CHAPTER 6

Easy Text Markup with the BlueCloth Gem

BlueCloth is an implementation of Markdown (http://daringfireball.net/projects/markdown/), a text-to-HTML converter. It features a simple syntax, and is easy to understand visually. It allows you to write documents and posts in an easy, standard format; additionally, since it's easy to learn, even nontechnical users become comfortable very quickly.

BlueCloth is a very simplified markup language, so not all features of HTML are available; it's very similar to a message-board markup language such as BBCode. BlueCloth isn't designed to replace HTML, but rather to make writing and editing text documents much easier—blog posts, message-board posts, articles, and so forth. As a result, BlueCloth documents can be easily read and written—and even if you aren't familiar with BlueCloth, it's reasonably easy to understand what's going on.

How Does It Work?

BlueCloth's syntax is reasonably simple. It's a plain text format, and additional formatting, like headers, lists, and so forth, all have a simple markup. A few symbols, like number signs, dashes, and so forth, can control your output—as a bonus, they also look very understandable when the document is in text form, in contrast to HTML. Let's examine a few types of BlueCloth syntax.

A header is started with one or more hash marks—the more hash marks the higher the header level, so a single hash mark means an <h1> tag, two means an <h2>, and so forth. For example, the following BlueCloth will produce an <h1> and an <h2>:

```
# I Always Wanted to Learn Lisp

## I Never Wanted to Learn Cobol
```

```
<h1>I Always Wanted to Learn Lisp</h1>

<h2>I Never Wanted to Learn Cobol</h2>
```

Paragraphs are separated by a blank line. Some HTML-to-text converters turn all new lines into line breaks; this means that if a paragraph is wrapped manually, the browser can't rewrap it automatically. `BlueCloth` is flexible—it will handle paragraphs that are one long line or paragraphs that have multiple hand-wrapped lines.

You can do a number of other things in `BlueCloth`, of course; for example, code blocks are produced by indenting every line with spacing—at least four spaces or a tab. (This means you can't have a paragraph of noncode text starting with a tab or four spaces, of course.) Consider an example:

```
Some code:

    9.times do
      puts "test"
    end
```

```
<p>Some code:</p>

<pre><code>9.times do
  puts "test"
end
</code></pre>
```

You can get more details at the Markdown homepage, `http://daringfireball.net/projects/markdown/`.

`BlueCloth` has a number of other options, for block quotes, tables, and more. You can get more details on `BlueCloth` syntax at the `BlueCloth` homepage, `http://www.deveiate.org/projects/BlueCloth`.

To install `BlueCloth`, use the `gem install` command:

```
gem install bluecloth
```

BlueCloth-to-HTML Converter

To demonstrate `BlueCloth`, let's put together a tiny utility that will convert from Markdown to HTML (see Listing 6-1).

Listing 6-1. *bluecloth2html.rb*

```
require 'bluecloth'

puts BlueCloth.new( ARGF.read ).to_html
```

`ARGF` is a special variable Ruby provides for our use. It give stream methods, like `read`, to any files passed on the command line. If no files are passed, then it will read from standard input. A new BlueCloth object is created from the input, and then its `to_html` method is called —which, of course, converts it to HTML. The HTML is then printed to the screen using `puts`.

Consider Listing 6-2; the text is placed n a file named `test.txt`.

Listing 6-2. *test.txt*

```
#Why you should use Ruby.

Ruby is an open source, powerful programming language. It's a scripting language,
much like Perl or Python. However, it's surprisingly elegant; complex techniques
can be implemented in just a few lines of code. It's also harmonious, in that
things—even complex things—work the way you might expect them to, even when
used in surprising ways.

For example, Ruby handles iterators in an interesting way. It defines a simple yet
powerful way to deal with arbitrary blocks of code.
The following code prints out 1 through 10:

    1.upto(10){ |x|
      print x
    }
```

We could convert it like this:

```
ruby bluecloth2html.rb test.txt
```

The output would be this:

```
<h1>Why you should use Ruby.</h1>

<p>Ruby is an open source, powerful programming language. It's a scripting
language, much like Perl or Python. However, it's surprisingly elegant; complex
techniques can be implemented in just a few lines of code. It's also
harmonious, in that things—even complex things—work the way you might expect them
to, even when used in surprising ways.</p>

<p>For example, Ruby handles iterators in an interesting way. It defines a simple
yet powerful way to deal with arbitrary blocks of code.
The following code prints out 1 through 10:</p>

<pre><code>1.upto(10){ |x|
  print x
}
</code></pre>
```

