FREE eBook

LEARNING Nokogiri

Free unaffiliated eBook created from
Stack Overflow contributors.

#nokogiri

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: nokogiri

It is an unofficial and free Nokogiri ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Nokogiri.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Nokogiri

Remarks

This section provides an overview of what Nokogiri is, and why a developer might want to use it.

It should also mention any large subjects within Nokogiri, and link out to the related topics. Since the Documentation for Nokogiri is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting Nokogiri set up or installed.

How to parse HTML or XML into a Nokogiri XML or HTML document

There isn't much to add to Nokogiri's "Parsing an HTML/XML Document" tutorial, which is an easy introduction to the subject, so start there, then return to this page to help fill in some gaps.

Nokogiri's basic parsing attempts to clean up a malformed document, sometimes adding missing closing tags, and will add some additional tags to make it correct.

This is an example of telling Nokogiri that the document being parsed is a complete HTML file, and Nokogiri discovering it isn't:

```
require 'nokogiri'

doc = Nokogiri::HTML('<body></body>')
puts doc.to_html
```

Which outputs:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/REC-
html40/loose.dtd">
<html><body></body></html>
```

Notice that the DTD statement was added, along with a wrapping `<html>` tag.

If we want to avoid this we can parse the document as a DocumentFragment:

```
require 'nokogiri'

doc = Nokogiri::HTML.fragment('<body></body>')
puts doc.to_html
```

which now outputs only what was actually passed in:

```
<body></body>
```

There is an XML variant also:

```
require 'nokogiri'

doc = Nokogiri::XML('<node />')
puts doc.to_xml
```

Which outputs:

```
<?xml version="1.0"?>
<node/>
```

and:

```
doc = Nokogiri::XML.fragment('<node />')
puts doc.to_xml
```

resulting in:

```
<node/>
```

A more verbose variation of `fragment` is to use `DocumentFragment.parse`, so sometimes you'll see it written that way.

Occasionally, Nokogiri will have to do some fix-ups to try to make sense of the document:

```
doc = Nokogiri::XML::DocumentFragment.parse('<node ><foo/>')
puts doc.to_xml
```

With the modified code now being:

```
<node>
  <foo/>
</node>
```

The same can happen with HTML.

Sometimes the document is mangled beyond Nokogiri's ability to fix it, but it will try anyway, resulting in a document that has a changed hierarchy. Nokogiri won't raise an exception, but it does provide a way to check for errors and the actions it took. See "How to check for parsing errors" for more information.

See the Nokogiri::XML::ParseOptions documentation for various options used when parsing.

How to check for parsing errors

Nokogiri is somewhat like a browser, in that it will attempt to provide something useful even if the incoming HTML or XML is malformed. Unfortunately it usually does it silently, but we can ask for a list of the errors using `errors`:

```
require 'nokogiri'

doc = Nokogiri::XML('<node><foo/>')
doc.errors
# => [#<Nokogiri::XML::SyntaxError: 1:13: FATAL: Premature end of data in tag node line 1>]
```

whereas, the correct XML would result in no errors:

```
doc = Nokogiri::XML('<node><foo/></node>')
doc.errors
# => []
```

This also applies to parsing HTML, but, because HTML is a relaxed form of XML, Nokogiri will pass over missing end-nodes often and will only report malformed nodes and more pathological errors:

```
doc = Nokogiri::HTML('<html><body>')
doc.errors
# => []

doc = Nokogiri::HTML('<html><body><p')
doc.errors
# => [#<Nokogiri::XML::SyntaxError: 1:15: ERROR: Couldn't find end of Start Tag p>]
```

If, after parsing, you can't find an node that you can see in your editor, this could be the cause of the problem. Sometimes it helps to pass the HTML through a formatter, and see if the nesting helps reveal the problem.

And, because Nokogiri tries to fix the problem but sometimes can't do it correctly, because it can be a very difficult thing for software to do, we'll have to pre-process the file and touch up lines prior to handing it off to Nokogiri. How to do that depends on the file and the problem. It can vary from simply finding a node and adding a trailing `>`, to removing embedded malformed markup that was injected by a bad scraping routine, so it's up to the programmer how best to intercede.

How to extract text from a node or nodes

How to correctly extract text from nodes is one of the most popular questions we see, and almost invariably is made more difficult by misusing Nokogiri's "searching" methods.

Nokogiri supports using CSS and XPath selectors. These are equivalent:

```
doc.at('p').text   # => "foo"
doc.at('//p').text # => "foo"

doc.search('p').size   # => 2
doc.search('//p').size # => 2
```

The CSS selectors are extended with many of jQuery's CSS extensions for convenience.

`at` and `search` are generic versions of `at_css` and `at_xpath` along with `css` and `xpath`. Nokogiri makes an attempt to determine whether a CSS or XPath selector is being passed in. It's possible to create a selector that fools `at` or `search` so occasionally it will misunderstand, which is why we have the more specific versions of the methods. In general I use the generic versions almost always, and only use the specific version if I think Nokogiri will misunderstand. This practice falls under the first entry in "Three Virtues".

If you are searching for one specific node and want its text, then use `at` or one of its `at_css` or `at_xpath` variants:

```
require 'nokogiri'

doc = Nokogiri::HTML(<<EOT)
<html>
  <body>
    <p>foo</p>
    <p>bar</p>
  </body>
</html>
EOT

doc.at('p').text # => "foo"
```

`at` is equivalent to `search(...).first`, so you *could* use the longer-to-type version, but why?

If the text being extracted is concatenated after using `search`, `css` or `xpath` then add `map(&:text)` instead of simply using `text`:

```
require 'nokogiri'

doc = Nokogiri::HTML(<<EOT)
<html>
  <body>
    <p>foo</p>
    <p>bar</p>
  </body>
</html>
EOT

doc.search('p').text         # => "foobar"
doc.search('p').map(&:text) # => ["foo", "bar"]
```

See the `text` documentation for NodeSet and Node for additional information.

Read Getting started with Nokogiri online: https://riptutorial.com/nokogiri/topic/10172/getting-started-with-nokogiri

Credits

S. No	Chapters	Contributors
1	Getting started with Nokogiri	Community, the Tin Man