WDOM Documentation

Release 0.1.3

Hiroyuki Takagi

May 20, 2016

Contents

WDOM is a python GUI library for browser-based desktop applications. WDOM controls HTML elements (DOM) on browser from python, as if it is a GUI element. APIs are same as browser DOM, but of course, you can write logic codes in python.

This library includes web-server (tornado/aiohttp), but is not intended to be used as a web framework, please use for **Desktop** GUI Applications!

CHAPTER 1

Disclaimer

WDOM is in the early development stage, and may contain many bugs. All APIs are not stable, and may be changed in future release.

CHAPTER 2

Features

- Pure python implementation
- APIs based on DOM specification
  - Not need to learn new special classes/methods for GUI
  - Implemented DOM features are listed in Wiki page
- Theming with CSS frameworks (see ScreenShots on Wiki)
- JavaScript codes are executable on browser
- Testable with browsers and Selenium WebDriver
- Licensed under MIT licence

CHAPTER 3

Requirements

Python 3.4.4+ and any modern-browsers are required. Also supports Electron and PyQt's webkit browsers. IE is not supported, but most of features will work with IE11 (but not recomended).

CHAPTER 4

Installation

Install by pip:

```
pip install wdom
```

Or, install latest version from github:

```
pip install git+http://github.com/miyakogi/wdom
```

As WDOM depends on tornado web framework, it will be installed automatically. Optionally supports aiohttp, which is a web framework natively supports asyncio and is partly written in C. Using aiohttp will result in better performance. If you want to use WDOM with aiohttp, install it with pip:

```
pip install aiohttp
```

Any configurations are not required; when aiohttp is available, WDOM will use it automatically.

CHAPTER 5

Example

Simple example:

```
import asyncio
from wdom.document import get_document
from wdom.server import start_server, stop_server

if __name__ == '__main__':
    document = get_document()
    h1 = document.createElement('h1')
    h1.textContent = 'Hello, WDOM'
    document.body.appendChild(h1)

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Execute this code and access `http://localhost:8888` by browser. `"Hello, WDOM"` will shown on the browser. To stop process, press `CTRL+C`.

As you can see, methods of WDOM (`document.createElement` and `document.body.appendChild`) are very similar to browser JavaScript.

WDOM provides some new DOM APIs (e.g. `append` for appending child) and some tag classes to easily generate elements:

```
import asyncio
from wdom.tag import H1
from wdom.document import get_document
from wdom.server import start_server, stop_server

if __name__ == '__main__':
    document = get_document()
    h1 = H1()
    h1.textContent = 'Hello, WDOM'
    document.body.append(h1)

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Of course, WDOM can handle events:

```
import asyncio
from wdom.tag import H1
from wdom.server import start_server, stop_server
from wdom.document import get_document

if __name__ == '__main__':
    document = get_document()
    h1 = H1('Hello, WDOM', parent=document.body)
    def rev_text(event):
        h1.textContent = h1.textContent[::-1]
    h1.addEventListener('click', rev_text)
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

When string `"Hello, WDOM"` is clicked, it will be flipped.

Making components with python class:

```
import asyncio
from wdom.tag import Div, H1, Input
from wdom.server import start_server, stop_server
from wdom.document import get_document

class MyApp(Div):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.text = H1('Hello', parent=self)
        self.textbox = Input(parent=self, placeholder='input here...')
        self.textbox.addEventListener('input', self.update)

    def update(self, event):
        self.text.textContent = event.target.value
        # or, you can write as below
        # self.text.textContent = self.textbox.value

if __name__ == '__main__':
    document = get_document()
    document.body.append(MyApp())
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

WDOM package includes some tiny examples. From command line, try:

```
python -m wdom.exapmles.rev_text
python -m wdom.exapmles.data_binding
python -m wdom.exapmles.timer
```

Source codes of these examples will be found in wdom/examples.

More documents are in preparation, but you can see them in **docs** directory of this repository.

CHAPTER 6

Contributing

Contributions are welcome!!

If you find any bug, or have any comments, please don't hesitate to report to issues on GitHub. All your comments are welcome!

CHAPTER 7

About This Document

This document is still incomplete, and will contain typos and bad descriptions. Furthermore, my English is so bad, so if you find any incorrect expression, please let me know at issues or send PR. Any small improvements/suggestions (e.g. just changing **a** to **the**) are welcome!

