JavaSource to Pandoc Converter Documentation

Michael Mosmann

May 8, 2013

Intro

This is an experiment. We want to generate a documentation from source code, so that in the best case the code in the documentation is fresh and tested. Maybe this can simplify the process of code documentation, because you will have your documentation and your code in the same source file.

The only thing you have to do is to mark snippets in your source file which should be rendered to the final documentation. It does not matter if you mark a comment or a code. We will be able to detect the difference.

To mark your snippet, you have to write something like this in your JavaDoc comment to mark the start:

```
* -->
```

and

```
* <--
```

to mark the end.

It is not much different for your code or line comments. This for the start:

```
// -->
```

and this

```
// <--
```

for the end.

JavaDoc Tags and Hints

To make it easy to write markdown style text in your JavaDoc we will you provide with some tipps.

Supported Tags

Following JavaDoc Tags are supported:

- pre
- code

If you want some generic markdown code block, you can wrap your text with <pre>-Tags. To make this a code block like this

```
a code block
```

you have to write something like this in your JavaDoc comment:

```
<pre>
  a code block
</pre>
```

Escaping

If you want to use Tags in your text you have to escape the tag this way:

```
... this is the \<tag\> i want to used ...
```

Lists

List are potential target for code format issues. You have to leave a empty line around a list for valid markdown markup. You can write markdown lists, but if you reformat the code they will be gone. List Tags are not supported now.

- Summer
- Winter

Headers

There are two kinds of headers, Setext and atx.

Setext-style headers

A setext-style header is a line of text "underlined" with a row of = signs (for a level one header) of - signs (for a level two header):

```
A level-one header
==================



A level-two header
------------------
```

The header text can contain inline formatting, such as emphasis (see Inline formatting, below).

Atx-style headers

An Atx-style header consists of one to six # signs and a line of text, optionally followed by any number of # signs. The number of # signs at the beginning of the line is the header level:

```
## A level-two header



### A level-three header ###
```

As with setext-style headers, the header text can contain formatting:

```
# A level-one header with a [link](/url) and *emphasis*
```

Standard markdown syntax does not require a blank line before a header. Pandoc does require this (except, of course, at the beginning of the document). The reason for the requirement is that it is all too easy for a # to end up at the beginning of a line by accident (perhaps through line wrapping). Consider, for example:

```
I like several of their flavors of ice cream:
#22, for example, and #5.
```

Sample of Test Code in Documentation

You can set the mark to the beginning of the code block which should show up in your documentation. You need the second mark on the end of the block.

```
int a=2;
// -->
String s="This is the code which will show up";
// <--
s="and this not";
```

Test Code for a simple Comment

```
ILineMatcher startMatcher = JavaSourceLineMatcher.startMatcher();
ILineMatcher endMatcher = JavaSourceLineMatcher.endMatcher();
BlockToBlockListConverter converter = new BlockToBlockListConverter(startMatcher, endMatcher

List<String> lines = Lists.newArrayList();

lines.add(" // -->");
lines.add(" // shift one left");
lines.add(" // this line too");
lines.add(" // <--");

List<Block> blocks = converter.convert(new Block(lines));

assertEquals(1, blocks.size());
Block block = blocks.get(0);
assertNotNull(block);
assertEquals("// shift one left", block.lines().get(0));
assertEquals("// this line too", block.lines().get(1));
```

Test Code for an embedded Marker

```
ILineMatcher startMatcher = JavaSourceLineMatcher.startMatcher();
ILineMatcher endMatcher = JavaSourceLineMatcher.endMatcher();
BlockToBlockListConverter converter = new BlockToBlockListConverter(startMatcher, endMatcher

List<String> lines = Lists.newArrayList();

lines.add(" // -->");
lines.add(" // shift one left");
lines.add("     // <--");
lines.add("     // -->");
```

```
lines.add("     // shift one left (embedded)");
lines.add("     // this line too (embedded)");
lines.add("     // <--");
lines.add("     // -->");
lines.add(" // this line too");
lines.add(" // <--");

List<Block> blocks = converter.convert(new Block(lines));

assertEquals(1, blocks.size());
Block block = blocks.get(0);
assertNotNull(block);
assertEquals("// shift one left", block.lines().get(0));
assertEquals("  // shift one left (embedded)", block.lines().get(1));
assertEquals("  // this line too (embedded)", block.lines().get(2));
assertEquals("// this line too", block.lines().get(3));
```