swole Documentation

Release 0.0.1

Nicolas REMOND

Mar 01, 2023

Get Started

Swole is a framework that let you build web apps only in a few lines of Python code.

It's very similar to `streamlit`, but more performant and easier to customize.

You can find the code in this repository.

CHAPTER 1

Install

To install the latest stable version of this library, just run :

```
pip install swole
```

Warning: This package is still in Alpha version. It's a proof-of-concept, a lot of features are missing.

You can also install the bleeding-edge version, from *master* :

```
pip install git+https://github.com/astariul/swole.git
```

CHAPTER 2

Create your own app

Let's create your first app with *Swole*, step by step.

2.1 Step 1 : Add some Input Widgets

In a python file (let's name it `app.py`), import and create Widgets :

```
from swole.widgets import Input, Markdown

i_a = Input()
i_b = Input()
m = Markdown("Result : ")
```

2.2 Step 2 : Add an AJAX request

To make our app interactive, we create an AJAX request that will be called when we click on a button. This AJAX request will simply compute the result of an addition :

```
from swole.widgets import button
from swole import ajax

@ajax(i_a, i_b)
def addition(a, b):
    res = a + b
    m.set("Result : {}".format(res))

Button("Compute", onclick=addition)
```

2.3 Step 3 : Serve your app

Finally add the code to serve the app in the `main` :

```
from swole import Application

if __name__ == "__main__":
    Application().serve()
```

2.4 Step 4 : Start your app

Now that the code is written, you can start your app with :

You can visit 127.0.0.1:8000 and see your app running !

CHAPTER 3

Get started with Examples

The easiest way to get started is through examples.

All examples are self-contained, ready-to-deploy. You can run any example with :

```
python examples/<example_name>.py
```

3.1 basic_addition.py

Minimal example with a basic AJAX request, doing addition between two numbers.
If the result is *42*, also update the input fields to be *42* as well.

CHAPTER 4

How it works ?

Swole tries to be as transparent as possible. Applications are made with *HTML*, *CSS*, and *Javascript*.

Note: You can always use the "inspect" tool of your browser on a Swole application to view the code.

4.1 Core architecture

An application is represented by the `Application` object. It contains references to one or several `Page` object.

Each `Page` has a unique route. The `Page` is made of several `Widget` object. These `Widget` are the basic building blocs, used to create the content of each page.

Each type of `Widget` is independently built into HTML. The HTML generated for each `Page` is just the addition of HTML from each of its `Widget`.

4.2 Styling

Styling is done using CSS, through the `Skin` object.

`Skin` is just a bunch of CSS rules. Each `Page` is assigned a `Skin`, and the styling of the elements of the `Page` is done using the rules from the `Skin`.

By decoupling style and content, changing the style of the page is as simple as using another `Skin`.

4.3 Server-client interaction

For server-client interaction, Swole uses **AJAX requests**.

AJAX requests allow the web application to send and receive data from the server in the background. This allow the web page to change dynamically without the need to reload the entire page.

By using AJAX requests, we also have better control of what is send and when, as developer.

AJAX requests are represented by the class *Ajax*, and can be easily declared using the decorator `ajax`.

CHAPTER 5

Make you own Skin

As mentioned in *Styling*, you can change the skin of your webpage by simply specifying the path to the skin to use.

This means you can create your own `Skin` !

5.1 Structure of the Skin file

A `Skin` file is simply a CSS file.

But the content is devided in 3 parts :

- External CSS
- External Fonts
- Custom rules

External CSS and **External Fonts** are specified as commented links, one link per line. For example :

```
/* https://fonts.googleapis.com/my_font */
```

These links will automatically be included in the final CSS.

Finally, **Custom rules** are specified as normal CSS. For example :

```
body {
    text-align: center;
}
```

Each part is divided by an empty line.

