Manual Writer's Guide Documentation

Release 4.3

Kestrel Institute

September 30, 2013

1 Using the documentation tools **3**
1.1 Prerequisites . . . 3
1.2 Building Documentation . . . 3

2 Writing Documentation with reStructuredText **5**
2.1 Sectioning Commands . . . 5
2.2 Inline Formatting . . . 6
2.3 Code Blocks . . . 6
2.4 Lists . . . 7
2.5 Shell commands . . . 8
2.6 Index Entries . . . 8
2.7 BNF Grammar Descriptions . . . 8

Index **11**

Contents:

CHAPTER

ONE

USING THE DOCUMENTATION TOOLS

1.1 Prerequisites

To build the manuals, you need to have python and its "Sphinx Python Document Generator" package. Assuming that you've installed a recent version of python that includes the `setuptools` package (which it probably does), you can install `sphinx` with:

```
sudo easy_install sphinx
```

Do not use Macports or Homebrew to install this, because they will find the "Sphinx Open Source Search Server", a different tool.

In addition, you need the `pdflatex` program. Generally this is a symlink to `pdftex`. On the Mac, you can get this from http://www.tug.org/mactex/

1.2 Building Documentation

Each manual is in its own directory. To build a particular manual, you change to the directory associated with the manual, then use the included `Makefile`, which in turn invokes the `sphinx` tools. There are a number of output formats supported, as well as a few utilities. To view a list of all targets, simply invoke `make` without an argument:

```
> make
```

There are a few targets of particular interest.

`html` Build the `html` version of the documentation.

`latexpdf` Generate latex, then build the resulting latex with `pdflatex`.

`epub` Generate an EPUB version of the manual.

`linkcheck` Run the tools on the input files to check for correctness in links.

In each case, the makefile will generate output in a subdirectory of a top-level `_build` directory. The subdirectory name will be associated with the makefile target (e.g. `html` for the `html` target, `latex` for the `pdflatex` target).

The directory contains two important files that dictate the structure of the manual. First, `index.rst` defines the files that will be included in the manual. Second `conf.py` defines configuration variables for the documentation. Editing the `index.rst` file is described in *Writing Documentation with reStructuredText*, while documentation for editing the `conf.py` file is available on the Sphinx website.

CHAPTER

TWO

WRITING DOCUMENTATION WITH RESTRUCTUREDTEXT

2.1 Sectioning Commands

RestructuredText uses ASCII formatting to delineate chapters/sections/subsections, etc. The format is quite flexible, and doesn't dictate what format is used to identify a section level, but the Specware Manuals use a standard convention.

Chapters The beginning of chapters is denoted by writing the chapter title with a line of == immediately above and below the chapter title:

```
======================
This is a Chapter Title
======================
```

Sections The beginning of sections is denoted by writing the section title with a line of ## immediately below the section title. The output includes a number preceding the title like **NN.NN**. For example:

```
This is a Section Title
#######################
```

Subsections The beginning of subsections is denoted by writing the section title with a line of ===== immediately below the section title. The output includes a number preceding the title like **NN.NN.NN**. For example:

```
This is a Subsection Title
==========================
```

Subsubsections The beginning of subsubsections is denoted by writing the section title with a line of hyphens (-----) immediately below the section title. The subsubsection name output is larger and bolder than the following text but it has no preceding number. For example:

```
This is a Subsubsection Title
-----------------------------
```

2.1.1 References

To refer to a section, simply put the title of the section in single-backtick quotes, followed by an underscore. This will create a link to the section. For example:

```
`Sectioning Commands`_
```

Creates a link to (this) Sectioning Commands section.

That form works only within a single `.rst` file. To allow cross-references between rst files within a single document, you can give an explicit label preceding the chapter, section, or subsection title:

```
.. _Sec Label:

-----------------
This is a section
-----------------
```

Then, when you want to refer to this label, just write `:ref:`Sec Label``.

To refer to an external link, use the same syntax for the reference. To define the target of the reference, use the same text use when referring to the link, but *preceeded* by an underscore. For example:

```
The `Kestrel Homepage`_ is recently revamped.

.. _`Kestrel Homepage`: http://www.kestrel.edu
```

Will typeset as:

The Kestrel Homepage is recently revamped.

