XML to PDF with ConTeXt

Background

One of the DocWolves product lines is an on-line production environment for documents related to decision workflows. From the users' input, we create published HTML pages and PDF documents.

Our clients input text fragments into a web form built around a customized installation of CKEditor. CKEditor is a free collection of javascript components for WYSIWYG editing of HTML that runs inside the web browser, without needing any client-side plugins (see http://ckeditor.com for more details). We save these text fragments along with various bits of meta-information in a MySQL database. When the editing cycle is done and the client decides to publish a document we combine these various text fragments, meta-information and any needed images into either an HTML page or an XML file. In the case of the XML file, that is then converted into PDF using ConTeXt.

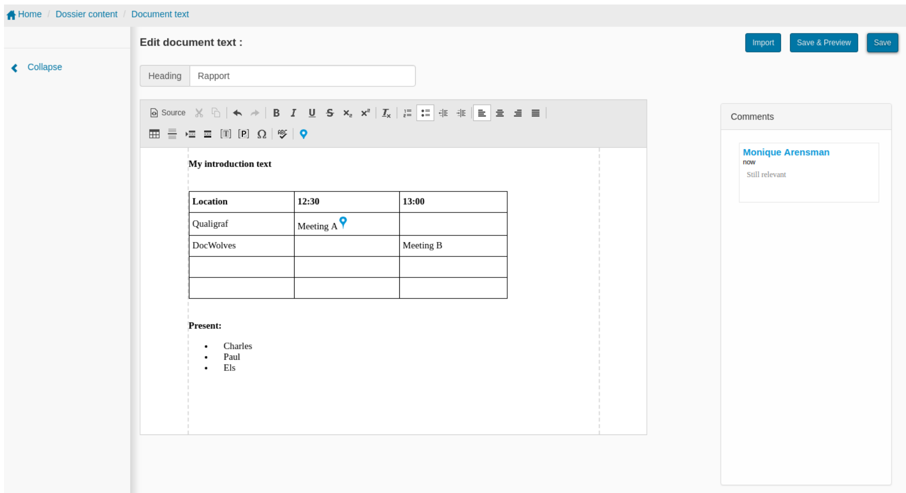

Figure 1. Example of the web-based input screen

Various Perl scripts control and monitor all stages of the document creation process. Before being stored, the raw textual (HTML) input is passed through a parser that removes anything that cannot be handled properly by both the HTML and the XML-to-PDF backends. But the monitoring system starts even earlier: cut and paste within the CKEditor is modified to prevent weird HTML (for example from word processing software) from entering the input stream. Also, various options have been disabled in the CKEditor.

To make sure that we know exactly what is in our records, the input parser also converts the angle brackets of the acceptable HTML tags into square brackets in the stored data, and it removes any unsupported HTML attributes. At the same time square brackets in the input are converted to HTML character entities.

Using this intermediate format for storage means that at the output stage, the backend file generator can convert any and all angle brackets it encounters into character entities, thus ensuring that the generated output is always valid XML/HTML.

Besides blocking weird (and potentially damaging) input, there is a another reason for all the filtering. The generated published documents often have to adhere to a specific client house style, with (for example) predefined margins, font settings, and a page background. Thus, removing visual markup that conflicts with the house style settings is a secondary but quite important function of the filtering system. By the way, those style settings are also stored in our system, somewhere separate from the actual text fragments. I do not have to explain here why that is handy, TeX users are quite familiar with separating content from presentation, but for many outside the TeX community this still a foreign concept.

Here is a small part of a text fragment as it is actually stored inside the database:

```
[p align="left" class="western" lang="fr-FR"][br  /][/p][table cellpadding="7"
cellspacing="0" width="615"][tr ][td style="border: 1px solid #000000;
padding: 0in 0.08in" valign="top" width="599"][h3 class="western"
lang="fr-FR"]RAPPORT DE PRESENTATION[/h3][h3 class="western" lang="fr-FR"]BUDGET
PRIMITIF 2018 - EAU[/h3][/td ][/tr ][/table ][p align="justify" class="western"
lang="fr-FR"][br  /][/p][p align="justify" class="western" lang="fr-FR"]Le
budget du service de l'Eau s'équilibre en Dépenses et Recettes, aussi
bien en Fonctionnement qu'en Investissement à hauteur de [b ]11 459
610,00 €, [/b]soit :[/p][p align="left" class="western" lang="fr-FR"][br
/][/p]
....
[p align="justify" class="western" lang="en-US"][img
data-image-id="2279" height="305" src="data:image/png;base64,iVBORw0KGgo
....
Y2UyODA1hnDS2gAAAABJRU5ErkJggg==" width="675"][/p][p align="justify"
class="western" lang="fr-FR"][br  /][/p]
```

There are normally no line breaks in the tag portions of the database record, the line breaks are added here just for this example. In the actual input text, we preserve the whitespace characters from the input.