CHAPTER 8

Contents

8.1 User Guide

8.1.1 Basic DOM Features

First Example

Program to show "Hello, WDOM" on browser is:

```
import asyncio
from wdom.document import get_document
from wdom.server import start_server, stop_server

if __name__ == '__main__':
    document = get_document()
    h1 = document.createElement('h1')
    h1.textContent = 'Hello, WDOM'
    document.body.appendChild(h1)

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Save and execute code and access http://localhost:8888 by browser. Then, web page with "Hello, WDOM" will be shown.

The four lines beginning from document = get_document() are very similar to JavaScript. document returned by get_document() is a Document Node, equivalent to document object in JavaScript on browsers. document.createElement('{tag-name}') generates new element with the given tag-name. appendChild method inserts the child node at the last of its child nodes. Not used in the sample code, by using removeChild method, one can remove the child node.

Model of WDOM Elements

GUI elements of the WDOM are composed of HTML tags (elements or nodes) on a browser. Usually they have one-by-one relationships: thus document.createElement('h1') returns one WDOM element, and it is rendered as an <h1> tag on a browser. All WDOM elements have their own HTML representations, which can be obtained by their html property.

Elements created by WDOM are all based on DOM Living Standard and related standards (HTML Living Standard, CSS Object Model, DOM Parsing, and Custom Elements in WebComponents).

As elements are modeled by DOM, you can add/remove/replace them as same as the way in JavaScript on browsers. Currently, not all of DOM features have been implemented in WDOM, but lots of frequently-used methods/properties are available. Implemented features are listed in wiki pages at gihub.

Create New Element

To make elements, WDOM provides two methods.

One is `document.createElement` mentioned above, and the other is to instantiate classes defined in `wdom.tag` module. For details about the `wdom.tag` module, see *Python Classes and Extensions of WDOM* section.

Note: Every element does not appear on browser until inserted to the DOM tree which roots on the document node returned by `get_document()`.

Append/Insert Node

To insert nodes on the DOM tree, use `appendChild` or `insertBefore` method. `A.appendChild(B)` append the node B at last of child nodes of the parent node A. `A.insertBefore(B, C)` inserts new element B just before the reference node C. The reference node C must be a child node of the parent node A.

These method names are quite long, so some methods specified in DOM specification are also available on WDOM: `prepend`, `append`, `before`, `after`, and `replaceWith`. Details about these methods are described in *Newest DOM Features* section.

Remove Node

It is also able to remove child node from the DOM tree.

`A.removeChild(B)` removes the child node B from the parent node A. If B is not a child node of A, it will raise Error.

More simple method `B.remove()` is also available, see *Newest DOM Features* section.

Access Child/Parent/Sibling Nodes

`childNodes` property returns list-like live-object which contains its direct child nodes. `firstChild` and `lastChild` property returns its first/last child node.

On the other way, `parentNode` property returns its parent node. These properties are same as JavaScript's DOM.

`nextSibling` and `previousSibling` returns its next/previous sibling node.

If there is no corresponding node, all these properties return `None`.

Attributes

To get element's attributes like `class="..."` in HTML tag, use `getAttribute('attribute-name')` method. If called `getAttribute` to the attribute which does not exists, it will return `None`.

To set or change attribute's value, use `setAttribute('attribute-name', value)` method. And to remove an attribute, use `removeAttribute` method.

To obtain all attributes set for the element, access `attributes` property. This property returns dictionary-like abject `NamedNodeMap`. This object has attributes and its value as `{'attribute-name': value, ...}`.

Special Attributes

Some attributes are accessible via special properties, for exmaple, `A.id` returns its ID attribute. Available properties will be found in wiki page at gihub.

With `getAttribute` returns string or None, but attributes accessed via its properties return different types depending on its property. For example, `element.id` return always string even if it is not set (in case id is not set, `element.id` returns empty string, not None). Similarly, `element.hidden` returns boolean (`True` or `False`) and `element.style` returns `CSSStyleDeclaration`.

- **References**
  - HTMLElement | MDN
  - element.id | MDN
  - element.style | MDN

Style Attribute (CSSStyleDeclaration)

`CSSStyleDeclaration` obtained by `element.style` provides property access to its css properties. For example, `element.style.color = 'red'` makes element's color red.

Some css properties including `-` will be converted to CamelCase name, for example, `background-color` will become `element.style.backgroundColor`. For more examples, please refer to CSS Properties Reference

Using HTML

Making large applications with `appendChild` and `insertBefore` are quite difficult. So writing HTML and parse it to WDOM elements is sometimes useful.

It can be done by `innerHTML` method, as same as JavaScript. An example to make large list is below:

```
from wdom.tag import Ul

ul = Ul()
ul.innerHTML = '''\
<li>item1</li>
<li>item2</li>
<li>item3</li>
<li>item4</li>
'''

print(ul.html_noid)  # <ul><li>...

