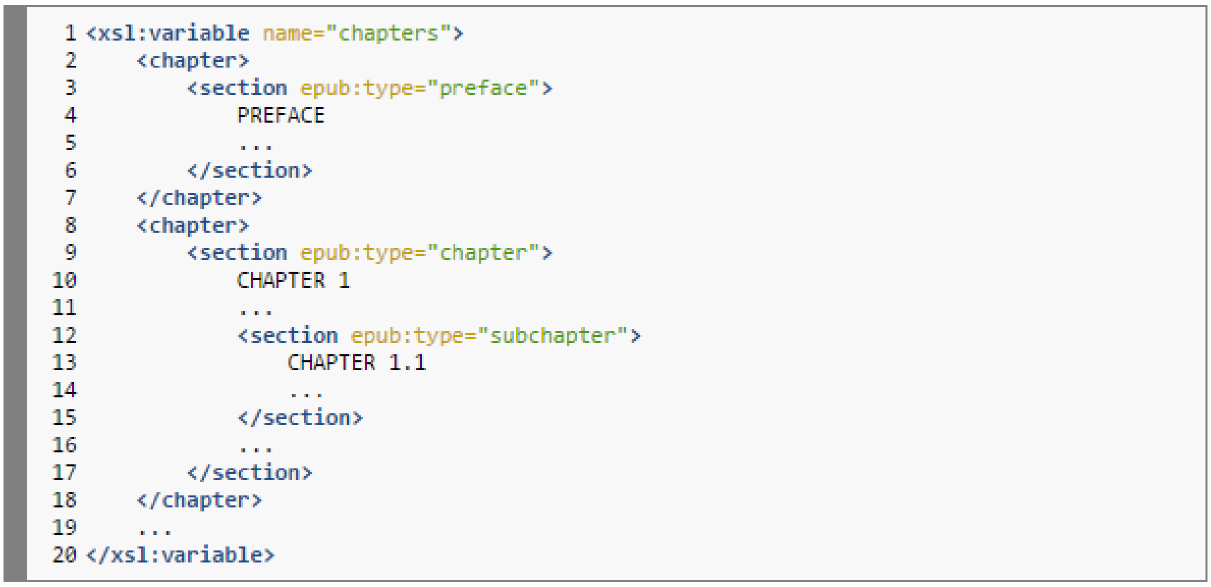
The XSLT stylesheet used for the generation of the EPUB document makes heavy use of `xsl:result-document` to generate all the files that make up an EPUB document. It will generate at least eight files, that will be written to the zip archive when used with the converter.

Additionally to the five minimum required files described at the beginning of this chapter it will generate a CSS stylesheet, an additional xhtml file for the cover of the ebook and a secondary table of context file called toc.ncx that is included for compatibility with readers that do not support the EPUB 3.0 format.

The main job of the stylesheet however lies in creating a hierarchical structure from the flat html source that Google Docs provides. To achieve this it makes use of a rather complicated set of nested `xsl:for-each-group` calls that builds a hierarchy based on the headings used in the document. `xsl:for-each-group` when used with its `group-starting-with` attribute works as described in the recommendation [XslSpec], *The nodes in the population are examined in population order. If a node matches the pattern, or is the first node in the population, then a new group is created and the node becomes its first member. Otherwise, the node is assigned to the same group as its preceding node within the population.* This means every element that appears before the first matching pattern is found will be grouped in the first group, until a node matching the defined pattern is found, after that every time a matching node occurs a new group is created.

If grouping is only done for `h1` headings this is quite straight forward, every time an `h1` is encountered a new group is created, which will be the current chapter. Everything that is encountered before the first `h1` is found will be put in a group that will create the EPUB's preface section. If however additional grouping should be done by `h2` headings it starts getting complicated. As everything before the first `h2` in the current `h1` group will be copied to the `h1` group while new groups are created for every `h2` in the current `h1` group. To achieve this the complete group created by the `h1` grouping will be stored in a temporary variable, the part of that variable that appears before the first `h2` will be copied to the current section while everything that appears from the first `h2`, will be used for the next nested for-each-group call. This process will be repeated for `h3` headings if the provided grouping-level parameter is set to 3. The process could be further repeated for more heading levels, which however could lead to memory problems because the whole set of nested for-each-groups is placed within a global variable of the stylesheet, that again contains variables of each group.

This variable will contain a preprocessed hierarchical version of the document that is accessed from other templates within the stylesheet, to generate the Package Document, Navigation Document and all the Content Documents. **Figure 14** shows how the chapters are stored in the variable, it will contain the complete document tree preprocessed into a hierarchical structure with chapters and subchapters.