As you can see, there is still a fair bit of HTML and CSS supported in the system.

ConTeXt input files

The main ConTeXt input is the generated XML file. This contains not only the collected text fragments, but also a block of house style settings, and some meta-information about the document.

The XML processing takes place via an environment file called `dw-workflow`. Most of the content of this file is Lua code, but there is also a bit of plain TeX code and a few ConTeXt patches included in there.

There is only one other include file, and that contains our typescript definitions.

In the future `dw-workflow` may be split into a few more generic modules so that code may be reused in other DocWolves products, but for now everything is in one (fairly long) file.

XML structure

The listing below shows the general structure of the generated XML file.

```
<?xml version="1.0"?>
<root>
   <settings>
      <setting name="papersize" value="A4"/>
      <setting name="textwidth" value="170"/>
      <setting name="..." value="..."/>
   </settings>
   <documentinfo>
     <meta name="DOSSIER_REF" value="1819"/>
     <meta name="..." value="..."/>
```

```
    <document_content>
      <content format="html">
        <fragment id="19804" dos_id="2157" dsd_id="5562">
          <h2>asdf</h2> ...
        </fragment>
        <fragment>...</fragment>
      </content>
    </document_content>
</root>
```

XML settings and setup

The <meta> information is directly copied to the PDF XMP info, and otherwise ignored.

The <setting> tag is where the style options are communicated to ConTEXt. Currently, there are about two dozen settings. As is to be expected, most of them deal with typical page setup. There are settings that are converted into arguments for \setuppapersize and \setuplayout, settings for \setupfootertexts and \setupheadertexts, and settings for \switchtobodyfont.

A few of the settings are more interesting. In particular, there is a setting for the font size and a small group of settings for setting up the optional background image for each page.

As is customary in CSS, the document font size does not only set up the size of the main text font, but it also sets up the relative sizes of the headings and footnotes, the white space before and after headings, et cetera, using a multiplier of the base font size value. And since these multiplications can produce odd body font sizes, there is a Lua function that not only sets up the actual ConTEXt commands, but also executes the needed \definebodyfontenvironment commands. The start of this function looks like this:

```
function userdata.setupheads(argsize)
    local thesize = string.gsub(argsize,"pt",'')
    local size = tonumber(thesize)
    local e  = math.floor(size*cssparser.typemaps.font_size['xx-large'] + 0.5)
    local d  = math.floor(size*cssparser.typemaps.font_size['x-large'] + 0.5)
    local c  = math.floor(size*cssparser.typemaps.font_size['large'] + 0.5)
    local x  = math.floor(size*cssparser.typemaps.font_size['small'] + 0.5)
    local xx = math.floor(size*cssparser.typemaps.font_size['x-small'] + 0.5)
    context.definebodyfontenvironment({e ..'pt'})
    context.definebodyfontenvironment({d ..'pt'})
    context.definebodyfontenvironment({c ..'pt'})
    context.definebodyfontenvironment({x ..'pt'})
    context.definebodyfontenvironment({xx ..'pt'})
    -- h1
    context.setuphead({'part'},
      {align="flushleft", page='no',placehead='yes',number='no',
       style='\\switchtobodyfont['..e ..'pt]\\bf ',
       before='{\\blank['..xx ..'pt]}',
       after='{\\blank['..xx ..'pt]}',
       aligntitle='yes',
       })
    ...
```

Per-client page background images are used so that we do not have to write settings for each and every client with a special logo in the page top or bottom. We have set up the system such that for each client, dedicated single-page PDF files are searched for. If found, these are then added to the page on their own layer. For each combination of document type and paper size, there are two possible PDFs: one for the first page, and another for all following pages (often client logos are larger on the first page of a document). Such backgrounds are searched for in four locations: the client folder with and without paper size, and the global locations for the same (the global locations contain single-page empty files).