# Accessing child nodes
# for child in ul.childNodes:
#     print(child.html)

# or, first/lastChild
print(ul.firstChild.html)
```

```
print(ul.lastChild.html)

# excluding Text nodes
for child in ul.children:
    print(child.html)

# first/lastElementChild
print(ul.firstElementChild.html)
print(ul.lastElementChild.html)
```

Note: Assignment to `innerHTML` **removes all child nodes** and insert parsed elements.

Each child nodes can be accessed via `childNodes` property, which returns list-like live-object, but not able to modify its values.

`insertAdjacentHTML({position}, {html})` also parses HTML and insert new elements to the `position`. This method is also same as JavaScript's one, so for details please see Element.insertAdjacentHTML() | MDN or DOM specification.

`outerHTML` is not implemented.

Events

Reverse Text on Click

WDOM's event handling is also same as JavaScript:

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document

if __name__ == '__main__':
    document = get_document()
    h1 = document.createElement('h1')
    h1.textContent = 'Hello, WDOM'
    def rev_text(event):
        h1.textContent = h1.textContent[::-1]
    h1.addEventListener('click', rev_text)
    document.body.appendChild(h1)
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Run this code and click `"Hello, WDOM"`. Then it will be reversed. When clicked again, it will be reversed again and back to `"Hello, WDOM"`.

`addEventListener('{event-type}', {handler})` method registers `handler` to the given `event-type`. In the sample code, `rev_text` function is registered to `click` event. Values available for event type are same as JavaScript's DOM, as listed in Event reference | MDN.

When the `h1` element is clicked, registered function `rev_text` is called with a single argument, event, which is an Event object, though it is not used in the above example.

User Input Event

The below sample shows how to use event object:

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document

if __name__ == '__main__':
    document = get_document()
    h1 = document.createElement('h1')
    h1.textContent = 'Hello, WDOM'
    input = document.createElement('textarea')
    def update(event):
        h1.textContent = event.target.value
    input.addEventListener('input', update)
    document.body.appendChild(input)
    document.body.appendChild(h1)

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

In this sample, `textarea` element is added. When user inputs some text on the `textarea`, it will be shown as `h1` element.

In the `update` function, `event.target` has a reference to the element which emitted the event, in this case it is a `textarea` element. And, as same as JavaScript, a `textarea` element (and `input` element) contains its current value at `value` attribute. At the moment `update` function is called, `textarea.value` is already updated to the latest value. So the above code you can use `textarea.value` instead of `event.target.value`. In the sample code, setting its value to `h1` element's `textContent`.

8.1.2 Python Classes and Extensions of WDOM

Generating new elements by `document.createElement` every time is quite mess. So WDOM's `wdom.tag` module provides simple tag classes usually used.

By using tag classes, the previous example can be written as:

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document
from wdom.tag import Div, H1, Input

class MyElement(Div):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.h1 = H1()
        self.h1.textContent = 'Hello, WDOM'
        self.input = Input()
        self.input.addEventListener('input', self.update)
        self.appendChild(self.input)
        self.appendChild(self.h1)

    def update(self, event):
```

```
        self.h1.textContent = event.target.value

if __name__ == '__main__':
    document = get_document()
    document.body.appendChild(MyElement())
    server = start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Instead of using `document.createElement`, in the above example is using `H1` class and `Input` class to generate h1 element and input element. Furthermore, `MyElement` class inherits `Div` class. Its instance is rendered as `div` element, or `<div>` tag on a browser. These instances are same as instances generated by `document.createElement` method.

Names of pre-defined classes on `wdom.tag` module are same as related tag names, starting with upper-case and followed by lower-case. For example, `<button>` tag is `Button` class, `<br>` tag is `Br` class, and `<textarea>` tag is `Textarea`.

Actual HTML strings can be obtained by `html` property of each elements. For example, `print(MyElement().html)` shows:

```
<div rimo_id="..."><input rimo_id="..."><h1 rimo_id="...">Hello, WDOM</h1></div>
```

`rimo_id` attribute is used internally. If you want to omit it for tests, use `html_noid` property instead:

```
print(MyElement().html_noid)
# -> <div><input><h1>Hello, WDOM</h1></div>
```

Append to Parent Node

By using `parent` argument of constructor, newly generated elements will be automatically appended to the parent node. Using this, the above example can be written as:

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document
from wdom.tag import Div, H1, Input

class MyElement(Div):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.h1 = H1(parent=self)
        self.h1.textContent = 'Hello, WDOM'
        self.input = Input(parent=self)
        self.input.addEventListener('input', self.update)

    def update(self, event):
        self.h1.textContent = event.target.value

if __name__ == '__main__':
    document = get_document()
    document.body.appendChild(MyElement())
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
```

```
    except KeyboardInterrupt:
        stop_server()
```

At the line `self.h1 = H1(parent=self)`, new `h1` element is automatically appended to `self` as its child node.

Append Child Nodes

In the other way, child nodes can be appended on generation.

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document
from wdom.tag import Div, H1, Input

class MyElement(Div):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.h1 = H1('Hello, WDOM', parent=self)
        self.input = Input(parent=self)
        self.input.addEventListener('input', self.update)

    def update(self, event):
        self.h1.textContent = event.target.value

if __name__ == '__main__':
    document = get_document()
    document.body.appendChild(MyElement())
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

At the `self.h1 = H1('Hello, WDOM', parent=self)`, the first argument is converted to Text Node and appended to the newly generated `h1` element. With multiple arguments, more than one child node can be appended, like `H1(H2(), P(), ...)`.

Initialization with Attributes

Attributes are also able to be defined with keyword arguments on the constructor.

```
from wdom.tag import Input

input = Input(type='checkbox')
print(input.html_noid)  # <input type="checkbox">

# this is equivalent to:
input = Input()
input.setAttribute('type', 'checkbox')
# also same as:
input.type = 'checkbox'
```

`class` is a python's keyword, so use `class_` (trailing underscore) instead.

```
from wdom.tag import H1

h1 = H1(class_='title')
print(h1.html_noid)  # <h1 class="title"></h1>

# this is equivalent to:
h1 = H1()
h1.setAttribute('class', 'title')
# also same as:
h1.classList.add('title')
# classList.add accepts mutliple arguments
h1.classList.add('title', 'heading', '...', )
```

Default Class Attribute

User-defined class can have default class attributes.

```
from wdom.tag import Button

class MyButton(Button):
    class_ = 'btn'

print(MyButton().html_noid)
# <button class="btn"></button>

# This is almost same as:
class MyButton2(Button):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.setAttribute('class', 'btn')
        ...

# class-level classes are not able to remove from instance
btn = MyButton()
btn.classList.remove('btn')
print(btn.html_noid)  # <button class="btn"></button>

btn2 = MyButton2()
btn2.classList.remove('btn')
print(btn2.html_noid)  # <button></button>

# Inherited class_ not overrides super-classes class_
class DefaultButton(MyButton):
    class_ = 'btn-default'

db = DefaultButton()
print(db.html_noid)  # <button class="btn btn-default"></button>
```

`class_` class-variable is added to the instance. This attribute cannot be removed at its instance and it is inherited to the subclasses. If you don't want to inherit parent's class attributes, add `inherit_class = False` as a class variable.

Shortcut of Class Definition

Defining lots of similar classes by `class` statement is quite mess. WDOM provides `wdom.tag.NewTagClass` function to make new user-defined classes.

`MyButton` class and `DefaultButton` class defined in the above example can be defined simply by using `NewTagClass` function as below:

```
from wdom.tag import Button, NewTagClass

# Making new class easily
MyButton = NewTagClass('MyButton', 'button', Button, class_='btn')
DefaultButton = NewTagClass('DefaultButton', 'button', MyButton, class_='btn-default')

print(MyButton().html_noid)
print(DefaultButton().html_noid)
```

The first argument is a name of new class, the second is a tag name, the third is a base class, and the firth and other keyword arguments are class-variables of the new class. To inherit multiple classes, use tuple at the third argument.

These features are not DOM standard and specially defined for WDOM.

Execute JavaScript on Browser

(to be written)

8.1.3 Loading Static Contents

Contents on the Web

As an example, use bootstrap.