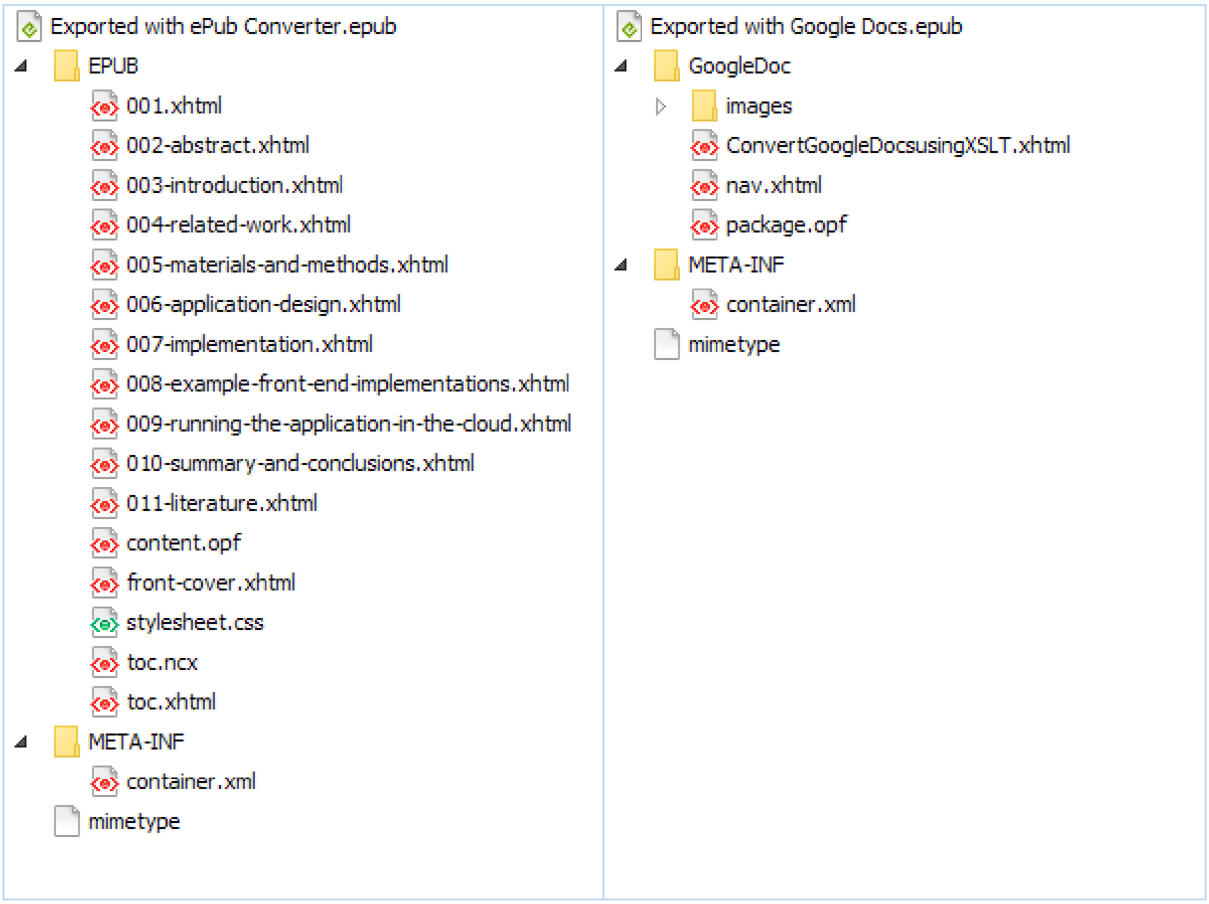
```
<xsl:variable name="chapters">
    <chapter>
        <section epub:type="preface">
            PREFACE
            ...
        </section>
    </chapter>
    <chapter>
        <section epub:type="chapter">
            CHAPTER 1
            ...
            <section epub:type="subchapter">
                CHAPTER 1.1
                ...
            </section>
            ...
        </section>
    </chapter>
    ...
</xsl:variable>
```

Figure 14: Global variable holding preprocessed hierarchical structure.

Creating multiple files per chapter using the preprocessed variable is trivial and is achieved by calling `<xsl:apply-templates select="$chapters/chapter"/>` this will look for a template that matches on the `chapter` elementa stored inside the variable. The applied template then creates a result document for each chapter found in the variable and copies its content to the resulting file.