The code that deals with this is:

```
local paths = {}
paths[#paths+1] = valueordefault(settings.clientbackgrounddirectory .. '/'
                           .. settings.papersize,nil)
paths[#paths+1] = valueordefault(settings.clientbackgrounddirectory,nil)
paths[#paths+1] = valueordefault(settings.backgrounddirectory .. '/'
                           .. settings.papersize,nil)
paths[#paths+1] = valueordefault(settings.backgrounddirectory,nil)
context.setupexternalfigures({directory = table.concat(paths,',') })
if settings.clientbackgroundname and #settings.clientbackgroundname>0 then
   context.resetbackgroundfigure(settings.clientbackgroundname,"1")
end
```

where `\resetbackgroundfigure` is a separate macro that ensures the PDF image is included:

```
\def\pagebackgroundfigure{}

\def\resetbackgroundfigure#1#2%
  {\gdef\outputpagen{#2}%
   \gdef\pagebackgroundfigure
       {\externalfigure[#1-page-\outputpagen.pdf]
          [width=\the\paperwidth,height=\the\paperheight,page=1]}}

\defineoverlay
  [pagebackground]
  [{\pagebackgroundfigure \gdef\outputpagen{n}}]

\setupbackgrounds
  [page]
  [state=repeat,background={pagebackground}]
```

Now, if the above looks a bit sneaky to you? ... yeah, I know!
This setup is inherited from an older (`mkii`) project, where continuous redefining of the `\pagebackgroundfigure` was necessary. And it works fine, so I saw no reason to implement something else.

Before getting into the actual processing of the XML, let me introduce some of the `dw-workflow` code that is needed for almost all XML, and especially HTML processing. First, there is the setup that connects XML tags to Lua functions.

```
\startxmlsetups xml:oursetups
      \xmlsetfunction {\xmldocument}{*}               {xml.functions.panic}
      \xmlsetfunction {\xmldocument}{root}            {xml.functions.flush}
      \xmlsetfunction {\xmldocument}{settings}        {xml.functions.settings}
      \xmlsetfunction {\xmldocument}{setting}         {xml.functions.setting}
      \xmlsetfunction {\xmldocument}{document_content} {xml.functions.document}
      \xmlsetfunction {\xmldocument}{content}         {xml.functions.flush}
      \xmlsetfunction {\xmldocument}{fragment}        {xml.functions.fragment}
      \xmlsetfunction {\xmldocument}{h1}              {xml.functions.h1}
      ....
\stopxmlsetups

\xmlregistersetup{xml:oursetups}
```

At the dotted line are all the functions for the separate HTML tags we support, which are skipped here for brevity.

The `\xmlsetfunction` for `*` is a visualization trick. The `panic` function typesets all child data in a bold, red, and ugly way. Even with all the precautions we take for making sure the input is predictable, it is still possible that something sneaks through. One example of that happening was when we updated the whole workflow subsystem in development to support some extra tags, but we had forgotten to update the `dw-workflow` accordingly. The `panic` function's output made that mistake clearly visible during testing.

Next up in the dw-workflow is a set of definitions like this:

```
\def\htmlentity#1#2#3#4{\xmlsetentity{#2}{#1}}
\def\htmltexentity#1#2#3#4{\xmltexentity{#2}{#1}}

% latin chars
\htmlentity{À}{Agrave}{192}{Capital a with grave accent}
\htmlentity{Á}{Aacute}{193}{Capital a with acute accent}
\htmltexentity{~}{nbsp}{160}{Non-breaking space}
...
```

There are some 250 lines of these, adding support for all of the predefined HTML entities. ConTEXt converts numeric entities automatically, but the named HTML versions need explicit definitions. There are four arguments because this information is converted to TEX macros from a HTML table listing all the entities. In an earlier stage of development, the third and fourth arguments were used to typeset a ConTEXt table for comparison to that HTML table.

XML text fragments

The text fragments are written out as <fragment> tags in the XML file, and the content of each of those is basically HTML with a bit of optional inline CSS. Let's start with a bit of example listing:

```
<fragment id="18202" dos_id="3279" dsd_id="6980" title="no">
  <p style="text-align:center;"><strong>COMMISSION &test; PERMANENTE</strong></p>
  <p style="text-align:center;">Séance du </p>
  <p style="text-align:center;"><strong>DOSSIER N°</strong> <strong></strong></p>
</fragment>
<fragment id="18203" dos_id="3279" dsd_id="6980" title="no">
  <table border="1" cellpadding="1" cellspacing="1" style="width:639px;">
    <tbody>
      <tr>
        <td style="height:22px; width:626px;"><br/>
          <b>Politique : </b>COPY<br/><br/>
          <b>Programma: </b><placeholder>Arensman Monique</placeholder><br/><br/>
           Opération : SOME TEXT<br/><br/><br/>
         </td>
      </tr>
   </tbody>
  </table>
 </fragment>
```

Most of the attributes of the fragment tag in the XML example above are for debugging purposes only and are ignored during typesetting. The one processed attribute is title. As you can see in figure 1, each text fragment can have a system-supplied heading. If that heading is not given, we add an extra blank to give some visual separation between adjacent text fragments.