To use bootstrap, one css file (bootstrap.min.css) and two js files (jquery and bootstrap.min.js) are need to be loaded. To load css file, use `<link>` tag and insert into the `<head>`. And to load js files, use `<script>` tag and insert into the `<body>`. Both `<head>` and `<body>` tags can be accessed via `document.head` and `document.body` (same as JavaScript).

```
import asyncio
from wdom.document import get_document
from wdom.server import start_server, stop_server
from wdom.tag import Link, Script, Button

if __name__ == '__main__':
    document = get_document()
    # Add <link>-tag sourcing bootstrap.min.css on <head>
    document.head.appendChild(Link(rel='stylesheet', href='https://maxcdn.bootstrapcdn.com/bootstrap/

    # Add <script>-tag sourcing jquery and bootstrap.min.js to <body>
    document.body.appendChild(Script(src='https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery
    document.body.appendChild(Script(src='https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstra

    # Add bootstrap button element
    document.body.appendChild(Button('click', class_='btn btn-primary'))

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

As frequently required to load css and js files on document, WDOM provides shortcut method; `document.add_cssfile({/path/to/cssfile})` and `document.add_jsfile({/path/to/jsfile})`.

```
import asyncio
from wdom.document import get_document
from wdom.server import start_server, stop_server
from wdom.tag import Button

if __name__ == '__main__':
    document = get_document()
    # Add <link>-tag sourcing bootstrap.min.css on <head>
    document.add_cssfile('https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css')
    # Add <script>-tag sourcing jquery and bootstrap.min.js to <body>
    document.add_jsfile('https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js')
    document.add_jsfile('https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js')

    # Add bootstrap button element
    document.body.appendChild(Button('click', class_='btn btn-primary'))

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Local Resources

User's css files and other static contents, like images or html files, are also available on WDOM app.

For the directory tree like below:

```
.
-- static
|   -- css
|       -- app.css
-- app.py
-- module1.py
-- ...
...
```

When want to use `static/css/app.css` from `app.py`, `app.py` will become as follows:

```
import asyncio
from os import path

from wdom.document import get_document
from wdom.server import start_server, stop_server, add_static_path
from wdom.tag import Button

if __name__ == '__main__':
    static_dir = path.join(path.dirname(path.abspath(__file__)), 'static')
    document = get_document()
    document.add_cssfile('/static/css/app.css')

    # Add button element
    document.body.appendChild(Button('click'))

    add_static_path('static', static_dir)
```

```
    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

The first argument of the `add_static_path` is a prefix to access the static files and the second argument is a path to the directory to be served. Files under the assigned directory can be accessed by URL like `http://localhost:8888/prefix/(dirname/)filename`. For example, if accessed to `http://localhost:8888/static/css/app.css` with a browser, `app.css` will be shown.

It's not necessary to use the same name for the prefix as the directory name to be registered. For example, in case to use `my_static` as a prefix, change to `add_static_path('my_static', static_dir)` and then can be accessed to `app.css` from `http://localhost:8888/my_static/css/app.css`.

Not only css files but also any static files, like js files, html files, or images are able to be served.

Any prefixes can be used if it is valid for URL, but `_static` and `tmp` is already used by WDOM internally, so do not use them for a prefix.

8.1.4 Newest DOM Features

Some new features of DOM, which have not been implemented yet on browsers are available on WDOM.

ParentNode and ChildNode Interfaces

`appendChild` method add only one child node, but `append` method can append multiple nodes at once. Furthermore, strings are also available (strings are automatically converted to Text Node).

```
from wdom.tag import Ul, Li

ul = Ul()
li1 = Li('item1')
li2 = Li('item2')
...

ul.appendChild(li1)
ul.appendChild(li2)
...
print(ul.html_noid)

# by append
ul2 = Ul()
ul2.append(Li('item1'), Li('item2'))
print(ul2.html_noid)
```

Similarly, `prepend`, `after`, and `before` methods are available. Furthermore, `remove`, `replaceWith`, `children`, `firstElementChild`, and `lastElementChild` methods are also available on WDOM.

Internally, these methods update view on the browser at once, so using these methods usually result in better performance.

- References
  - ParentNode | DOM Standard
  - NonDocumentTypeChildNode | DOM Standard
  - ChildNode | DOM Standard

Custom Element

WDOM provides limited supports on custom elements (experimentally).

User Defined Custom Tags

As an example, define `MyElement` as a custom tag (`<my-element>`).