So in the end, your custom `Skin` file may look like :

```
/* https://cdn./my/imported/css */

/* https://fonts.googleapis.com/my_font */
```

(continues on next page)

(continued from previous page)

```
body {
    text-align: center;
}
```

5.2 Use your own Skin

Create your *Skin* file following the structure mentioned in previous section.

Note: You can create a copy of the base skin (`swole/skins/base.css`) as starting point, it's easier !

Then, when you create your page, specify the path to your custom *Skin* :

```
Page(skin_path="path/to/my/custom.css")
```

CHAPTER 6

Make you own Widget

Swole is bundled with some general-use *Widget*. But sometimes you need more than that !

To fit your specific needs, you can create a custom *Widget*.

Let's see together in this guide how to create a custom *Widget*. We will create a Widget making two buttons.

6.1 Step 1 : sub-class Widget

Let's create a sub-class of *Widget* :

```
from swole.widgets import Widget

class TwoButtons(Widget):
    pass
```

Let's change the constructor to take two strings (the text of each button) :

```
class TwoButtons(Widget):
    def __init__(self, text1="Button 1", text2="Button 2", **kwargs):
        super().__init__(**kwargs)
        self.text1 = text1
        self.text2 = text2
```

6.2 Step 2 : create HTML

Now we have to overwrite the method *html()*, where the HTML of the *Widget* is defined :

```
from dominate.tags import button, div

class TwoButtons(Widget):
```

(continues on next page)

(continued from previous page)

```
    def html(self):
        attributes = {"id": self.id}        # Best practice : Always add the ID of
↪the Widget
        self.add_css_class(attributes)      # Method from super to add custom CSS
↪classes

        d = div(**attributes)
        with d:
            button(self.text1)
            button(self.text2)
        return d
```

Warning: You should use the dominate library to create the HTML object returned by `html()`.

6.3 Step 3 : define setter and getter

setter and *getter* are methods used by *Javascript* for the AJAX requests. It's the gateway between *Python* and *Javascript*.

In our case it's simple :

```
class TwoButtons(Widget):
    def get(self):
        return [self.text1, self.text2]

    def set(self, x):
        self.text1 = x[0]
        self.text2 = x[1]
```

6.4 Step 4 : define AJAX (optional)

Optionally, your `Widget` can also overwrite the method `ajax()`. This method is used to retrieve AJAX requests linked to this `Widget`. It should return a `Ajax` object.

You can refer the implementation of `Button` for an example.

6.5 Step 5 : use your widget

You're all done ! Now you just have to use your `Widget` is one of your web application !

```
TwoButtons(text2="My 2 buttons")

if __name__ == "__main__":
    Application().serve()
```

CHAPTER 7

Widgets

7.1 Display

class Header(*text='Header'*, *level=2*, *center=False*, ***kwargs*)
Widget to create a header.

Parameters

- **text** (*str*, optional) – Text of the header. Defaults to *Header*.
- **level** (*int*, optional) – Level of the header. *1* to *6* only. Defaults to *2*.
- **center** (*bool*, optional) – Wether to put this header in the middle or not. Defaults to *False*.

class Title(**args*, ***kwargs*)
Widget to create a title.

A title is a Header with level *1*.

class SubHeader(**args*, ***kwargs*)
Widget to create a sub-header.

A sub-header is a Header with level *3*.

class Markdown(*content=''*, ***kwargs*)
A general widget to write Markdown.

Parameters **content** (*str*, optional) – Markdown content. Defaults to empty string.

7.2 Interactivity

class Input(*label=None*, **args*, ***kwargs*)
Widget to create an input.

Parameters

- **type** (*str*, optional) – Type of the input. Defaults to *number*.

- **placeholder** (*str*, placeholder) – Placeholder for the input. If *None*, no placeholder is used. Defaults to *None*.
- **label** (*str*) – Label to give to the Widget.

class Button(*text='Button'*, *primary=False*, *onclick=None*, ***kwargs*)

Widget to create a button.