The fragment also supports an is_framed argument, which is not used in the above example. This creates a border around the whole fragment using \starttextbackground. The system makes use of text backgrounds to make sure that the fragment can still break across pages, which is an absolute requirement.

```
function xml.functions.fragment(t)
   local framed = false
   if t.at.is_framed and t.at.is_framed == 'true' then
      cssparser.prependstyle(t,"border: 1px solid black; padding: 10px;")
      framed = true
   end
   context.flushsidefloats() -- clear left/right divs
   if t.at.title and t.at.title:lower() == "no" then
     context.blank({'line'})
   end
   if framed then
     local args = textbackgroundarguments(t)
     context.definetextbackground({'fragmentbackground'.. tonumber(t.at.id)},args)
```

```
    context.starttextbackground({'fragmentbackground'.. tonumber(t.at.id)})
  end
  lxml.flush(t)
  if framed then
    context.stoptextbackground()
  end
end
```

Rather than try to process the various background options right in the above function, the requested frame is converted into CSS statements. The function `textbackgroundarguments()` converts that CSS specification into arguments for `\definetextbackground`.

Interpreting CSS specifications

Parsing CSS specifications is actually quite easy, especially for the only specification format we support right now: in-line style elements. A smallish Lua function does all of the initial work:

```
local P, S, C = lpeg.P, lpeg.S, lpeg.C

function cssparser.parse(t)
  local result = {}
  local found = {}
  if t.at.style then
    local function store(a,b)
      found[a] = b
    end
    local skipspace    = S(" \t")^0
    local colon        = P(":")
    local semicolon    = P(";")
    local eos          = P(-1)
    local somevalue    = (1 - (skipspace * (semicolon + eos)))^1
    local somekey      = (1 - (skipspace * (colon + eos)))^1
    local cssmatch = ((C(somekey) * skipspace * colon * skipspace * C(somevalue))
                     /store * skipspace * (semicolon + eos) * skipspace)^1 + eos

    lpeg.match(cssmatch,t.at.style)

    for i,v in ipairs(cssparser.registered) do
      local k = v[1]
      if found[k] then
        local f = v[2]
        f(t,result,k,found[k])
        if cssparser.inherited_trait[k] then
          if found[k] ~= 'inherit' then
            cssparser.inherit(t,k,found[k])
          end
        end
        found[k]=nil
      end
    end
    for i,v in pairs(found) do
      cssparser.report('unknown css property: '..i)
    end
    for k,v in pairs(result) do
      if v == 'inherit' then
        result[k] = cssparser.inherited(t,k, cssparser.inherited_trait[k])
      elseif v == 'initial' then
        result[k] = cssparser.inherited_trait[k]
      end
    end
  end
  return result
end
```

The first half of the above function is the LPEG match needed to split the string into a key–value table. The second half takes care of interpreting the registered traits. There are two separate tables that are used in this process:

`cssparser.registered`
: is a table of CSS traits that are known to the system. Each of the array values is a further array with two items: the name of the trait, and a processing function for that trait. These functions are then called to interpret the trait's CSS specification. They take care of things like converting special color names and oddball length specifications to something ConTEXt and MetaPost understand. The main table is an array instead of a dictionary because ordering is important in the case of shortcut traits, as we will see later.

`cssparser.inherited_trait`
: contains the initial values for inheritable traits. The processing taking place here is essentially a callback, since normally inheritance is handled by the processing functions in the previous loop.