As you can see, the converter produces standard HTML. Headers become `<h1>` and `<h2>` tags, ordered lists `<ol>` and `<li>` tags, and so forth. Code blocks use both `<pre>` and `<code>` tags—many Web programmers would use only the `<pre>` tags, but the `<code>` tag is an additional indicator to the browser that it's not just any preformatted text, but specifically preformatted code. (Incidentally, `BlueCloth` comes with a utility to convert `BlueCloth` to HTML; it works similar to the example in Listing 6-1.)

bluecloth2pdf BlueCloth-to-PDF Converter

In this section you'll learn how to convert a file marked up with `BlueCloth` syntax to PDF. This example script is called with two arguments—the input `BlueCloth` file and the output PDF. If you pass a dash as the input file, it will read from `STDIN` instead. Figure 6-1 has an example of what our converter looks like when run on the text in Listing 6-2.

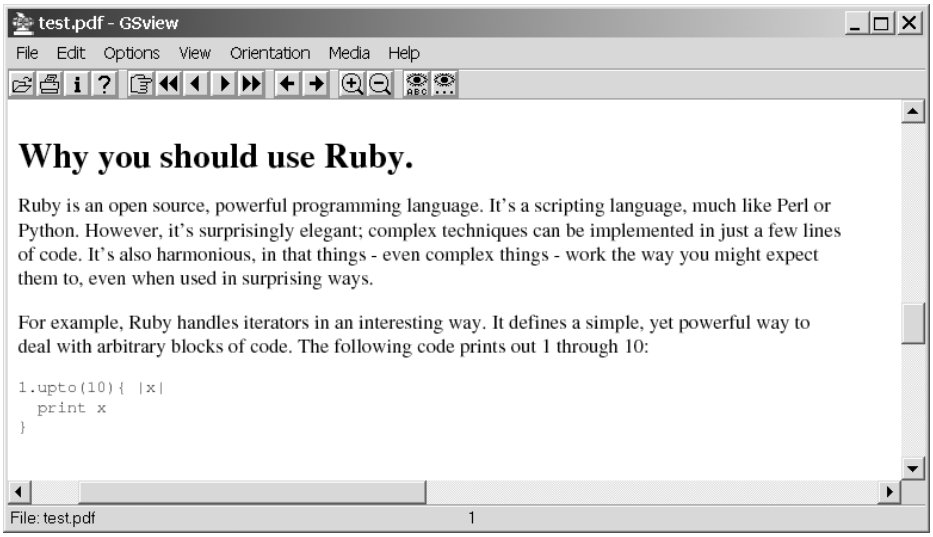

Figure 6-1. *The BlueCloth-to-PDF converter*

To run this example, you will need `html2ps` and `ghostscript` installed. Many Linux distributions come with both; for other platforms you can download them from the following URLs, respectively:

```
http://user.it.uu.se/~jan/html2ps.html
http://www.cs.wisc.edu/~ghost/
```

Regardless of platform, you'll need to fill in your full path to each utility in the script, as shown in Listing 6-3.

Listing 6-3. *bluecloth2pdf.rb*

```
require 'tempfile'  # Ruby builtin library - no need to install.
require 'bluecloth'

html2ps_command='/path/to/html2ps'
#on win32, you'll need to prefix this with 'perl '
#   - win32 won't know to run the perl interpreter on it.

ps2pdf_command='/path/to/ghostscript/lib/ps2pdf'
#ps2pdf comes with ghostscript - it'll be in your ghostscript/lib directory.
```

```
if ARGV[0]=='-'
  input_string = $stdin.read
else
  input_string = File.read(ARGV[0])
end
output_pdf_filename =ARGV[1]

# Convert our BlueCloth input into HTML output

bc = BlueCloth::new( input_string )
html_string = bc.to_html

tmp_html_filename ="#{Dir::tmpdir}/#{$$}.html"
tmp_ps_filename = "#{Dir::tmpdir}/#{$$}.ps"

# Next, we take our BlueCloth input and turn it
# into a full HTML document, not just a fragment.

File.open(tmp_html_filename, 'w') do |f|
  f << "<html><head><title>bluecloth2pdf</title></head>"
  f << "<body>"
  f << html_string
  f << "</body>"
  f << "</html>"
end

# First, we convert the HTML and convert it into postscript using
# html2ps, and then convert it into a PDF document using ps2pdf.

`#{html2ps_command} < "#{tmp_html_filename}" > "#{tmp_ps_filename}"`
`#{ps2pdf_command} "#{tmp_ps_filename}" "#{output_pdf_filename}"`
```

Assuming you still have the `test.txt` file from Listing 6-2, you can test your creation with the following command:

```
ruby bluecloth2pdf.rb test.txt test.pdf
```

This will take the `test.txt` file and convert it into a PDF. If all goes well, you can verify it by opening the `test.pdf` file in Ghostview or Adobe Acrobat. If not, check that you've installed both `html2ps` and `ghostscript`. Check your paths; it's easy to make a mistake.