6.1.5 Comparison with Google Docs EPUB export

Using XSLT the EPUB converter manages to create a semantic structure from a flat document by creating a set of rules based on the headings of the source document. Through the use of an XSLT parameter to configure the heading level used for creating chapters the EPUB converter is able to configure how the final EPUB is structured. The rules not only allowed the creation of "real" chapters, but also provided an additional semantic section that is relevant for a publication, by defining a simple rule, putting everything that occurs before the first occurrence of an H1, in the preface section. Similar rules could be constructed to define an appendix at the end of the document or other relevant sections. While Google Docs offers its own EPUB export, it does not provide those semantics and configurability. As shown in **Figure 15** its export to Google Docs will put the whole document into a single file, with the navigation document linking to the headings in the file. The EPUB converter will create one file for each chapter and preface, if configured to group by H2 or H3 each chapter will include additional subchapters in the same file. The navigation document will link to each file and if subchapter are created by ID to each subchapter in the corresponding file.


Exported with ePub Converter.epub
EPUB
001.xhtml
002-abstract.xhtml
003-introduction.xhtml
004-related-work.xhtml
005-materials-and-methods.xhtml
006-application-design.xhtml
007-implementation.xhtml
008-example-front-end-implementations.xhtml
009-running-the-application-in-the-cloud.xhtml
010-summary-and-conclusions.xhtml
011-literature.xhtml
content.opf
front-cover.xhtml
stylesheet.css
toc.ncx
toc.xhtml
META-INF
container.xml
mimetype

Exported with Google Docs.epub
GoogleDoc
images
ConvertGoogleDocsusingXSLT.xhtml
nav.xhtml
package.opf
META-INF
container.xml
mimetype

Figure 15: Comparing EPUB created with converter and Google Docs. The figure shows this paper converted with either export.