Both arrays are set up by calls to `cssparser.register`:

```
function cssparser.register (k,f,inherited)
   cssparser.registered[#cssparser.registered+1] = {k, f}
   if inherited then
      cssparser.inherited_trait[k] = inherited
   end
end
```

Throughout the rest of `dw-workflow`, there are dozens of calls like this:

```
cssparser.register('font-size',
   function (t,result,key,value) result[key] = value end, '11pt')
```

The simplest of those calls use an inline function as seen above. The more complicated ones define the function separately, just because that produces nicer formatting of the source. 'Complicated' is perhaps too big a word: the CSS parser callbacks do not do all that much work besides verifying the input syntax and resolving CSS shortcuts.

```
local function parse_padding_shortcut(t,result,key,value)
   local function process(a,b,c,d)
      local t,r,l,b = trlb(a,b,c,d)
      result[key ..'-top']    = cssparser.htmldimension(t)
      result[key ..'-bottom'] = cssparser.htmldimension(b)
      result[key ..'-right']  = cssparser.htmldimension(r)
      result[key ..'-left']   = cssparser.htmldimension(l)
   end
   local pattern = (cssparser.matches.width^-4/process)
   pattern:match(value)
end

local function parse_one_padding(t,result,key,value)
   local function process(a)
      result[key] = cssparser.htmldimension(a)
   end
   local pattern = (cssparser.matches.width^1/process)
   pattern:match(value)
end

cssparser.register("padding",       parse_padding_shortcut)
cssparser.register("padding-left",  parse_one_padding)
```

As you can see, there are some other helpers in the `cssparser` table. For example, `htmldimension` converts the CSS width keywords (`thin`, `medium`, `thick`) into discrete HTML lengths. There is also `htmlcolor`, which converts named colors into HEX values. The goal of these functions is not to produce the final trait value that will be used for typesetting, but just to get rid of some of the idiosyncrasies of CSS.

The cssparser.matches table contains a small set of predefined LPEG matches for common CSS data types. Besides width, there are definitions for length, color, and border_style.

The previous takes care of parsing CSS specifications. But how to use them?

When inside one of the XML tag processing functions, it is normally enough to call either cssparser.style() or cssparser.styled(). The latter takes care of implicit inheritance, the former just returns the locally specified CSS trait, if there is any.

```
function cssparser.styled(t,name)
  if not t.__style then
     t.__style = cssparser.parse(t)
  end
  if t.__style[name] then
     return t.__style[name]
  -- the next elseif is not done in cssparser.style
  elseif cssparser.inherited_trait[name] then
     return cssparser.inherited(t,name)
  end
  return nil
end
```

It is worth noting that cssparser.styled() is not just used for explicit inherit. What it actually returns is either a local value, or the value you would get *if* explicit inherit *was* present, even if it is not actually there. This turned out to be quite useful, for example to query the current font size and text. Both of these are quite often needed during processing, even in tags that do not actually inherit the value.

The functions cssparser.inherited and its companion cssparser.inherit are used to query and set inherited traits. They actually allow arbitrary keywords, so they can also be used to set up inheritance for ad-hoc values, as that turned out be quite useful. For example when dealing with the implicit CSS state, as in whether the current XML subtree is part of a floating object or not.

We have already seen cssparser.prependstyle(). In that example, it was used with a literal CSS string. But the most prevalent use of that function is to convert HTML attributes into CSS traits. We will see a usage example further down in this article.

Attribute values

Some CSS attribute values can be used directly in the Lua code, like for the various keyword-valued traits. These are commonly used for decisions in the processing step, and do not actually need to be passed to ConTEXt. Most of the ones that *do* need to be passed on can be handled by a simple hash. The table cssparser.typemaps contains a small set of tables that map CSS keywords to ConTEXt keywords for this purpose.

For example:

```
cssparser.typemaps.text_align = {
   left    = 'flushleft,verytolerant,extremestretch',
   right   = 'flushright,verytolerant,extremestretch',
   center  = 'center,verytolerant,extremestretch',
   justify = 'verytolerant,extremestretch'
}
```

At the moment, there are typemaps for float, font_size, font_weight, list_style_type, text_align and vertical_align.

Very handy, but this does not work for all attribute values. In particular, dimensions and colors require more attention.

CSS dimensions can have a fairly elaborate list of units, and only the basic ones actually match up with TEX dimensions. Our current system does not support all of the possible relative CSS dimensions like 'X percent of the viewport' and 'X percent of the width of the zero', but it does handle the absolute dimensions, em / ex, bare

numbers, and \%. We use a helper function from ConTEXt itself to convert the value from its CSS format to a number of TEX points.

```
function styledimension (val, full)
  if not full then
     full = userdata.settings.actualwidth
  else
     full = string.gsub(full,"pt",'') -- just in case
     full = tonumber(full)*65536
  end
  ret = (xml.css.dimension(val,72/\pixelsperinch*65536,full/100)/65536)
  return ret
end
```

CSS colors are even more flexible. Not only are there predefined named colors as mentioned above and the special keyword cases `transparent`, `inherit`, `initial`, and `currentcolor`; but colors can also be specified using RGB values, HEX values, HSL values, RGBA values, and HSLA values. The CSS parser has already converted the named colors and resolved inheritance into HEX colors when the actual color interpretation starts, but there is still quite a bit of processing needed.

