



# IDOM - It's React, but in Python

[IDOM](#) is a new declarative Python package for building highly interactive and composable user interfaces.



IDOM takes inspiration from [React](#), and wherever possible, attempts to achieve parity with the features it copies. One of the most notable features is the version of React's often lauded "[Hooks](#)" that IDOM implements in Python.

At a glance, the similarities between IDOM and React are rather striking. Below is a React component which displays a slide that updates when a user clicks on it. Immediately following that is the same view implemented in Python using IDOM.

```
import React, { useState } from react;

function Slideshow() {
  const [index, setIndex] = useState(0);
  return (
```

```

    <img
      src={ `https://picsum.photos/400?image=${index}` }
      onClick={ () => setIndex(index + 1) }
      style={ {cursor: "pointer"} }
    />
  )
}

```

```

import idom

@idom.component
def Slideshow():
    index, set_index = idom.hooks.use_state(0)
    return idom.html.img(
        {
            "src": f"https://picsum.photos/400?image={index}",
            "onClick": lambda event: set_index(index + 1),
            "style": {"cursor": "pointer"},
        }
    )

idom.run(Slideshow)

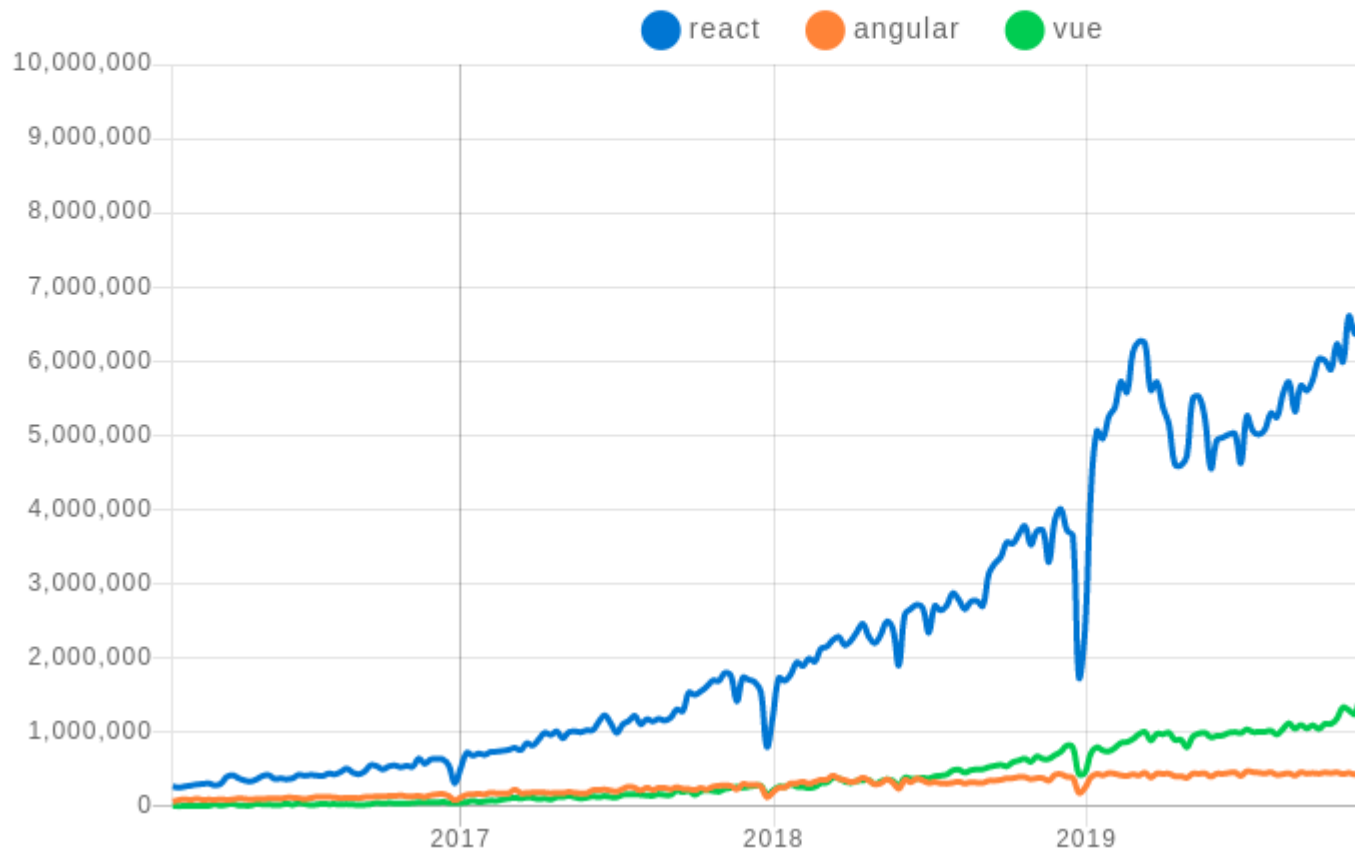
```

## Why Do We Need IDOM?

Over the [past 5 years](#) front-end developers seem to have arrived at the conclusion that declarative programming is more simply, mutable state in programs can quickly lead to unsustainable complexity. This trend is largely evident in frameworks like [Vue](#) and [React](#) which describe the logic of computations without explicitly stating their control flow.

react x angular x vue x + @angular/core + ember-source

## Downloads in past 5 Years ▾



So what does this have to do with Python and IDOM? Well, because browsers are the de facto "operating system" for web applications, frameworks like Python have had to figure out clever ways to integrate with them. While standard [REST](#) APIs are well suited for server-side applications, modern browser users expect a higher degree of interactivity than this alone can achieve.