```
import asyncio
from wdom.tag import Div, H1, Input
from wdom.document import get_document
from wdom.server import start_server, stop_server

class MyElement(Div):
    tag = 'my-element'  # custom tag name
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.h1 = H1('Hello, WDOM', parent=self)
        self.input = Input(parent=self)
        self.input.addEventListener('input', self.update)

    def update(self, event):
        self.h1.textContent = event.target.value

if __name__ == '__main__':
    document = get_document()
    # Register MyElement
    document.defaultView.customElements.define('my-element', MyElement)
    # Make instance of MyElement from HTML
    document.body.insertAdjacentHTML('beforeend', '<my-element></my-element>')
    # Of, from createElement
    my_element = document.createElement('my-element')
    document.body.appendChild(my_element)

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

Difference is a class variable `tag = 'my-element'`.

To register `MyElement` class as a custom tag, use `document.defaultView.customElements.define()` method.

Note: Formarly, `document.registerElement` method was used to define custom tags, but in the latest specification uses `customElements.define` method to reagister custom tags. WDOM supports the same method as the latest specification.

`document.defaultView` property returns a reference to the window object, which is same as the `window` object of JavaScript on browsers.

Now you can use the registered custom tag from `document.createElement('my-element')` or `innerHTML = '<my-element></my-element>>'`. Both these methods return new instance of ``MyElement

Extended Custom Tags

WDOM supports to extend existing tags with `is` attribute.

For example, to define `MyButton` or `DefaultButton` as a custom tag:

```
import asyncio
from wdom.server import start_server, stop_server
from wdom.document import get_document
from wdom.tag import Button, Div

class MyButton(Button):
    # tag = 'button'  <- tag name is already defined in Button class
    class_ = 'btn'
    is_ = 'my-button'  # set name at is_

class DefaultButton(MyButton):
    class_ = 'btn-default'
    is_ = 'default-button'

if __name__ == '__main__':
    document = get_document()
    # Register MyElement
    document.defaultView.customElements.define('my-button', MyButton, {'extends': 'button'})
    document.defaultView.customElements.define('default-button', DefaultButton, {'extends': 'button'

    # Load css and js file for bootstrap
    document.add_cssfile('https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css')
    document.add_jsfile('https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js')
    document.add_jsfile('https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js')

    div = Div(parent=document.body)
    div.innerHTML = '''
        <button is="my-button">MyButton</button>
        <button is="default-button">DefaultButton</button>
    '''.strip()
    print(isinstance(div.firstChild, MyButton))  # True
    print(isinstance(div.lastChild, DefaultButton))  # True

    start_server()
    try:
        asyncio.get_event_loop().run_forever()
    except KeyboardInterrupt:
        stop_server()
```

On the class statements, add class variable `is_` and specify the name of the custom tag. Class variable `tag` is a tag name to be extended, but in the above example, it is already defined in `Button` class.

When registering the custom tag, pass the name (value of `is_`) at the first argument to `customElements.define` and pass dictionary which contains `'extends'` field to specify the tag name to be extended, at the third argument.

After the registration, an HTML like `<button is="my-button">` will be parsed to an instance of `MyElement`, and `<button is="default-button">` to `DefaultButton`.

Caution: Register custom tags as early as possible. If the instance was generated before registering it, it becomes different class. When the `customElements.define` is called and registerd, WDOM will try to update the class of existing instances but `__init__` will not be not called.
Additionally, changing `is` attribute of the existing instances, likely `element.setAttribute('is', '...')`, do not change its class currently.
In future, Lifecycle callback methods or silimar features will be implemented, but still it's safer to register custom tags before instanciate it.

8.1.5 Theming with CSS Frameworks

(to be written)

8.1.6 Freezing Application

(to be written)

With cx_Freeze

(to be written)

With Electron

(to be written)

8.2 API Reference

8.2.1 DOM API

(This section is not complete and not enough yet...)

`wdom.dom.Tag` class provides DOM implementation which is synchronized on python and browser.

8.2.2 Application and Server

WDOM supports two web-server library, tornado and aiohttp. tornado is required to use WDOM, but aiohttp is optional.

tornado is written in pure-python, so it works well in any environment. aiohttp is implemented with C-extention, and natively asyncio friendly.

WDOM server module is provided by `wdom.server`. If aiohttp is available, wdom will use it. This module wraps web-servers, tornado or aiohttp, so users don't need to care which server is used now.