Here is an example of some of the possibilities borrowed from `w3schools.com`:

```
<h1 style="color:Tomato;">Hello World</h1>
<h1 style="background-color:rgb(255, 99, 71);">...</h1>
<h1 style="background-color:#ff6347;">...</h1>
<h1 style="background-color:rgba(255, 99, 71, 0.5);">...</h1>

<h1 style="background-color:hsl(9, 100%, 64%);">...</h1>
<h1 style="background-color:hsla(9, 100%, 64%, 0.5);">...</h1>
```

HSL values and HSLA values are not supported currently. We have not encountered any HTML generating software yet that actually uses HSL, and as we do not allow the users to key in raw HTML code, we are very unlikely to encounter such definitions. Until we actually need these, they are not worth the hassle.

We *do* support transparency, both the `transparent` keyword and the `rgba()` format. There are two separate Lua functions, `texcolor()` and `mpcolor()` to convert the value into either `\definecolor` / `\directcolored` ConTEXt style (`r=1,b=0.45,g=0.4`), or to a format that our MetaPost macros understand. The latter is used for all arguments to `\framed` and `\definetextbackgrounds`, so it is used much more often.

For MetaPost, HEX values are converted into CSS `rgb()` or `rgba()` syntax. This is the most straightforward way to pass around the color values in the Lua processing code.

But it means using transparency in our MetaPost macros requires a little trick, because the MetaFun macro for transparent colors is based on a different (and more flexible) syntax:

```
def rgb(expr a,b,c) = (a/255,b/255,c/255) enddef;
def rgba(expr a,b,c,d) = (a/255,b/255,c/255,d) enddef;
def checkedcolor(expr a)=
  if cmykcolor a:
    (cyanpart a, magentapart a, yellowpart a) withtransparency(1, blackpart a)
  else:
     a
  fi
enddef;
```

Actual XML processing

There are some two dozen of XML tag processing functions. Some are shorter, some are longer, but most all of them are easy to understand. Showing all of the processing

functions seems overkill, but let's look at some of the more interesting ones in a bit of detail.

Images

```
function xml.functions.img(t)
   local function style(a)  return cssparser.style(t,a) end
   if t.at.width  then cssparser.prependstyle(t, 'width:'..  t.at.width)      end
   if t.at.height then cssparser.prependstyle(t, 'height:' .. t.at.height)    end
   if t.at.align  then cssparser.prependstyle(t, 'text-align:' .. t.at.align) end
   local args = {}
   if style('width')  then args.width  = texdimension(style('width')) .. 'pt'  end
   if style('height') then args.height = texdimension(style('height')) .. 'pt' end
   if not userdata.patchimagesource(t) then
     t.at.src = userdata.settings.externalfigures .. '/' .. t.at.src
   end
   local textalignmap = cssparser.typemaps.text_align
   if style('align') then context.startalign({textalignmap[style('align')]}) end
   if cssparser.inherited(t,'infloat', 0) ~= 1 then context.dontleavehmode() end
   context.externalfigure({t.at.src}, args)
   if style('align') then context.stopalign() end
end
```

Most noteworthy here is the call to `userdata.patchimagesource()`. That function checks the HTML `src` attribute for the existence of inline base64 JPEG or PNG images.

```
<img data-image-id="2590" height="164" src="data:image/png;base64,iVB64..."/>
```

If it finds one of these, it writes the binary data to a disk file and returns `true`. Since those images will always be in the local directory, there is no need to prepend the client's image directory to the `src` value.

The check for the virtual `infloat` trait is there because if the image is not inside of a `\placefigure`, then it should be handled in TEX's horizontal mode. The `\dontleavehmode` forces the start of a paragraph in that case.