Let's briefly recap this relatively complicated script: we took a markdown input, turned it into HTML using the `BlueCloth` gem, converted that into PostScript using `html2ps`, and then finally converted the PostScript into PDF using `ps2pdf`. It's a fairly convoluted process, but the result is transparent to the user and the final PDF is very slick.

There are other options for each step, of course; you could modify the `BlueCloth` gem to use the `pdfwriter` gem as output, although that would be fairly complicated. Depending on your circumstance, you might wish to use an alternate HTML conversion tool, such as `HTMLDOC`. You can get more information on `HTMLDOC` at `http://www.htmldoc.org/`.

■ **Tip** You also could use Ghostscript to output to formats other than PDF: you could output to PNG or JPEG snapshots. For example, we could replace the last line of the script with this:

```
`gs -sDEVICE=jpeg -sOutputFile="#{output_pdf_filename}" "#{tmp_ps_filename}"`
```

That will print to a JPEG filename—just like the first version printed to a PDF. (You should probably refactor the `output_pdf_filename` variable to a different name, of course, since you won't be outputting to PDF files.)

Additionally, you could use Ghostscript's printer drivers to create a hard copy; you can get the full details on Ghostscript printer drivers—as well as other Ghostscript output devices—at `http://www.cs.wisc.edu/~ghost/doc/cvs/Devices.htm`.

Dissecting the Example

Let's take a look at a few important lines from Listing 6-3. The first few lines set up the paths to the two programs we'll use in this script: `html2ps` and `ps2pdf`. The next few lines parse the input—if the input argument is a dash, then parse from the standard in or `STDIN`—typically the keyboard. Otherwise, read from the file provided. Next we hard-code the paths to the various programs we will use later:

```
html2ps_command='/path/to/html2ps'

ps2pdf_command='/path/to/ghostscript/lib/ps2pdf'
```

Next we create a new `BlueCloth` object from the input and turn it into HTML using the `to_html` method:

```
bc = BlueCloth::new( input_string )
html_string = bc.to_html
```

Next we create two temporary filenames: one for the HTML input to `html2ps`, and one for the intermediate PostScript file. It uses the `Dir::tmpdir` variable to locate the temporary directory for your operating system; to access this variable, we must have the `require tempfile` statement, even though we aren't using the `tempfile` class.

```
tmp_html_filename ="#{Dir::tmpdir}/#{$$}.html"
tmp_ps_filename = "#{Dir::tmpdir}/#{$$}.ps"
```

The HTML temporary file is filled with the output from `BlueCloth` surrounded by the the skeleton of an HTML document, as follows:

```
File.open(tmp_html_filename, 'w') do |f|
  f << "<html><head><title>bluecloth2pdf</title></head>"
  f << "<body>"
  f << html_string
  f << "</body>"
  f << "</html>"
end
```

`html2ps` needs the `<body>` and `<html>` tags, but since `BlueCloth` produces only HTML fragments and not complete documents, we need to add them ourselves. Note that this is a good thing: it means you can use `BlueCloth` to represent blog posts, for example, and put the output HTML inside a larger document.

Finally, we have two backtick commands:

```
`#{html2ps_command} < "#{tmp_html_filename}" > "#{tmp_ps_filename}"`
`#{ps2pdf_command} "#{tmp_ps_filename}" "#{output_pdf_filename}"`
```

These run system commands; specifically, the first runs `html2ps`, connecting its input to the temporary HTML file and its output to the temporary PostScript flie. The second command calls `ps2pdf`, but unlike `html2ps` it takes input and output files as arguments, so we don't need the redirection. Its input is the temporary PostScript file that `html2ps` produced; its output is our output PDF file. Once the second command is run, our program is finished.

Conclusion

BlueCloth is a powerful, easy-to-use way to write and edit documents, and it'd fit in well in a variety of environments, including online content-management systems, blogs, forums, software changelogs, and much more.