Parameters

- **text** (*str*, optional) – Text of the button. Defaults to *Button*.
- **primary** (*bool*, optional) – Wether this button is primary or not. Defaults to *False*.
- **onclick** (*Ajax*, optional) – Ajax request to call if the button is clicked. If *None*, nothing happend on click. Defaults to *None*.

7.3 Base

class Widget(*cls=None*)

Base class for all Widgets.

jquery_fn

The name of the JQuery function to use to get the value of the widget from the HTML page. Defaults to *text*.

Type str

cls

List of CSS classes to apply to this widget.

Type list of str

Parameters **cls** (*str or list of str, optional*) – Class(es) to add to the Widget. Can be a single class (*str*) or several classes (*list of str*). If *None* is given, no additional class is added. Defaults to *None*.

add_css_class(*attr*)

Utils method to add the class attribute in the given dictionary. This dictionary can then be used in *dominate*.

Parameters **attr** (*dict*) – The attributes dictionary to be used in *dominate* tag class.

ajax()

Method to get the Ajax request (if any) of the widget. Or *None* if there is no Ajax call for this widget.

Returns Ajax request. *None* if there is no Ajax request.

Return type *Ajax*

get()

Method to get the current value of the widget.

Note: This method should be overwritten.

Returns Current value of the widget.

Return type object

get_str()
Method to get the current value of the widget, as a string.

Returns Current value of the widget.

Return type str

html()
Method to get the *dominate* HTML of the widget. This HTML needs to be rendered.

Note: This method should be overwritten.

Returns HTML document corresponding to the widget.

Return type dominate.document

set(*x*)
Method to set the current value of the widget.

Note: This method should be overwritten.

Parameters **x** (*str*) – Value of the widget to set.

class WideWidget(*wide=False*, **args*, ***kwargs*)
Class for Widgets that can be wide.

Parameters **wide** (*bool, optional*) – If set to *True*, the widget will take all the available width. Defaults to *False*.

CHAPTER 8

Swole

8.1 Application

SWOLE_CACHE = '~/.cache/swole'
Default directory to use for caching.

class Application
Class representing an application. Application are used to serve declared pages.

fapi
FastAPI app.

Type *fastapi.FastAPI*

serve (*folder='~/.cache/swole'*, *host='0.0.0.0'*, *port=8000*, *log_level='info'*)
Method to fire up the FastAPI server !

Parameters

- **folder** (*str*, optional) – Folder where to save HTML files. Defaults to *SWOLE_CACHE*.
- **host** (*str*, optional) – Run FastAPI on this host. Defaults to *0.0.0.0*.
- **port** (*int*, optional) – Run FastAPI on this port. Defaults to *8000*.
- **log_level** (*str*, optional) – Log level to use for FastAPI. Can be [*critical*, *error*, *warning*, *info*, *debug*, *trace*]. Defaults to *info*.

8.2 Page

HOME_ROUTE = '/'
Default route for the Home page.

DEFAULT_FAVICON = Doge
Default favicon (Doge).

class Page(*route='/'*, *skin='base'*, *skin_path=None*, *title='Home'*, *favicon='https://user-images.githubusercontent.com/22237185/95144545-e35d1200-07b3-11eb-9216-362b2a19c9aa.png'*)

Class representing a page.

Parameters

- **route** (*str*, optional) – The route to access this page. Defaults to *HOME_ROUTE*.
- **skin** (*str*, optional) – The name of the skin to use for this page. If *None* is given, no skin is loaded. Defaults to *base*.
- **skin_path** (*str*, optional) – The path of the Skin file to use. If different than *None*, the *skin* argument is ignored, and this file is used instead. Useful to provide custom Skin file. Defaults to *None*.
- **title** (*str*, optional) – The title of the page. Defaults to *Home*.
- **favicon** (*str*, optional) – The path to the favicon to use for this page. Defaults to *DEFAULT_FAVICON*.

__enter__()

Context manager for easy definition of Widgets inside the page : Remember the declared widgets at this point.