Inline font switches

Some of the processing functions make use of dedicated subroutines, like the ones for inline font switches:

```
function xml.functions.b(t) -- also strong
   cssparser.prependstyle(t, 'font-weight:bold;font-style:inherit;')
   context.start()
   context.dontleavehmode()
   handlefontspan(t)
   lxml.flush(t)
   context.stop()
   context("{}")
end
```

The `handlefontspan()` routine takes care of all inline font switches and typical <span> settings like color changes and text decoration options. It is used throughout `dw-workflow` where <span>-style traits need to be set.

Text blocks and structure

Similarly, there is the `textbackgroundarguments()` function (that we saw earlier) to take care of typical block level traits like vertical whitespace, margins, and frames. This makes processing <div> quite simple.

The various <h1..6> tags are simply mapped onto the ConTEXt sectioning commands.

Lists

Itemization lists are a bit problematic because CSS essentially treats every item in a list separately. The net result of that is that ConTEXt needs to start and stop itemization lists regularly, and while that works ok, it is quite suboptimal.

```
function xml.functions.li(t)
   if t.at.type then
      cssparser.prependstyle(t,'list-style-type:' .. t.at.type)
   end
   cssparser.inherit(t,'inlist', true)
   local typemap = cssparser.typemaps.list_style_type
   local itemstyle = cssparser.style(t,'list-style-type')
   if itemstyle then
      context.stopitemize()
      if t.at.value then
         context.setupitemize({start = t.at.value})
      end
      context.startitemize({typemap[itemstyle]})
   elseif t.at.value then
      context.stopitemize()
      context.setupitemize({start = t.at.value})
      context.startitemize({typemap[cssparser.styled(t,'list-style-type')]})
   end
   style  = cssparser.styled(t,'text-align') or 'left'
   context.testpage({'1'})
   context.startitem()
    -- \vadjust to fix vertical spacing for p
   context('\\vadjust{\\kern -\\baselineskip}\\nobreak')
   context.startalign({cssparser.typemaps.text_align[style]})
   lxml.flush(t)
   context.par()
   context.stopalign()
   context.stopitem()
end
```

The `\vadjust` here is particularly ugly, but it is needed because the textual input can have item content both with and without <p> tags.

At some time in the future, we should move to a completely new list model that is closer to how CSS thinks about list items. The CSS model for list items is much closer to plain TEX's way of having separate `\item` commands than to the more structured ConTEXt way of having an enclosing environment. On first sight, you would think that the ConTEXt way is very close to the HTML structure for lists. But on closer examination, CSS allows so many low-level traits on the actual list items that it will probably work better if we switch to something more low-level than the `\startitem` ...`\stopitem` environment. For now, these low-level CSS traits are on the 'unsupported' list.

Tables

Tables are problematic as well. Not so much because of the cell formatting itself (`\bTABLE` generally does a fine job of that), but because all of the possible border styles and spacing variations around those cells.

Individual `\bTD` cells inside a `\bTABLE` are actually disguised `\framed` calls. This is great in that it allows various border and background settings. But `\framed` by itself is not quite powerful enough to do everything that is possible in CSS. As a result of that, quite a large section of `dw-workflow` consists of small extensions to `\framed` and a rather long list of MetaPost graphic definitions.

The normal `\framed` already has four detail values for `frame`: `leftframe=on`, etc. Our version has a similar splits for `margin`, `rulethickness`, and `framecolor`. All of these variables can be set independently. Also, the `..frame` keys like `leftframe` take named versions for all of the CSS border styles instead of just `on` or `off`. The options are: `dotted`, `solid`, `double`, `dashed`, `none`, `hidden`, `groove`, `ridge`, `inset`, and `outset`. And the color settings are a bit different as well: instead of an actual ConTEXt color definition, they take a triplet of `(r,g,b)` or a quarted of `(r,g,b,a)`.

Most of this data is passed on to MetaPost macros that take care of the actual typesetting of the border segments. A little section of that part of `dw-workflow` looks like this:

```
def border_left_dotdash(expr wid, col, w, h, pre, post, dist, dotted,
                             left, top, bottom) =
  if wid>0:
     pickup pencircle scaled wid;
     n := floor((h-pre-post-top-bottom)/(dist));
     if not odd n: n:=n-1; fi
     nw := (h-pre-post-top-bottom)/n ;
     linecap := butt;
     draw ((wid/2+left,post+bottom)--(wid/2+left,h-pre-top))
       dashed dashpattern(if dotted: off nw on nw else: on nw off nw fi)
         withcolor checkedcolor(col);
     fill (wid+left,post+bottom)--(left,post+bottom)--(left,bottom)--cycle
       withcolor checkedcolor(col);
     fill (wid+left,h-pre-top)--(left,h-top)--(left,h-pre-top)--cycle
       withcolor checkedcolor(col);
  fi
enddef;
def border_left_dotted(expr wid, col, w, h, pre, post, left, top, bottom) =
  border_left_dotdash(wid,col,w,h,pre,post,wid,true, left, top, bottom)
enddef;
def border_left_dashed(expr wid, col, w, h, pre, post, left, top, bottom) =
  border_left_dotdash(wid,col,w,h,pre,post,3*wid,false, left, top, bottom)
enddef;
```

