# PyAnyAPI Documentation

***Release 0.6.1***

**Dmitry Dygalo**

**Nov 07, 2017**

# Contents

Contents:

CHAPTER 1

# Usage

The library provides an ability to create API over various content. Currently there are bundled tools to work with HTML, XML, CSV, JSON and YAML. Initially it was created to work with `requests` library.

## 1.1 Basic setup

Basic parsers can be declared in the following way:

```
from pyanyapi.parsers import HTMLParser


class SimpleParser(HTMLParser):
    settings = {'header': 'string(.//h1/text())'}


>>> api = SimpleParser().parse('<html><body><h1>Value</h1></body></html>')
>>> api.header
Value
```

Or it can be configured in runtime:

```
from pyanyapi.parsers import HTMLParser


>>> api = HTMLParser({
    'header': 'string(.//h1/text())'
}).parse('<html><body><h1>Value</h1></body></html>')
>>> api.header
Value
```

To get all parsing results as a dict there is `parse_all` method. All properties (include defined with `@interface_property` decorator) will be returned.

```
from pyanyapi.parsers import JSONParser

>>> JSONParser({
    'first': 'container > 0',
    'second': 'container > 1',
    'third': 'container > 2',
}).parse('{"container":["first", "second", "third"]}').parse_all()
{
    'first': 'first',
    'second': 'second',
    'third': 'third',
}
```

## 1.2 Complex setup

In some cases you may want to apply extra transformations to result list. Here comes "base-children" setup style.

```
from pyanyapi.parsers import HTMLParser


class SimpleParser(HTMLParser):
    settings = {
        'test': {
            'base': '//test',
            'children': 'text()|*//text()'
        }
    }


>>> api = SimpleParser().parse('<xml><test>123 </test><test><inside> 234</inside></
↪test></xml>')
>>> api.test
['123 ', ' 234']
```

There is another option to interact with sub-elements. Sub parsers!

```
from pyanyapi.parsers import HTMLParser


class SubParser(HTMLParser):
    settings = {
        'href': 'string(//@href)',
        'text': 'string(//text())'
    }


class Parser(HTMLParser):
    settings = {
        'elem': {
            'base': './/a',
            'parser': SubParser
        }
    }

>>> api = Parser().parse("<html><body><a href='#test'>test</a></body></html>")
```

```
>>> api.elem[0].href
#test
>>> api.elem[0].text
test
>>> api.parse_all()
{'elem': [{'href': '#test', 'text': 'test'}]}
```

Also you can pass sub parsers as classes or like instances.

## 1.3 Settings inheritance

Settings attribute is merged from all ancestors of current parser.

```
from pyanyapi.parsers import HTMLParser


class ParentParser(HTMLParser):
    settings = {'parent': '//p'}


class FirstChildParser(ParentParser):
    settings = {'parent': '//override'}


class SecondChildParser(ParentParser):
    settings = {'child': '//h1'}


>>> FirstChildParser().settings['parent']
//override

>>> SecondChildParser().settings['parent']
//p

>>> SecondChildParser().settings['child']
//h1

>>> SecondChildParser({'child': '//more'}).settings['child']
//more
```

## 1.4 Results stripping

Parsers can automagically strip trailing whitespaces with `strip=True` option.

```
from pyanyapi.parsers import XMLParser


>>> settings = {'p': 'string(//p)'}
>>> XMLParser(settings).parse('<p> Pcontent </p>').p
 Pcontent
>>> XMLParser(settings, strip=True).parse('<p> Pcontent </p>').p
Pcontent
```

CHAPTER 2

# Parsers

## 2.1 HTML & XML

For HTML and XML based interfaces XPath 1.0 syntax is used for settings declaration. Unfortunately XPath 2.0 is not supported by lxml. XML is about the same as HTMLParser, but uses a different lxml parser internally. Here is an example of usage with `requests`:

```
>>> import requests
>>> import pyanyapi
>>> parser = pyanyapi.HTMLParser({'header': 'string(.//h1/text())'})
>>> response = requests.get('http://example.com')
>>> api = parser.parse(response.text)
>>> api.header
Example Domain
```

If you need, you can execute more XPath queries at any time you want:

```
from pyanyapi.parsers import HTMLParser


>>> parser = HTMLParser({'header': 'string(.//h1/text())'})
>>> api = parser.parse('<html><body><h1>This is</h1><p>test</p></body></html>')
>>> api.header
This is
>>> api.parse('string(//p)')
test
```