A variety of Python packages have since been created to help solve this problem:

- [IPyWidgets](#) - Adds interactive widgets to [Jupyter Notebooks](#)
- [Dash](#) - Allows data scientists to produce enterprise-ready analytic apps
- [Streamlit](#) - Turns simple Python scripts into interactive dashboards
- [Bokeh](#) - An interactive visualization library for modern web browsers

However they each have drawbacks that can make them difficult to use.

1. **Restrictive ecosystems** - UI components developed for one framework cannot be easily ported to any of the others, or are complex, undocumented, or are structurally inaccessible.

2. **Imperative paradigm** - IPyWidgets and Bokeh have not embraced the same declarative design principles as React or Vue, and Dash on the otherhand, are declarative, but fall short of the features provided by React or Vue.
3. **Limited layouts** - At their initial inception, the developers of these libraries were driven by the visualization of data, and creating complex UI layouts may not have been a primary engineering goal.

A future article will address specific comparisons to each of the projects mentioned above, but for now, we'll focus on the problems above.

## Ecosystem Independence

IDOM has a flexible set of core abstractions that allow it to easily interface with its peers. At the time of writing, IDOM supported (Streamlit and Bokeh are in the works):

- [idom-jupyter](#) (try it now with [Binder](#))
- [idom-dash](#)

By providing well defined interfaces and straightforward protocols, IDOM makes it easy to swap out any part of the ecosystem you need to. For example, if you need to use a different web server for your application, IDOM already has 3 options, or you can create your own.

- [Sanic](#)
- [Flask](#)
- [Tornado](#)

You can even target your usage of IDOM in your production-grade applications with IDOM's Javascript [React client](#) and connect a back-end websocket that's serving up IDOM models. IDOM's own [documentation](#) acts as a primer, and the page is static HTML, but embedded in it are interactive examples that feature live views being served from the server.

# The Game Snake

Click to start playing and use the arrow keys to move 🎮

Slow internet may cause inconsistent frame pacing 😊

Python Code

Live Example

```
import asyncio
import enum
import random
import time

import idom

class GameState(enum.Enum):
    init = 0
    lost = 1
    won = 2
    play = 3
```

## Declarative Components

IDOM, by adopting the hook design pattern from React, inherits many of its aesthetic and functional characteristics. Its declarative interfaces are composed of basic [HTML elements](#) that are constructed and returned by special functions called hooks, those components can be made to have state. Consider the component below which returns two buttons.

```
import idom

@idom.component
def OnOff():
    state, set_state = idom.hooks.use_state(False)
    return idom.html.div(
        idom.html.button({"onClick": lambda event: set_state(True), "On"},
        idom.html.button({"onClick": lambda event: set_state(False), "Off"},
        idom.html.p("The button is " + ("on" if state else "off")),
    )
```



The button is off

Here's a very high level summary of how it works... the first time a view of the component above is rendered `state` is `False`. The function then returns a series of HTML elements with callbacks that respond to client-side events. The component then realizes that declaration and displays two buttons with the text "The button is off". Then, when a user clicks the "On" button, an event is triggered, the associated callback responds to it by setting the `state` to `True`, and a re-render of the component's machinery again goes to work to update the display, this time though, the text will read "The button is on".

Nowhere in the example above does the code describe how to evolve the frontend view when events occur. In a declarative approach, this is how it should look. It's then IDOM's responsibility to figure out how to make that happen. This behavior, where the code describes the state of the UI and the framework figures out how to achieve them, is what makes components in IDOM and React "declarative". For comparison, an imperative approach to defining the same interface might look similar to the following:

```
layout = Layout()

def on_off():
    state_text = html.p(children="The button is off")

    def set_on(event):
        state_text.update(children="The button is on")

    def set_off(event):
        state_text.update(children="The button is off")

    return html.div(
        html.button(on_click=set_on, children="On"),
        html.button(on_click=set_off, children="Off"),
        state_text,
    )

layout.add_element(on_off())
```

In this imperative incarnation, we must explicitly state how the `count_button` updates `count_text` via its `on_click` callback. In a declarative approach, `state` is mutated by amending its `'count'` - a side effect which can be avoided with hooks.

It's important to note that neither declarative nor imperative design principle are inherently better in all circumstances. Asserting the way a view should look is easier than describing how it should come to look that way.

## Flexible Layouts

Constructing complex layouts is also made easier when done declaratively because the elements, state, and logic are separated. In the `OnOff` component shown above, code responsible for managing business logic and manipulating state is separated from the code responsible for structuring the elements of the layout. The great advantage of this approach is that these separate functions if either the logic or the structure becomes too complex:

```
@idom.component
def OnOff():
    return on_off_buttons(*use_on_off_state())

def use_on_off_state():
    """manage logic and state"""
    state, set_state = idom.hooks.use_state(False)

    def set_on():
        set_state(True)

    def set_off():
        set_state(False)

    return state, set_on, set_off

def on_off_buttons(state, set_on, set_off):
    """define element structure"""
    return idom.html.div(
        idom.html.button({"onClick": lambda event: set_on(), "On"},
        idom.html.button({"onClick": lambda event: set_off(), "Off"},
        idom.html.p("The button is " + ("on" if state else "off")),
    )
```

While the refactoring above is overkill in such a simple case, attempting something similar with the earlier implementation is not straightforward because callbacks responsible for defining business logic must hold a reference to the element. The description of the layout in code is often muddled by semantic limitations of the business logic that make it difficult to refactor and grows old.

## Conclusion

Building highly interactive web applications as a Python developer has historically been a great challenge. However, by combining HTML, CSS, and Python, you can make everything from [slideshows](#) to [dashboards](#) and use it wherever you need it in an existing web application.

To learn more check it out:

- [installation instructions](#)
- [where to get started](#)
- [interactive examples](#)



- and much more!