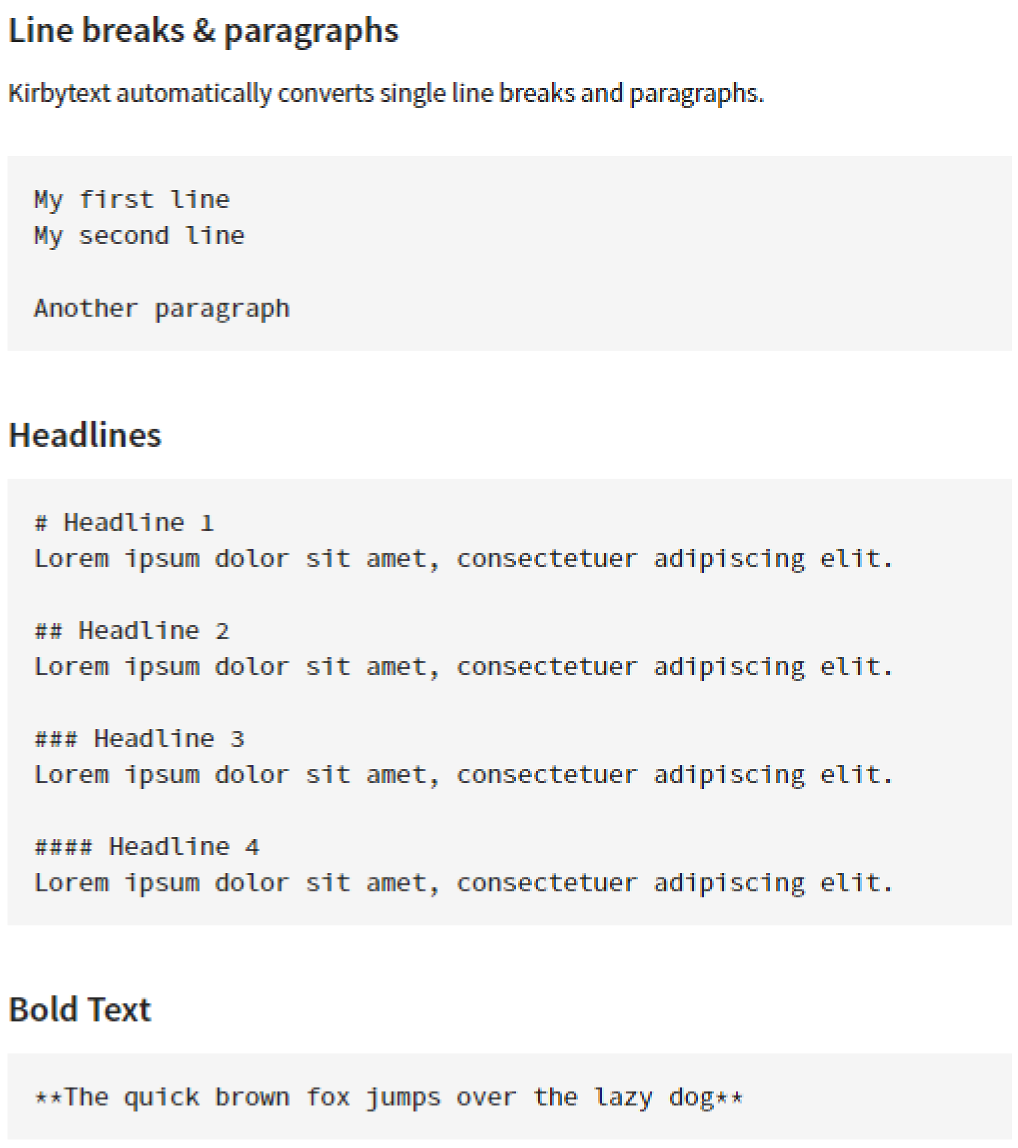
6.2 Export to Markdown for Kirby CMS

Kirby is a file based Content Management System that uses text based documents instead of a database to store its content. Working collaboratively on pages of a website can however be challenging if a pages content is stored in a single text file, so it suggests itself to look for a solution like Google Docs where collaboration is one of its biggest features and find a way to export from there. This section describes the markdown format used by Kirby CMS and how an implementation that supports the format was implemented using a Google Docs Add-on that is written in Google Apps Script.

6.2.1 Markdown format

Kirby uses an extended version of the wildly popular Markdown format that was created by John Gruber and has been adopted by many popular sites, most prominently the code hosting platform github and wikipedia. Markdown is a purely text based format that as John Gruber [Gruber] describes *"is intended to be as easy-to-read and easy-to-write as is feasible."*

Instead of tags like html markdown uses punctuation characters and linebreaks to markup or rather markdown content. As shown in Figure 16 for example paragraphs are defined by an empty line between paragraphs, a heading is defined by starting a line with the hash character (#) where the amount of characters used defines the heading level or bold text is marked up by enclosing it in two star symbols (**).



Line breaks & paragraphs

Kirbytext automatically converts single line breaks and paragraphs.

```
My first line
My second line

Another paragraph
```

Headlines

```
# Headline 1
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

## Headline 2
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

### Headline 3
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

#### Headline 4
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
```

Bold Text

```
**The quick brown fox jumps over the lazy dog**
```

Figure 16: Example markdown used by Kirby CMS [Kirby]

The markdown used by Kirby CMS is extended by storing additional meta information needed to render the webpage in the head section of the document. Since this is required for the files used by Kirby it should be noted that the markdown format created here is specifically tailored to be used by Kirby and is not necessarily meant to be used as a general markdown export.

6.2.2 Basic Usage

For the Kirby Export addon a sidebar has been implemented that can be opened from within Google Docs' Add-ons menu, which can be seen in **Figure 17**.