__exit__(*type*, *value*, *traceback*)

Context manager for easy definition of Widgets inside the page : Add any newly declared widgets.

add(*widget*)

Method to add a widget to this page.

Parameters **widget** (*Widget*) – Widget to add.

8.3 Ajax

class Ajax(*callback*, *inputs*)

Class representing an AJAX request. It is used as callback to update the webpage dynamically.

Constructor : Make the AJAX from the function to execute, and define the inputs as given.

Parameters

- **callback** (*callable*) – Function to execute when the AJAX is called.
- **inputs** (*list of Widget*) – Inputs to the callback.

__call__(*page*, *input_data*)

Main method, being called by the application with the right inputs. This method keep track of the value of each widget of the page, and based on what was changed, return only the element to change in the page.

Parameters

- **page** (*Page*) – Page calling the AJAX.
- **input_data** (*dict*) – Inputs data retrieved from the page after the AJAX request was triggered. It's a dict of *str* -> *str* where the key is the ID of the widget and the value is the value of the widget.

Returns Dictionary of ID to value, containing all widgets to update.

Return type dict

js()
Method writing Javascript equivalent for this AJAX request.

Returns Javascript equivalent.

Return type *str*

8.4 Skin

class Skin(*name='base'*, *path=None*)
Class representing the skin to use for the page. A skin is basically a CSS file, with additional imports for CSS libraries and fonts.

path
Path of the CSS file for this skin.

Type str

libs
External CSS libraries to import additionally.

Type list of str

fonts
External fonts to import additionally.

Type list of str

rules
Custom rules as defined in the Skin file.

Type str

Constructor : Try to locate the file, then read it to extract external CSS resources links.

Parameters

- **name** (*str, optional*) – The name of the skin to use. Defaults to *base*.
- **path** (*str, optional*) – If different than *None*, the path of the CSS file to use. If different than *None*, *name* argument is ignored. Useful to provide custom skin. Defaults to *None*.

8.5 utils

route_to_filename(*route*)
Function to transform a route to a filename.

Parameters **route** (*str*) – Route name to transform to a filename.

Returns Filename corresponding to the route.

Return type *str*

CHAPTER 9

FAQ

Q. Why using `swole` ? Why not `streamlit` ?

Don't get me wrong, `streamlit` is an awesome framework. `swole` just try to fix a few problematic issues of `streamlit` :

- It uses Flask, which is outdated and not performant when compared to FastAPI
- No customization possible
- No control over user's interaction with the page
- Not transparent
- No Doge

Q. How *Swole*'s backend and frontend communicate ?

Unlike `streamlit`, which use a system of cache and reload the page everytime someone interact with it, `swole` uses AJAX requests to update the frontend.

Reloading the page every interaction is very inefficient, and irritating for the user's experience.

Using AJAX instead makes the whole process almost invisible for the user, and everything is more clear for the developper, because we know what data is sent when.

Q. Why do you say *Swole* is *transparent* but *Streamlit* is *opaque* ?

On a `swole` page, try to "view the page source" (right-click).
Now do the same on a `streamlit` page, and compare.

Q. Why this name ?

It all comes from a meme :


MY BACKEND
SKILL


MY FRONTEND
SKILL

CHAPTER 10

Contribute

Note: The library is in alpha version, any help is greatly appreciated, as well as ideas !

To contribute, simply fork the repository, clone it locally, install the library and create your own branch :

```
git clone https://github.com/astariul/swole.git
cd swole
pip install -e .
git checkout -b my_branch
```

Add your *dogesome* code !

Note: Don't forget to update tests and documentation as well !

Check if your code is well-formated :

```
pip install flake8

flake8 . --count --max-complexity=10 --max-line-length=127 --statistics --per-file-
↪ignores="__init__.py:F401"
```

Ensure tests are passing :

```
pip install pytest

python -m pytest -W ignore::DeprecationWarning
```

Then push your changes in your fork and submit your PR !

Index