## 2.2 XML Objectify

Lxml provides interesting feature - objectified interface for XML. It converts whole XML to Python object. This parser doesn't require any settings. E.g:

```
from pyanyapi.parsers import XMLObjectifyParser


>>> XMLObjectifyParser().parse('<xml><test>123</test></xml>').test
123
```

## 2.3 JSON

Settings syntax in based on PostgreSQL statements syntax.

```
from pyanyapi.parsers import JSONParser


>>> JSONParser({'id': 'container > id'}).parse('{"container":{"id":"123"}}').id
123
```

Or you can get access to values in lists by index:

```
from pyanyapi.parsers import JSONParser


>>> JSONParser({'second': 'container > 1'}).parse('{"container":["first", "second",
↪"third"]}').second
second
```

And executes more queries after initial parsing:

```
from pyanyapi.parsers import JSONParser


>>> api = JSONParser({'second': 'container > 1'}).parse('{"container":[],"second_
↪container":[123]}')
>>> api.parse('second_container > 0')
123
```

## 2.4 YAML

Equal to JSON parser, but works with YAML data.

```
from pyanyapi.parsers import YAMLParser


>>> YAMLParser({'test': 'container > test'}).parse('container:\n    test: "123"').test
123
```

## 2.5 Regular Expressions Interface

In case, when data has wrong format or is just very complicated to be parsed with bundled tools, you can use a parser based on regular expressions. Settings are based on Python's regular expressions. It is the most powerful parser, because of its simplicity.

```
from pyanyapi.parsers import RegExpParser


>>> RegExpParser({'error_code': 'Error (\d+)'}).parse('Oh no!!! It is Error 100!!!').
↪error_code
100
```

And executes more queries after initial parsing:

```
from pyanyapi.parsers import RegExpParser


>>> api = RegExpParser({'digits': '\d+'}).parse('123abc')
>>> api.parse('[a-z]+')
abc
```

Also, you can pass flags for regular expressions on parser initialization:

```
from pyanyapi.parsers import RegExpParser


>>> RegExpParser({'test': '\d+.\d+'}).parse('123\n234').test
123
>>> RegExpParser({'test': '\d+.\d+'}, flags=re.DOTALL).parse('123\n234').test
123
234
```

## 2.6 CSV Interface

Operates with CSV data with simple queries in format 'row_id:column_id'.

```
from pyanyapi.parsers import CSVParser


>>> CSVParser({'value': '1:2'}).parse('1,2,3\r\n4,5,6\r\n').value
6
```

Also, you can pass custom kwargs for *csv.reader* on parser initialization:

```
from pyanyapi.parsers import CSVParser


>>> CSVParser({'value': '1:2'}, delimiter=';').parse('1;2;3\r\n4;5;6\r\n').value
6
```

## 2.7 AJAX Interface

AJAX is a very popular technology and often use JSON data with HTML values. Here is an example:

```
from pyanyapi.parsers import AJAXParser


>>> api = AJAXParser({'p': 'content > string(//p)'}).parse('{"content": "<p>Pcontent</
↪p>"}')
```

```
>>> api.p
Pcontent
```

It uses combination of XPath queries and PostgreSQL-based JSON lookups. Custom queries execution is also available:

```
from pyanyapi.parsers import AJAXParser


>>> api = AJAXParser().parse('{"content": "<p>Pcontent</p><span>123</span>"}')
>>> api.parse('content > string(//span)')
123
```

## 2.8 Custom Interface

You can easily declare your own interface. For that you should define `execute_method` method. And optionally `perform_parsing`. Here is an example of naive CSVInterface, which provides an ability to get the column value by index. Also you should create a separate parser for that.

```
from pyanyapi.interfaces import BaseInterface
from pyanyapi.parsers import BaseParser


class CSVInterface(BaseInterface):

    def perform_parsing(self):
        return self.content.split(',')

    def execute_method(self, settings):
        return self.parsed_content[settings]


class CSVParser(BaseParser):
    interface_class = CSVInterface


>>> CSVParser({'second': 1}).parse('1,2,3').second
2
```

## 2.9 Extending interfaces

Also content can be parsed with regular Python code. It can be done with special decorators `interface_method` and `interface_property`.

