



workflow  
support in context

# Contents

Introduction	2
1 Accessing resources	3
2 Graphics	6
3 Suspects	9
4 Injectors	10
5 XML	12
6 Setups	14

# Introduction

This manual contains some information about features that can help you to manage workflows or CONTEXT related processes. Because we use CONTEXT ourselves all that we need ends up in the distribution. When you discover something workflow related that is not yet covered here, you can tell me. I simply forget about all there is, especially if it's made for projects. Don't expect this manual to be complete or extensive, it's just a goodie.

Hans Hagen,  
PRAGMA ADE  
Hasselt NL  
May 2015

# 1 Accessing resources

One of the benefits of  $\text{\TeX}$  is that you can use it in automated workflows where large quantities of data is involved. A document can consist of several files and normally also includes images. Of course there are styles involved too. At PRAGMA ADE normally put styles and fonts in:

```
/data/site/context/tex/texmf-project/tex/context/user/<project>/...  
/data/site/context/tex/texmf-fonts/data/<foundry>/<collection>/...
```

alongside

```
/data/framework/...
```

where the job management services are put, while we put resources in:

```
/data/resources/...
```

The processing happens in:

```
/data/work/<uuid user space>/
```

Putting styles (and resources like logos and common images) and fonts (if the project has specific ones not present in the distribution) in the  $\text{\TeX}$  tree makes sense because that is where such files are normally searched. Of course you need to keep the distributions file database upto-date after adding files there.

Processing has to happen isolated from other runs so there we use unique locations. The services responsible for running also deal with regular cleanup of these temporary files.

Resources are somewhat special. They can be stable, i.e. change seldom, but more often they are updated or extended periodically (or even daily). We're not talking of a few files here but of thousands. In one project we have 20 thousand resources, that can be combined into arbitrary books, and in another one, each chapter alone is about 400 XML and image files. That means we can have 5000 files per book and as we have at least 20 books, we end up with 100K files. In the first case accessing the resources is easy because there is a well defined structure (under our control) so we know exactly where each file sits in the resource tree. In the 100K case there is a deeper structure which is in itself predictable but because many authors are involved the references to these files are somewhat instable (and undefined). It is surprising to notice that publishers don't care about filenames (read: cannot control all the parties involved) which means that we have inconsistent use of mixed case in filenames, and spaces, underscores and dashes creeping in. Because typesetting for paper is always at the end of the pipeline (which nowadays is mostly driven by (limitations) of web products) we need to have a robust and flexible lookup mechanism. It's a side effect of the click and point culture: if objects are associated (filename in source file with file on the system) anything you key in will work, and consistency completely depends

on the user. And bad things then happen when files are copied, renamed, etc. In that stadium we can better be tolerant than try to get it fixed.<sup>1</sup>

```
foo.jpg
bar/foo.jpg
images/bar/foo.jpg
images/foo.jpg
```

The xml files have names like:

```
b-c.xml
a/b-c.jpg
a/b/b-c.jpg
a/b/c/b-c.jpg
```

So it's sort of a mess, especially if you add arbitrary casing to this. Of course one can argue that a wrong (relative) location is asking for problems, it's less an issue here because each image has a unique name. We could flatten the resource tree but having tens of thousands of files on one directory is asking for problems when you want to manage them.

The typesetting (and related services) run on virtual machines. The three directories:

```
/data/site
/data/resources
/data/work
```

are all mounted as nfs shares on a network storage. For the styles (and binaries) this is no big deal as normally these files are cached, but the resources are another story. Scanning the complete (mounted) resource tree each run is no option so there we use a special mechanism in `CONTEXT` for locating files.

Already early in the development of MKIV one of the locating mechanisms was the following:

```
tree:///data/resources/foo/**/drawing.jpg
tree:///data/resources/foo/**/Drawing.jpg
```

Here the tree is scanned once per run, which is normally quite okay when there are not that many files and when the files reside on the machine itself. For a more high

---

<sup>1</sup> From what we normally receive we often conclude that copy-editing and image production companies don't impose any discipline or probably simply lack the tools and methods to control this. Some of our workflows had checkers and fixers, so that when we got 5000 new resources while only a few needed to be replaced we could filter the right ones. It was not uncommon to find duplicates for thousands of pictures: similar or older variants.