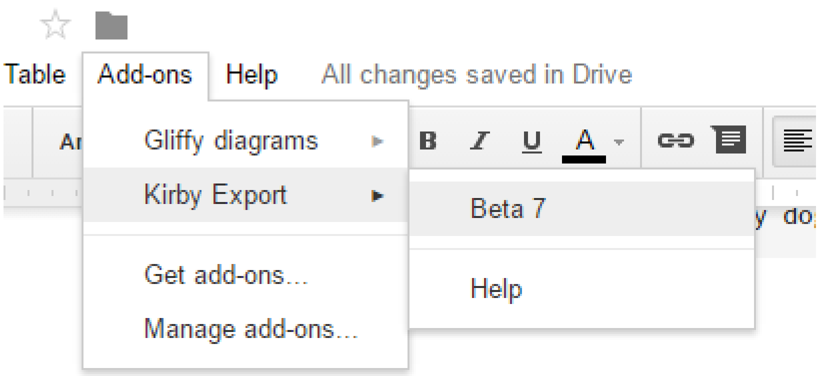
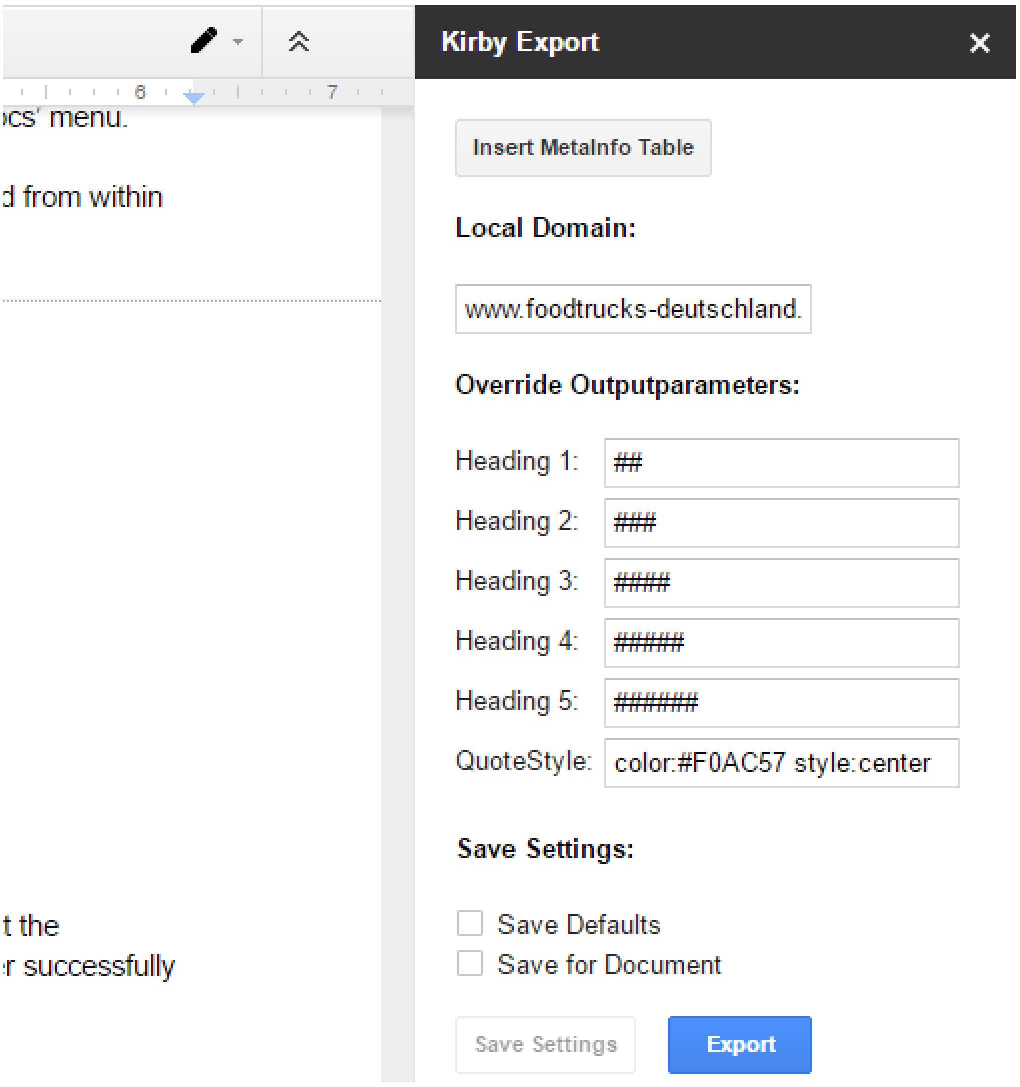


Figure 17: Google Docs menu with integrated entry for the Add-on

Upon selecting the Add-on from the menu, Google Docs will automatically detect the permissions needed to run the add-on and will request those from the user. After successfully receiving authorization the sidebar shown in **Figure 18** will open.



Local Domain: www.foodtrucks-deutschland.

Override Outputparameters:

Heading 1:	##
Heading 2:	###
Heading 3:	####
Heading 4:	#####
Heading 5:	######
QuoteStyle:	color:#F0AC57 style:center

Save Settings:

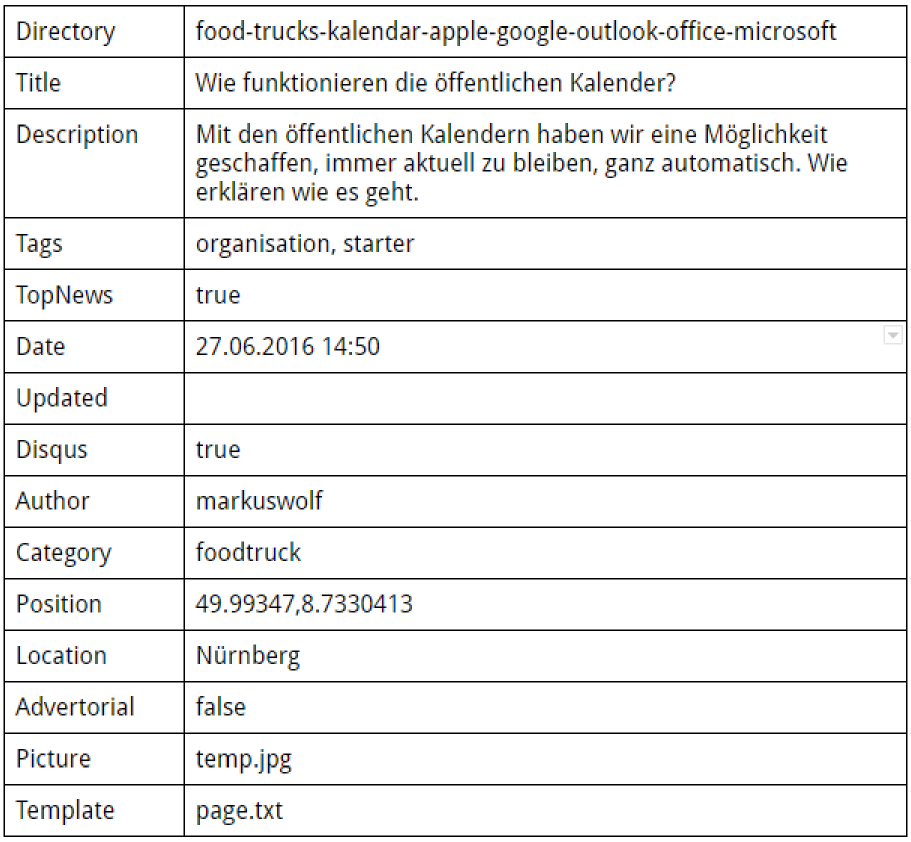
☐ Save Defaults
☐ Save for Document

Figure 18: Kirby Export sidebar in Google Docs

Using the sidebar the add-on lets the user configure certain features and parameters for the export of the document. The following sections will explain how those work.

MetaInfo Table

The “Insert MetaInfo Table” button will insert a preconfigured table meant to store the meta information needed for the Kirby Markdown format shown in Figure 19. If a table is already present on the first page of the document the add-on will detect that and notify the user instead of adding another table. As a convention that has been decided with the users of the add-on, every document that can be exported has to have the meta information table on the first page of the document. All Information that is entered into the table will be written to the header section of the exported markdown document. Additionally the add-on will create the file name for the exported document by concatenating the text entered in the row labeled “Directory” and “Template”, this will conform to the naming convention used by the content management system.

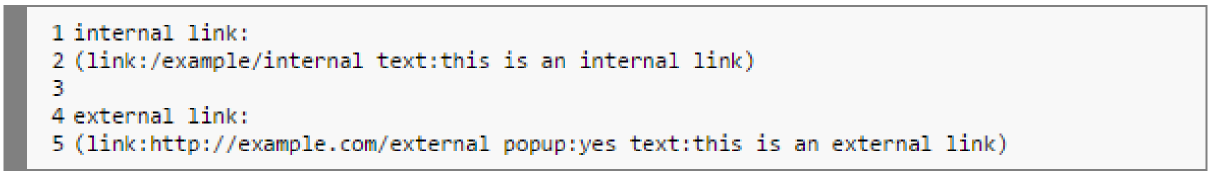


Directory	food-trucks-kalendar-apple-google-outlook-office-microsoft
Title	Wie funktionieren die öffentlichen Kalender?
Description	Mit den öffentlichen Kalendern haben wir eine Möglichkeit geschaffen, immer aktuell zu bleiben, ganz automatisch. Wie erklären wie es geht.
Tags	organisation, starter
TopNews	true
Date	27.06.2016 14:50
Updated	
Disqus	true
Author	markuswolf
Category	foodtruck
Position	49.99347,8.7330413
Location	Nürnberg
Advertorial	false
Picture	temp.jpg
Template	page.txt

Figure 19: MetaInfo table created by Kirby Export Add-on, every row listed in the table will be transferred to a section in the header of the resulting markdown file.

Local Domain

The second configurable parameter of the add-on is the local domain, which is used to distinguish between external and local links placed in the Google Docs document. Links that have the entered value as their base url will be treated as internal links, while every other link will be treated as external. The value entered here will be given to the transformation as an XSLT parameter. The transformation will strip the part entered in the field from internal links while keeping the full URL for external links, as can be seen in the example in **Figure 20**.



```
internal link:
(link:/example/internal text:this is an internal link)

external link:
(link:http://example.com/external popup:yes text:this is an external link)
```

Figure 20: internal and external links in the Kirby markdown format

Override Outputparameters

This section allows the user to override parameters set for the Output. For example by default a Heading 1 in Google Docs will produce `# Heading 1 text` in the resulting markdown export, by editing the default value of each field it is possible to markup the document differently in Google Docs than in the resulting markdown. If configured as seen in Figure 18 every heading level in Google Docs would result in a level that is one lower in the resulting markup. Additionally the QuoteStyle parameter can be configured. It will be added to the markdown that is generated when a quote is detected. The resulting markdown will look like this: `(Quote: This is my Quote color:#F0AC57 style:center)`. Because Google Docs has no specific markup for a quote, it has been decided to use the subtitle style as an indicator of a quote, which is another convention that has been decided to be used with the users of the add-on.

Settings and Export

The values entered in the fields of the sidebar can be either stored as default for the add-on itself or for the currently edited document, by ticking the corresponding checkboxes and pressing the save button. This allows for the users to save a default configuration for all documents, if used within a Google Apps account that is tied to a domain, this will also save defaults for all users of that domain.

