sphinx-codefence

Oct 28, 2019

Contents:

This is a single-module sphinx extension that monkey-patches docutils adding the ability to parse code fences. For example, the following code block was generated with a codefence (check the "page source"):

```
def hello_codefence():
  print("I am in a codefence!")
```

CHAPTER 1

Why?

It can be cumbersome when copy-pasting many chunks of code into and out of reStructuredText documents due to the syntactic indentation required for literal text or code directives. Code fences allow you to copy-paste snippets into and out of your doc pages without having to fixup the indentation.

1.1 Installation

1.1.1 Local Install

The extension is a single file (*sphinx_codefence.py*) so the easiest thing to do is grab it and put it somewhere that sphinx can find it. For example, we can follow the recommendations of the sphinx documentation hello world extension. If your sphinx document tree looks like this:

```
├── build
├── Makefile
└── source
    ├── conf.py
    ├── index.rst
    ├── _static
    └── _templates
```

Then add a directory *_ext* to *source/* and put *sphinx_codefence.py* in it:

```
├── build
├── Makefile
└── source
    ├── conf.py
    ├── _ext
    │   └── sphinx_codefence.py
    ├── index.rst
    ├── _static
    └── _templates
```

Now update your *conf.py* with:

```
import os
import sys

# Add the local extension directory to the python path
sys.path.insert(0, os.path.abspath('./_ext'))

# include the condefence parser monkeypatch
extensions = [
  "sphinx_codefence"
]
```

1.1.2 PyPi

The extension is available via PYPI. You can install it using *pip*:

```
pip install sphinx-extension
```

And then update your *conf.py* adding "sphinx_codefence" to your list of extensions, such as:

```
extensions = [
  "sphinx_codefence"
]
```

1.2 Examples

The content of a codefence is parsed the same as the content of a `.. code::` directive.

For example, the following:

```
```
Hello world!
```
```

Is rendered as:

```
Hello world!
```

Code fences support languages. The language keyword is passed as the optional argument to the `.. code::` directive. For example:

```
```cpp
int main(int argc, char** argv){
  exit(0);
}
```
```

Is rendered as:

```
int main(int argc, char** argv){
  exit(0);
}
```

Code fences can also be nested within indented structures, such as:

```
.. tip::

   This code-fence is nested within an admonition.

   ```py
   def hello_world():
     print("hello world")
   ```
```

which is rendered as:

Tip: This code-fence is nested within an admonition.

```py
def hello_world():
  print("hello world")
```

However the whole point of using a code-fence is to avoid the indentation so I'm not sure why you'd want to do that.

There are two styles of codefence. You can either use triple-tick or triple-tilde. The examples thus-far have been triple-tick. Triple-tilda looks like this:

```
~~~py
def hello_codefence():
  print("I am in a codefence!")
~~~
```

Which renders as:

```py
def hello_codefence():
  print("I am in a codefence!")
```