The actual connection between `\framed` and these MetaPost definitions is done by

```
\startuseMPgraphic{cellbackground}
   pickup pencircle scaled 0.0001;
   drawdot(0,0) withcolor transparent(1,0,(1,1,1));
   drawdot(\overlaywidth,\overlayheight) withcolor transparent(1,0,(1,1,1));
   border_left_\framedparameter{leftframe}
     (\framedparameter{leftrulethickness},  \framedparameter{leftframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{toprulethickness},\framedparameter{bottomrulethickness},
      \framedparameter{leftmargin}, \framedparameter{topmargin},
      \framedparameter{bottommargin});
   border_right_\framedparameter{rightframe}
     (\framedparameter{rightrulethickness}, \framedparameter{rightframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{bottomrulethickness},\framedparameter{toprulethickness},
      \framedparameter{rightmargin}, \framedparameter{bottommargin},
      \framedparameter{topmargin});
   border_top_\framedparameter{topframe}
     (\framedparameter{toprulethickness},   \framedparameter{topframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{rightrulethickness},\framedparameter{leftrulethickness},
      \framedparameter{topmargin}, \framedparameter{rightmargin},
      \framedparameter{leftmargin});
   border_bottom_\framedparameter{bottomframe}
     (\framedparameter{bottomrulethickness},\framedparameter{bottomframecolor},
      \overlaywidth,\overlayheight,
      \framedparameter{leftrulethickness},\framedparameter{rightrulethickness},
      \framedparameter{bottommargin}, \framedparameter{leftmargin},
      \framedparameter{rightmargin});
\stopuseMPgraphic
```

with the aid of a simple overlay that contains the `cellbackground` graphic, this is used as the background for `\framed`.

In case you are wondering: the two `drawdots` are needed to 'anchor' the graphic inside of the overlay in cases where not all sides are actually drawn.

The hardest part of table processing is support for the CSS property `border-collapse`. Our current version is not quite perfect, but it comes close. Close enough for our

clients. The remaining fault is that when two borders are collapsed into one, the 'winning' border should be placed in the center of the space between the two cells. Our code does not do this recentering. In most tables, this is fine. But the flaw is noticeable in tables where some of the rows (of a single table) have different left and right border widths, or where some of the columns have different top and bottom border widths. Considering the complexity of the collapsed border model, this is a limitation that we can live with. For the moment, at least.

Here is a cleaned up example of the kind of table input we have to process:

```
<table cellspacing="3" style="width:210mm;border-collapse: separate;">
  <tbody>
    <tr>
      <td style="height:60px;border: 6px solid orange;padding: 4px;">
        6px solid orange</td>
      <td style="height:60px;border: 3px solid orange;padding: 4px;">
        3px solid orange</td>
    </tr>
    <tr>
      <td style="height:60px;border: 5px inset green;padding: 4px;">
        inset green</td>
      <td style="height:60px;border: 5px outset green;padding: 4px;">
        outset green</td>
    </tr>
  </tbody>
</table>
```

Figure 2 shows what comes out of our system.

Figure 2. Example output of a table with and without the `border-collapse: collapse` setting.

Summary of the current project state

This project has been in development for about a year now. In that time, we solved all of the acute problems so that we can correctly process the current input to PDF.

To that end, we:

- wrote a (partial) CSS parser
- with the help of the ntg-context mailing list we added ConTEXt support for the CSS `minheight` attribute in `\framed`
- wrote enough code that we are supporting nearly all of the CSS table border features
- can handle in-line images in base64 encoding
- figured out how to support transparent colors in borders.

But that does not mean that we are done. In the future:

- we will have to support more CSS properties as they become requested by our clients
- we should implement a better solution for CSS inheritance than the current brute-force method
- we will probably need to implement an itemization model that is closer to the CSS approach
- and likely more stuff will pop up as we go along.

Taco Hoekwater
DocWolves B.V.