performance approach using network shares we have a different mechanism. This time it looks like this:

```
dirlist:/data/resources/**/drawing.jpg
dirlist:/data/resources/**/Drawing.jpg
dirlist:/data/resources/**/just/some/place/drawing.jpg
dirlist:/data/resources/**/images/drawing.jpg
dirlist:/data/resources/**/images/drawing.jpg?option=fileonly
dirfile:/data/resources/**/images/drawing.jpg
```

The first two lookups are wildcard. If there is a file with that name, it will be found. If there are more, the first hit is used. The second and third examples are more selective. Here the part after the `**` has to match too. So here we can deal with multiple files named `drawing.jpg`. The last two equivalent examples are more tolerant. If no explicit match is found, a lookup happens without being selective. The case of a name is ignored but when found, a name with the right case is used.

You can hook a path into the resolver for source files, for example:

```
\usepath [dirfile://./resources/**]
\setupexternalfigures[directory=dirfile://./resources/**]
```

You need to make sure that `file(name)s` in that location don't override ones in the regular  $\text{T}_{\text{E}}\text{X}$  tree. These extra paths are only used for source file lookups so for instance font lookups are not affected.

When you add, remove or move files the tree, you need to remove the `dirlist.*` files in the root because these are used for locating files. A new file will be generated automatically. Don't forget this!

## 2 Graphics

### 2.1 Bad names

After many years of using CONTeXT in workflows where large amounts of source files as well as graphics were involved we can safely say that it's hard for publishers to control the way these are named. This is probably due to the fact that in a click-and-point based desktop publishing workflow names don't matter as one stays on one machine, and names are only entered once (after that these names become abstractions and get cut and pasted). Proper consistent resource management is simply not part of the flow.

This means that you get names like:

```
foo_Bar_01_03-a.EPS
foo__Bar-01a_03.eps
foo__Bar-01a_03.eps
foo BarA  01-03.eps
```

Especially when a non proportional screen font is used multiple spaces can look like one. In fancy screen fonts upper and lowercase usage might get obscured. It really makes one wonder if copy-editing or adding labels to graphics isn't suffering from the same problem.

Anyhow, as in an automated rendering workflow the rendering is often the last step you can imagine that when names get messed up it's that last step that gets blamed. It's not that hard to sanitize names of files on disk as well as in the files that refer to them, and we normally do that we have complete control. This is no option when all the resources are synchronized from elsewhere. In that case the only way out is signaling potential issues. Say that in the source file there is a reference:

```
foo_Bar_01_03-a.EPS
```

and that the graphic on disk has the same name, but for some reason after an update has become:

```
foo-Bar_01_03-a.EPS
```

The old image is probably still there so the update is not reflected in the final product. This is not that uncommon when you deal with tens of thousands of files, many editors and graphic designers, and no strict filename policy.

For this we provide the following tracing option:

```
\enabletrackers[graphics.lognames]
```

This will put information in the log file about included graphics, like:

```
system          > graphics > start names
```

```

used graphic      > asked      : cow.pdf
used graphic      > comment    : not found
used graphic      > asked      : t:/sources/cow.pdf
used graphic      > format     : pdf
used graphic      > found      : t:/sources/cow.pdf
used graphic      > used       : t:/sources/cow.pdf

system           > graphics > stop names

```

You can also add information to the file itself:

```
\usemodule[s-figures-names]
```

Of course that has to be done at the end of the document. Bad names are reported and suitable action can be taken.

## 2.2 Downsampling

You can plug in you rown converter, here is an example:

```

\startluacode

figures.converters.jpg = figures.converters.jpg or { }

figures.converters.jpg["lowresjpg.pdf"] =
  function(oldname,newname,resolution)
    figures.programs.run (
      [[gm]],
      [[convert -geometry %nx%x%ny% -compress JPEG "%old%" "%new%"]],
      {
        old = old,
        new = new,
        nx  = resolution or 300,
        ny  = resolution or 300,
      }
    )
  end
\stopluacode

```

You can limit the search to a few types and set the resolution with:

```

\setupexternalfigures
[order={pdf,jpg},
resolution=100,
method=auto]

```



And use it like:

```
\externalfigure[verybig.jpg] [height=10cm]
```

The second string passed to the `run` helper contains the arguments to the first one. The variables between percent signs get replaced by the variables in the tables passed as third argument.