In the current version of the add-on, which as of this writing is still in beta and actively tested, pressing the Export button will open a new Tab in the browser that makes a GET request to the converter, it is planned to replace this behaviour in the final version with a POST so no sensitive data is exposed through the URL making the request.

6.2.3 Implementation Add-on

A basic Google Docs Addon consist of a script written in Google Apps Script and an html file that describes all user facing parts like the UI and the integration into Google Docs' menu.

The sidebar of the Kirby Export add-on consists of a single html file that contains markup for the UI, some CSS as well as some JavaScript that is used to call the functions defined in the App Script file. Thanks to the Apps Script already running in the Google Drive environment, the add-on is automatically authorized and getting the necessary parameters for using the converter can be achieved with a few function calls. The Apps Script will gather the needed parameters and pass them to the UI, to do this the function `getExportParams()` creates a JavaScript object containing the **source**, **fname** and **token** parameters that will be sent with the request. The Add-on will call the `getExportParams()` function every 30 minutes while the sidebar is open, so the access token gets refreshed, as it is only valid for a certain amount of time. In the html document for the sidebar is a corresponding function by the same name, that will call the function defined in the Apps Script, and hand the created object to the callback function `setTransformURL()` that will create a URL that will be opened in a new tab when the export button is pressed and send the GET request to the servlet. Additionally all the parameters that have been set by the user in the sidebar will be added to the URL before the request is sent.

6.2.4 Stylesheet

The stylesheet used to convert the html export from Google Docs to markdown is fairly simple and can be written by someone that has a good understanding of XSLT in just a few hours. Except for the handling of the meta information table it consist of basic mapping from tags to markdown markup, as described in **Figure 21**.

```
<p>Paragraph</p>    ::= Paragraph\n\n
<h1>Heading 1</h1>  ::= # Heading 1
<h2>Heading 2</h2>  ::= ## Heading 2
...
```

Figure 21: basic mapping of html to markdown

There are however a few exceptions and caveats to take care of, for instance Google Docs does not use tags like strong and em to markup bold or italic text, instead all text blocks are written to span elements which utilize CSS classes to set the style, like font-weight and size. The XSLT stylesheet handles this by using a function that will check the CSS class for the corresponding font-weight and style.

Although the stylesheet for the conversion to markdown is not hard to understand and implement, it seems a general solution is not possible without communicating certain conventions with the users of the addon, which simply stems from the limited amount of actual markup that can be done inside of Google Docs which in this case has been done with the meta information table or the way quotes are recognized.

7 Running the Application in the Cloud

When first developing the application, deploying it in the Cloud was always a desired outcome. Initially the application was developed for Google App Engine, which seems like a clear choice when already dealing with other Google Products and APIs. However an issue where when the application was run from App Engine it was impossible to output UTF-8 encoded documents from the transformation, instead all documents were generated as us-ascii, which caused special characters to disappear, ruled App Engine out as the solution.

As with the developed application itself, a more general solution was chosen by using Docker, a product that in recent years became the standard for deploying applications in the cloud. Docker runs software in what is called a container, which is similar to the way a virtual machine works however instead of running a complete operating system it only runs certain components that are packaged to a single container image. As described on Dockers website, *“Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server.”* [Docker]. Thus, instead of running a virtual machine a docker instance requires far less resources.

Docker is supported by all major cloud vendors including the offerings of Amazon, Google and Microsoft.

Docker uses a file called Dockerfile to build container images that can be run on any computer that runs the docker software which is available for all major operating systems. Using the following Dockerfile shown in **Figure 2**2, it is possible to build and deploy a docker image that will run a version of the converter that was compiled to a war file on an Tomcat 8 Server.

```
1 FROM tomcat:8
2 COPY pugo-co.war /usr/local/tomcat/webapps/pugo-co.war
```

Figure 22: DockerFile used to run the converter, the command in line 1 will load a Docker image containing an instance of the Tomcat 8 application server from Docker’s repository, the command in line 2 will copy the war file into tomcat’s webapp dir.

Executing the command “`docker build -t pugo-co .`” inside the directory that contains the Dockerfile and the war package will create a docker image named pugo-co, that can be run locally with the command “`docker run -p 8080:8080 pugo-co`”. After executing the command the image can be accessed at dockers default ip address and the in the run command given port **`http://192.168.99.100:8080/pugo-co/`**. The same image can be run by either dedicated Dockers hosts, cloud vendors that offer hosting for just the image, or by running a virtual machine instance that is configured to run docker images which is a scenario that is supported by a large number of vendors that do not support dedicated Docker solutions.

