Smart playlists for the Music Player Demon (MPD)

Johannes Willkomm

June 12, 2015

Contents

1 Generating smart playlists for MPD with XSLT

Code generation with XSLT is after all the years a hot topic for me. I think it is a great technique to generate code for many means. Let's see an example for the code generation with XSLT.

The goal is to create a feature I had been using with XMBC: smart playlists. The idea is basically that you enter a static query which is saved with a name and when you load the smart playlist the query is evaluated against your database. This is of course great if you regularly add music to youy library. To me especially because I use to record from internet radios and thus end up with MP3 collections which are quite unwieldy. Especially annoying are the many spelling variations in the names and titles. This just demands some fuzzy flexible treatment with smart playlists.

The idea is to generate Bash shell code that updates the smart playlists from a problem definition. The first thing we need is such a problem definition, in XML of course. This is in the file `smartplaylists.xml`:

```
<smart-playlists version="1.0">
  <playlist name="el-rookie">
    <or>
      <and>
        <query type="artist">rookie</query>
      </and>
      <and>
        <query type="artist">roockie</query>
      </and>
    </or>
  </playlist>
  <playlist name="wisin-y-yandel">
    <or>
      <and>
        <query type="artist">wisin</query>
        <query type="artist">yandel</query>
      </and>
    </or>
  </playlist>
</smart-playlists>
```

As you can see the XML is not as simple as you might expect. This is because I developed the XML problem definition and the XSLT in tandem, such that one suits the other. Let's see how the XSLT (file `genupdate-sh.xsl`) looks:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text"/>
  <xsl:template match="text()"/>

  <xsl:template match="/">
    mpc rm tmp-update-pls
    mpc save tmp-update-pls
    <xsl:apply-templates/>
    mpc clear
    mpc load tmp-update-pls
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>

  <xsl:template match="playlist">
    mpc clear
    <xsl:apply-templates/>
    mpc rm <xsl:value-of select="@name"/>
    mpc save <xsl:value-of select="@name"/>
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>

  <xsl:template match="or">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="and">
    mpc search <xsl:apply-templates/> | mpc add
  </xsl:template>

  <xsl:template match="query">
    <xsl:text> </xsl:text>
    <xsl:value-of select="@type"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Let's step through the code to see what all the things do. The first lines are the XML declaration and the open tag of the top-level **stylesheet** element. Then in the **output** element we set the method to `text` since we whan to generate Bash source code.

Then next important thing is to provide a do-nothing template for the **text()** nodes, because we will have to tightly control the output, since newline characters matter in Bash shell code (they terminate a command).

Then comes the template for the root node `/` which is executed first. There we enter the commands that should be run first and last. We back up the current playlist to `tmp-update-pls` and restore it in the end. In between we invoke **apply-templates** which will generate the code we actually need.

In the template matching the **playlist** element we generate the code that creates a playlist according to the query and save it under the name given by attribute **@name**. As you see, there is again some stuff we have to do first and some other which we do in the end, and in between we invoke **apply-templates**. First we clear the current playlist. Then comes **apply-templates**, which we assume to compose the playlist. Then we can save it.

The template matching element **or** simply invokes **apply-templates**, because for each child some command is to be generated.

The template matching element **and** will simply invoke generate a `mpc add` command, which receives is fed the output from a `mpc search` command. Each child shall produce a tuple `<type> <query>`, which are intersected by `mpc add`, so we invokes **apply-templates** again.

Finally, the template matching element **query** will output the query type as its attribute **@type** and the query as its text content.

Now we want to see the XSLT in action: We run **xsltproc** with it on the smart playlist definition XML:

```
xsltproc genupdate-sh.xsl smartplaylists.xml
```

```

mpc rm tmp-update-pls
mpc save tmp-update-pls

mpc clear

mpc search  artist rookie | mpc add

mpc search  artist roockie | mpc add

mpc rm el-rookie
mpc save el-rookie

mpc clear

mpc search  artist wisin artist yandel | mpc add

mpc rm wisin-y-yandel
mpc save wisin-y-yandel
mpc clear
mpc load tmp-update-pls
```

The result is valid Bash code, so we can just pipe it to the shell:

```
xsltproc genupdate-sh.xsl smartplaylists.xml | bash
```

Then load one of the freshly generated and up-to-date playlists:

```
mpc load wisin-y-yandel
```

A few things to note:

- Running this playlist update command will save and restore your current playlist, but it will stop playback
- you might want to run `mpc update` before to update the database
- Also, other than with XBMC smart playlists, ours will not be updated automatically when you load them. You could run the command from your crontab, of course
- as I said before, the XML structure the XSLT were designed *together*, I hope you can got a little feeling for how the particular document structure enabled the simple XSLT implementation
- in XSLT the code '`<xsl:text>&#xa;</xsl:text>`' generates a newline (you can also use '`<xsl:text>&#x10;</xsl:text>`'). I prefer this form because it is robust against auto indenting
- the element **or** is optional and just facilitates human reading
- generally I think it is very important that you understand how XSLT interprets whitespace

The last two points deserve more words, but for now, I'd say, just play around with it and see what happens if you change something.

Happy code generating!