## 2.3 Trackers

If you want a lot of info you can say:

```
\enabletrackers[figures.*]
```

But you can be more specific. With `graphics.locating` you will get some insight in where files are looked for. The `graphics.inclusion` tracker gives some more info about actual inclusion. The `graphics.bases` is kind of special and only makes sense when you use the graphic database options. The `graphics.conversion` and related tracker `graphics.programs` show if and how conversion of images takes place.

The `graphics.lognames` will make sure that some extra information about used graphics is saved in the log file, while `graphics.usage` will produce a file `<jobname>-figures-usage.lua` that contains information about found (or not found) images and the way they are used.





















### 3 Suspects

When many authors and editors are involved there is the danger of inconsistent spacing being applied. We're not only talking of the (often invisible in editing programs) nobreak spaces, but also spacing inside or outside quotations and before and after punctuation or around math.

In `CONTEXT` we have a built-in suspects checker that you can enable with the following command:

```
\enabletrackers[typesetters.suspects]
```

The next table shows some identified suspects.

<code>foo\$x\$</code>	<code>foo</code>  <code>x</code>
<code>\$x\$bar</code>	<code>x</code>  <code>bar</code>
<code>foo\$x\$bar</code>	<code>foo</code>  <code>x</code>  <code>bar</code>
<code>\$f+o+o\$:</code>	<code>f + o + o</code> 
<code>;\$f+o+o\$</code>	 <code>f + o + o</code>
<code>; bar</code>	<code>; bar</code>
<code>foo:bar</code>	<code>foo</code>  <code>bar</code>
<code>\quote{ foo }</code>	 <code>foo</code> 
<code>\quote{bar }</code>	<code>'bar</code> 
<code>\quote{ bar}</code>	 <code>bar'</code>
<code>(foo )</code>	<code>(foo</code> 
<code>\{foo \}</code>	<code>{foo</code> 
<code>foo{\bf gnu}bar</code>	<code>foo</code>  <code>gnu</code>  <code>bar</code>
<code>foo{\it gnu}bar</code>	<code>foo</code>  <code>gnu</code>  <code>bar</code>
<code>foo\$x^2\$bar</code>	<code>foo</code>  <code>x</code> <sup>2</sup>  <code>bar</code>
<code>foo\nobreakspace bar</code>	<code>foo</code>  <code>bar</code>

Of course this analysis is not perfect but we use it in final checking of complex documents that are generated automatically from XML.<sup>2</sup>

---

<sup>2</sup> Think of math school books where each book is assembled from over a thousands files.

## 4 Injectors

When you have no control over the source but need to manually tweak some aspects of the typesetting, like an occasional page break or column switch, you can use the injector mechanism. This mechanism is part of list and register building but can also be used elsewhere.

We have two buffers:

```
\startmixedcolumns[balance=yes]
  \dotestinjector{test}line 1 \par
  \dotestinjector{test}line 2 \par
  \dotestinjector{test}line 3 \par
  \dotestinjector{test}line 4 \par
  \dotestinjector{test}line 5
\stopmixedcolumns
```

and

```
\startmixedcolumns[balance=yes]
  \dotestinjector{test}line 1 \par
  \dotestinjector{test}line 2 \par
  \dotestinjector{test}line 3 \par
  \dotestinjector{test}line 4 \par
  \dotestinjector{test}line 5
\stopmixedcolumns
```

When typeset these come out as:

<b>line 1</b>	<b>line 4</b>
<b>line 2</b>	<b>line 5</b>
<b>line 3</b>	

and

<b>line 1</b>	<b>line 4</b>
<b>line 2</b>	<b>line 5</b>
<b>line 3</b>	

We can enable (and show) the injectors with:

```
\doactivateinjector{test} \showinjector
```

Now we get:

<1> <b>line 1</b>	<4> <b>line 4</b>
<2> <b>line 2</b>	<5> <b>line 5</b>
<3> <b>line 3</b>	

and

<6> <b>line 1</b>	<9> <b>line 4</b>
<7> <b>line 2</b>	<10> <b>line 5</b>
<8> <b>line 3</b>	

The small numbers are injector points. These will of course change when we add more in-between. Let's add actions to some of the injection points:

```
\setinjector[test][13][{\column}]
\setinjector[test][17][{\column}]
```

As expected we now get column breaks:

<11> <b>line 1</b>	<13> <b>line 3</b>
<12> <b>line 2</b>	<14> <b>line 4</b>
	<15> <b>line 5</b>

and

<16> <b>line 1</b>	<17> <b>line 2</b>
	<18> <b>line 3</b>
	<19> <b>line 4</b>
	<20> <b>line 5</b>

## 5 XML

When you have an XML project with many files involved, finding the right spot of something that went wrong can be a pain. In one of our project the production of some 50 books involves 60.000 XML files and 20.000 images. Say that we have the following file:

```
<?xml version='1.0'?>
<document>
  <include name="temp-01.xml"/> <include name="temp-02.xml"/>
  <include name="temp-03.xml"/> <include name="temp-04.xml"/>
  <include name="temp-05.xml"/> <include name="temp-06.xml"/>
  <include name="temp-07.xml"/> <include name="temp-08.xml"/>
  <include name="temp-09.xml"/> <include name="temp-10.xml"/>
</document>
```

Before we process this file we will merge the content of the files defined as includes into it. When this happens the filename is automatically registered so it can be accessed later.

```
\startxmlsetups xml:initialize
  \xmlincludeoptions{#1}{include}{filename|name}{recurse,basename}
  \xmlsetsetup{#1}{p|document}{xml:*}
\stopxmlsetups

\startxmlsetups xml:document
  \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:p
  \inleftmargin{\infofont\xmlinclusion{#1}}
  \xmlflush{#1}
  \par
\stopxmlsetups

\xmlregistersetup{xml:initialize}

\xmlprocessbuffer{main}{demo}{{}}
```

In this case we put the name of the file in the margin. Depending on when and how elements are flushed other solutions, like overlays, can be used.

temp-01.xml	snippet 1
temp-02.xml	snippet 2
temp-03.xml	snippet 3
temp-04.xml	snippet 4
temp-05.xml	snippet 5

```
temp-06.xml snippet 6
temp-07.xml snippet 7
temp-08.xml snippet 8
temp-09.xml snippet 9
temp-10.xml snippet 10
```

At any moment you can see what the current node contains. The whole (merged) document is also available:

```
\xmlshow{main}
```

A small font is used to typeset the (sub)tree:

```
<?xml version='1.0'?>
<document>
  <p>snippet 1</p> <p>snippet 2</p>
  <p>snippet 3</p> <p>snippet 4</p>
  <p>snippet 5</p> <p>snippet 6</p>
  <p>snippet 7</p> <p>snippet 8</p>
  <p>snippet 9</p> <p>snippet 10</p>
</document>
```

You can also save the tree:

```
\xmlsave{main}{temp.xml}
```

This file looks like:

```
<?xml version='1.0'?>
<document>
  <p>snippet 1</p> <p>snippet 2</p>
  <p>snippet 3</p> <p>snippet 4</p>
  <p>snippet 5</p> <p>snippet 6</p>
  <p>snippet 7</p> <p>snippet 8</p>
  <p>snippet 9</p> <p>snippet 10</p>
</document>
```

## 6 Setups

Setups are a powerful way to organize styles. They are basically macros but live in their own namespace. One advantage is that spaces in a setup are ignored so you can code without bothering about spurious spaces. Here is a trick that you can use when one style contains directives for multiple products:

```
\startsetups tex:whatever
  \fastsetup{tex:whatever:\documentvariable{stylevariant}}
\stopsetups
```

```
\startsetups tex:whatever:foo
  FOO
\stopsetups
```

```
\startsetups tex:whatever:bar
  BAR
\stopsetups
```

Here we define a main setup `tex:whatever` that gets expanded in one of two variants, controlled by a document variable.

```
\setups{tex:whatever}
```

```
\setupdocument
  [stylevariant=foo]
```

```
\setups{tex:whatever}
```

```
\setupdocument
  [stylevariant=bar]
```

```
\setups{tex:whatever}
```

These lines result in:

FOO

BAR

In a similar fashion you can define XML setups that are used to render elements:

```
\startxmlsetups xml:whatever
  \xmlsetup{#1}{xml:whatever:\documentvariable{stylevariant}}
\stopxmlsetups
```

```
\startxmlsetups xml:whatever:foo
  FOO: \xmlflush{#1}
```

```
\stopxmlsetups
```

```
\startxmlsetups xml:whatever:bar
```

```
  BAR: \xmlflush{#1}
```

```
\stopxmlsetups
```