8 Summary and Outlook

Moving away from the original idea behind the project of creating a single purpose tool to publish EPUB documents towards an application that can be configured to produce any number of possible formats proved to be beneficial, which is shown by the fact that the Kirby Export add-on is already actively tested and used. Feedback from users so far has been extremely positive, as it greatly improves their workflow. They collaboratively work on the document in Google Docs, which thanks to features like automatic revisions and commenting allows them to work on different continents in different timezones and export the final version using the Google Docs add-on developed as part of this project.

As demonstrated by both implemented examples, the application can serve as a mutli purpose solution to extend Google Docs by supporting additional output formats that can be tailored to a specific purpose like feeding a content management system. However, stemming from the nature of Google Docs data format, it seems possible implementations will usually be tailored to a specific use case, where conventions of how to markup a document inside of Google Docs need to be communicated, while more general solutions will be harder to implement or just lack a certain set of features. This can be seen by looking at Google Docs' integrated export formats and their lack of semantic markup. Furthermore specifically the example implementation for EPUB has shown how following those conventions can help to create a format containing a minimal level of semantics, while more complex semantic solutions seem only possible in theory.

Using Google Docs as a general Content Creation platform beyond its designed purpose is possible, however depending on the output type that should be created can be quite challenging, which is shown by the hurdles that even a language like XSLT, which is designed to create hierarchical documents, has to take when trying to build a hierarchy out of a flat source. However with the help of Google Doc's API's, which after working with them are obviously designed to let developers make up for Google Docs' own shortcomings, its functionality can be greatly extended. For example as this paper itself was written in Google Docs, its final formatting to conform to the standard of a scientific paper has to be done externally, and most likely will be done in Google Docs' biggest competitor, which still is the offline version of Microsoft Word. The idea to actually use the project described in this paper and extend it with a configuration that provides a stylesheet that transforms a Google Docs document automatically into something conforming to the standard needed for a scientific paper seems not too far fetch, but would of course go beyond its own scope.

The demo installation of the application is currently hosted on a virtual server of a standard hosting provider, as this is more than sufficient for its current user base, however if a growing number of users would have to be supported, Docker proved to be invaluable and specifically simple to implement for the way the application was designed. With a wide range of cloud vendors supporting the format the application could support a large number of users when hosted in the cloud.

A big possibility for extending the application in the future would not only be by adding more supported file formats, but also by extending the application into the other direction by supporting different source providers. As the expected source format currently consist of a standard html document, any Cloud Office Solution that supports exporting to html could be used as a potential candidate to be used with the application. Furthermore the application even in its current state can be used to convert any html document using XSLT by developing additional XSL transformations.

Literature

Docker	Docker Inc: What is Docker? https://www.docker.com/what-docker last download: Aug. 1. 2016
EPUBGuide	International Digital Publishing Forum: EPUB 3 Accessibility Guidelines http://www.idpf.org/accessibility/guidelines/content/xhtml/sections.php last download: Aug. 1. 2016
Gruber	John Gruber, Markdown: Syntax http://daringfireball.net/projects/markdown/syntax last download: Aug. 1. 2016
Kirby	Bastian Allgeier GmbH: Formatting text https://getkirby.com/docs/content/text last download: Aug. 1. 2016
JavaTutorial	The Java EE 6 Tutorial: What is a Servlet? http://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html last download: Aug. 1. 2016
JavaDevGuide	Sun Java System Web Server 7.0 Developer's Guide to Java Web Applications https://docs.oracle.com/cd/E19146-01/819-2634/abxbe/index.htm last download: Aug. 1. 2016
JavaSpec	James Gosling, Bill Joy, Guy Steel, Gilad Bracha, Alex Buckley (2015): The Java® Language Specification - Java SE 8 Editioition, Oracle America Inc.
OCF	EPUB Open Container Format (OCF) 3.0, International Digital Publishing Forum http://www.idpf.org/epub/30/spec/epub30-ocf.html last download: Aug. 1. 2016
SaxonDoc	Michael H. Kay: Interface OutputURIResolver http://www.saxonica.com/documentation9.0/javadoc/net/sf/saxon/OutputURIResolver.html last download: Aug. 1. 2016
ServletSpec	Danny Coward, Yutaka Yoshida (2013): Java™ Servlet Specification Version 2.4, Sun Microsystems, Inc. http://download.oracle.com/otn-pub/jcp/servlet-2.4-fr-spec-oth-JSpec/servlet-2_4-fr-spec.pdf last download: Aug. 1. 2016
XslSpec	W3C: XSL Transformations (XSLT) Version 2.0, W3C Recommendation 23 January 2007 https://www.w3.org/TR/xslt20/#xsl-for-each-group last download: Aug. 1. 2016

Erklärung der Kandidatin / des Kandidaten

Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

[illegible]

30.6.2016
Datum

Unterschrift der Kandidatin / des Kandidaten