2.2 Inline Formatting

Inside a paragraph, you can use double-backticks to typeset text in a `monospaced` font:

```
Inside a paragraph, you can use double-backticks to typeset
text in a ``monospaced`` font.
```

Surround text with asterisks for *emphasis*, use double-asterisks for **strong** emphasis. This corresponds to *italics* and **bold** fonts:

```
Surround text with asterisks for *emphasis*, use double-asterisks
for **strong** emphasis. This corresponds to *italics* and **bold**
fonts.
```

2.3 Code Blocks

A preformatted code block begins with `::` at the current indentation level, then blank line, then the code, indented one level from the `::`. For example:

```
I'll show you come code below.

::

  map f []     = []
  map f (x:xs) = f x:map f xs
```

Becomes:

I'll show you come code below.

```
map f []     = []
map f (x:xs) = f x:map f xs
```

Rather than a blank line, you can use:

```
.. code-block:: common-lisp

   (defun map (f l) (if l (cons (f (car l)) (map f (cdr l))) l))
```

Which generates:

```
(defun map (f l) (if l (cons (f (car l)) (map f (cdr l))) l))
```

`common-lisp` can be replaced with the language that the code block is written in. When generating output for some formats, the tools will colorize and typeset the code appropriately, if it knows about the language. Sadly, Specware is not a supported language.

Finally, you can merge the `::` notation with the previous paragraph, as long as you skip a line. The tools will reduce the double-colon to a single colon:

```
This is a clever function::

  int f(int x, int y);
```

Results in the following output.

This is a clever function:

```
int f(int x, int y);
```

2.4 Lists

There are a number of ways to define lists.

A series of paragraphs, with the first line of each preceded by a hypen (-) will give a bullet list. If you want a second line at the same indent level, you have to leave a blank line. Otherwise the second line will be joined to the first line. This text:

```
- First Item.
  Not a Second line of first item.

- Second Item.

  A second line of second item.

- Third Item.
```

is displayed as:

- First Item. Not a Second line of first item.
- Second Item.

  A second line of second item.
- Third Item.

Other bullet list indicators, like `*`, are valid, as long as they are used uniformly for all items in the list.

Note that if you want to have multiple paragraphs under a bullet point (as with the first item above), indent the remaining paragraphs (separated by a blank line) as far as the text of the first paragraph of the item. These paragraphs can include sublists, simply by indenting the sublist to the level of the outer list item text.

A list will continue until it is followed by a paragraph (at the same indentation level as the list) that is not preceded by a list item indicator.

Numbered lists are much the same, except the list items are preceded by a `#.` instead of a hyphen (-):

```
#. First item.

#. Second item.
```

becomes:

1. First item.
2. Second item.

The tools automatically insert the proper numbers.

Definition lists are given as a series of lines, where the term being defined is given on the first line, then the definition is indented on the following line (with no separating blank line between the term and the definition:

```
one
  is the first number.
two
  comes right after one.
```

becomes:

one is the first number.

two comes right after one.

2.5 Shell commands

To typeset a shell command, use the `command` directive:

```
:command:`ls -la`
```

Which typesets as **ls -la**

2.6 Index Entries

To insert an entry into the index, use the `..  index` directive. The following will insert a single index entry `index entries` with subentry `defining` under `I`:

```
.. index::
   single: index entries; defining
```

The following is a shortcut way of defining two related index entries, each with a subentry. This will insert a pair of index entries, one for `shell-command`, with subentry `command-name`, and one for `command-name` with subentry `shell-command`:

```
.. index::
    pair: shell-command; command-name
```

The resulting index entries can be found at the end of this document. For complete usage information for the index directive, check the Sphinx Index Docs.

2.7 BNF Grammar Descriptions

Sphinx has support for typesetting BNFs. For example:

```
.. productionlist::
  wiffle: `waffle` [ `waffle_tail` ] |
        : `piffle` { + `piffle` }*
  piffle: 1 |
        : M { `piffle` }*
```

This will typeset as:

```
    wiffle  ::=   waffle [ waffle_tail ] |
                  piffle { + piffle }*
    piffle  ::=   1 |
                  M { piffle }*
```

Important points:

1. There's no blank line between the `..productionlist::` directive and the productions.
2. Each production begins with a nonterminal, followed by a colon, then a collection of right-hand sides.
3. In the RHS, if an identifier is surrounded by single-quotes then it's marked as a nonterminal. When docs are generated, the name in the RHS will be hyperlinked to the nonterminal's definition.
4. A production can span multiple lines, but each additional line after the first needs to begin with a colon horizontally aligned with the first line.
5. In the body of a document, you can make a hyperlinked reference to a nonterminal with the syntax: `:token:`name``, where `name` is the name of the nonterminal.

C

command-name
    shell-command, 8

I

index entries
    defining, 8

S

shell-command
    command-name, 8