`wdom.server.`**`add_static_path`**(*prefix*, *path*, *no_watch=False*)
Add directory to serve static files.

First argument `prefix` is a URL prefix for the `path`. `path` must be a directory. If `no_watch` is True, any change of the files in the path do not trigger restart if `--autoreload` is enabled.

wdom.server.**start_server**(*app=None*, *browser=None*, *address=None*, *check_time=500*, ***kwargs*)
Start web server.

wdom.server.**stop_server**(*server=None*)
Terminate web server.

8.2.3 Test Utilities

WDOM provides two Utility classes (`WebDriverTestCase` and `RemoteBrowserTestCase`) and some functions for running test on browser with Selenium WebDriver.

WebDriverTestCase class

`wdom.testing.WebDriverTestCase` class is designed for end-to-end UI test, which is useful for testing your app on browser. This class runs your app on subprocess and prepare WebDriver for tests.

Usage

Example code using py.test as a test runner.

```
# in your_test_dir/conftest.py
import pytest
from wdom.testing import start_webdriver, close_webdriver

@pytest.fixture(scope='session', autouse=True)
def browser(request):
    start_webdriver()  # Start WebDriver for this session
    request.addfinalizer(close_webdriver)


# in your test file
from unittest import TestCase
from wdom.misc import install_asyncio
form wdom.testing import WebDriverTestCase

def setUpModule():
    install_asyncio()  # force tornado to use asyncio

class TestApp(WebDriverTestCase, TestCase):
    def setUp(self):
        # do some setup, if need.
        super().setUp()  # MUST call base class's setUp.

    def get_app() -> wdom.server.Application:
        # Prepare and return application you want to test
        return your_app

    def test_case1(self):
        # Write your test here
        self.wd.get(self.url)  # you can access webdriver by self.wd

    def test_case2(self):
        # Write your another test here
        ...
```

RemoteBrowserTestCase Class

`RemoteBrowserTestCase` class is design to test `wdom` itself. Its features might not be so useful for library's users. `RemoteBrowserTestCase` class helps you to test your app by directly controlling `Node` object on python, from test scripts in the same process. This class is **Largely Experimental**.

`RemoteBrowserTestCase` runs application server on the same process, which is running tests, so that objects on the server can be directly controlled from test scripts. WebDriver is run on subprocess, and controlled by passing messages on pipe. This messaging process is wrapped by `RemoteBrowserTestCase` class and users don't need to care it, but owing to this architecture, not all of the features of WebDriver is available.

Usage

Example code using py.test as a test runner.

```
# in your_test_dir/conftest.py
import pytest
from wdom.testinsg import start_remote_browser, close_remote_browser

@pytest.fixture(scope='session', autouse=True)
def browser(request):
    start_remote_browser()  # Start browser process for this session
    request.addfinalizer(close_remote_browser)


# in your test file
from unittest import TestCase
from wdom.tag import Div
from wdom.document import get_document
from wdom.server import get_app
from wdom.tests.util import install_asyncio
from wdom.testing import RemoteBrowserTestCase

def setup_module():
    install_asyncio()  # force tornado to use asyncio module

class TestYourApp(RemoteBrowserTestCase, TestCase):
    def get_app(self) -> wdom.server.Application:
        # Prepare and return application you want to test
        self.root_node = Div()
        self.root_node.textContent = 'RootNode'
        self.doc = get_document()
        self.doc.body.prepend(self.root_node)
        self.app = get_app(self.doc)
        return self.app

    def test_senario1(self):
        self.set_element(self.root)  # find and set element
        # get text content of the target element
        self.assertEqual(self.text, 'RootNode')
```

For more examples, see wdom/tests/remote_browser and wdom/tests/webdriver directory.

8.3 Changes

8.3.1 Version 0.2

(next version)

Version 0.1.3 (2016-05-17)

- (bug fix) Add dependency of mypy-lang for python < 3.5

Version 0.1.2 (2016-05-15)

- `TestCase.wait` methods take two argument, `timeout` and `times`.
- Add `wait_until` method and `timeout` class variable on `TestCase`.
- Default value of `TestCase.wait_time` is same as local and travis ci. If longer wait time is required on travis, change `wait_time` on each test case.
- Support access log on aiohttp server

Version 0.1.1 (2016-05-15)

- minor update on meta data

8.3.2 Version 0.1 (2016-05-15)

First public release.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

W

A

S

W