Custom method example:

```
from pyanyapi.decorators import interface_method
from pyanyapi.parsers import interface_method


class ParserWithMethod(HTMLParser):
    settings = {'occupation': 'string(.//p/text())'}
```

```
    @interface_method
    def hello(self, name):
        return name + ' is ' + self.occupation


>>> api = ParserWithMethod().parse('<html><body><p>programmer</p></body></html>')
>>> api.occupation
programmer

>>> api.hello('John')
John is programmer
```

Custom property example:

```
from pyanyapi.decorators import interface_property
from pyanyapi.parsers import HTMLParser


class ParserWithProperty(HTMLParser):
    settings = {'p': 'string(.//p/text())', 'h1': 'string(.//h1/text())'}

    @interface_property
    def test(self):
        return self.h1 + ' ' + self.p


>>> api = ParserWithProperty().parse('<html><body><h1>This is</h1><p>test</p></body></
↪html>')
>>> api.h1
This is

>>> api.p
test

>>> api.test
This is test
```

Certainly the previous example can be done with more complex XPath expression, but in general case XPath is not enough.

CHAPTER 3

# Complex content parsing

## 3.1 Combined parsers

In situations, when particular content type is unknown before parsing, you can create combined parser, which allows you to use multiply different parsers transparently. E.g. some server usually returns JSON, but in cases of server errors it returns HTML pages with some text. Then:

```
from pyanyapi.parsers import CombinedParser, HTMLParser, JSONParser


class Parser(CombinedParser):
    parsers = [
        JSONParser({'test': 'test'}),
        HTMLParser({'error': 'string(//span)'})
    ]

>>> parser = Parser()
>>> parser.parse('{"test": "Text"}').test
Text
>>> parser.parse('<body><span>123</span></body>').error
123
```

## 3.2 Another example

Sometimes different content types can be combined inside single string. Often with AJAX requests.

```
{"content": "<span>Text</span>"}
```

You can work with such data in the following way:

```
from pyanyapi.decorators import interface_property
from pyanyapi.parsers import HTMLParser, JSONParser
```

```
inner_parser = HTMLParser({'text': 'string(.//span/text())'})


class AJAXParser(JSONParser):
    settings = {'content': 'content'}

    @interface_property
    def text(self):
        return inner_parser.parse(self.content).text


>>> api = AJAXParser().parse('{"content": "<span>Text</span>"}')
>>> api.text
Text
```

Now AJAXParser is bundled in pyanyapi, but it works differently. But anyway, this example can be helpful for building custom parsers.

CHAPTER 4

# Changelog

## 4.1 0.6.0 - 09.08.2016

- IndexOf parser.

## 4.2 0.5.8 - 14.07.2016

- Fixed XML content parsing for bytes input.

## 4.3 0.5.7 - 28.01.2016

- Added `parse_all` call on subparsers (#37).

## 4.4 0.5.6 - 24.11.2015

- Fixed `super` call in exception.

## 4.5 0.5.5 - 23.11.2015

- Add content to exceptions in case of parsing errors (#35).

## 4.6 0.5.4 - 15.11.2015

- Fixed `lxml` installation on PyPy (#34).

- Add support for subparsers (#32).

## 4.7 0.5.3 - 30.10.2015

- Disable stripping in XMLObjectifyParser on PyPy (#30).

## 4.8 0.5.2 - 20.10.2015

- Fix incorrect stripping in XMLObjectifyParser (#29).

## 4.9 0.5.1 - 20.10.2015

- Ability to override `strip` attribute at class level (#27).
- Fix `strip` in XMLObjectifyParser (#28).

## 4.10 0.5 - 05.10.2015

- Add `parse_all` to parse all settings (#20).
- Settings for regular expressions (#19).
- Add `strip` option to strip trailing whitespaces (#14).
- Add CSVParser (#11).

## 4.11 0.4 - 29.09.2015

- Add YAMLParser (#5).
- Add AJAXParser (#9).
- `parse` calls memoization (#18).

## 4.12 0.3 - 24.09.2015

- Add partial support for PyPy3 (#7).
- Add partial support for Jython (#6).
- Add ujson as dependency where it is possible (#4).
- Lxml will not be installed where it is not supported (#3).

## 4.13 0.2.1 - 23.09.2015

- Remove encoding declaration for XMLObjectifyParser

## 4.14 0.2 - 23.09.2015

- Add `parse` methods for JSONInterface & RegExpInterface (#8).
- Add universal wheel config (#2).

## 4.15 0.1 - 22.09.2015

- First release.

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search