



# 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 demonstrates a feature that updates when a user clicks on it:

```
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"} }
    />
  )
}

```

And this is the same component implemented in Python using IDOM:

```

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)

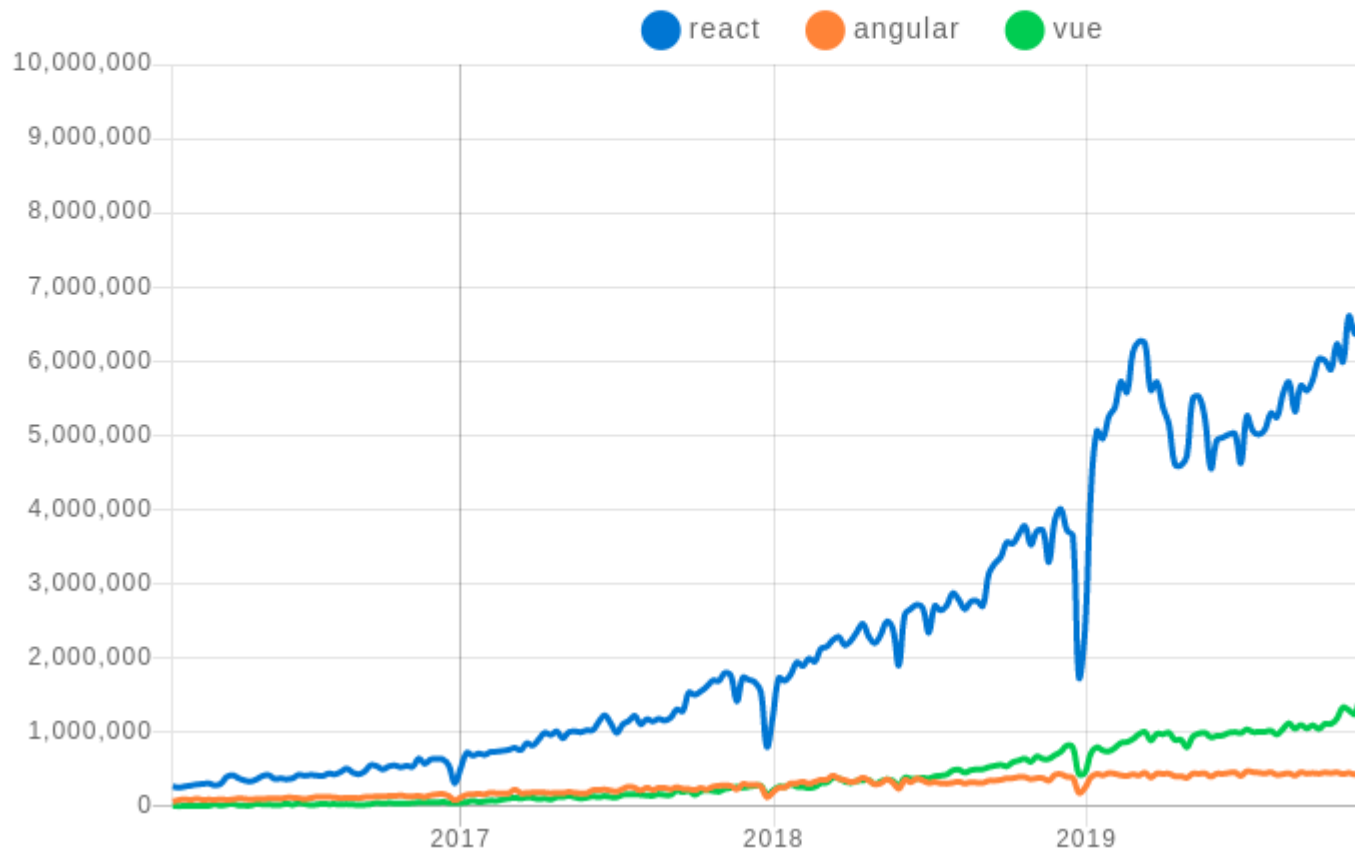
```

## 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's integration with 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 for this, as 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 component 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 displays a basic

```
import idom

@idom.component
def AndGate():
    input_1, toggle_1 = use_toggle()
    input_2, toggle_2 = use_toggle()
    return idom.html.div(
        idom.html.input(
            {"type": "checkbox", "onClick": lambda event: toggle_1()}
        ),
        idom.html.input(
            {"type": "checkbox", "onClick": lambda event: toggle_2()}
        ),
        idom.html.pre(f"{input_1} AND {input_2} = {input_1 and input_2}"),
```

```

    )

    def use_toggle():
        state, set_state = idom.hooks.use_state(False)

        def toggle_state():
            set_state(lambda old_state: not old_state)

        return state, toggle_state

    idom.run(AndGate)

```



False AND False = False

Here's a very high level summary of how it works... the first time a view of the component above is rendered, the `state` for `input_1` and `input_2` is `False`. The function then returns a series of HTML elements with callbacks. The machinery behind the scenes subsequently realizes that declaration and displays two checkbox buttons with labels. When a user clicks the now visible checkbox buttons, client-side events are triggered, the associated callbacks are called, `False` to `True`, and a re-render of the component is scheduled. When re-rendering, the function is again called, and the `input_1` and `input_2` have been updated to reflect the new `state`, thus causing the displayed text to change.

Consider the fact that nowhere in the example above does the code describe how to evolve the frontend view given a particular state, this is how the view should look. It's then IDOM's responsibility to figure out how to build the view. The separation of defining outcomes without stating the means by which to achieve them is what makes components in IDOM declarative. A more hypothetical, and a more imperative approach to defining the same interface might look similar to the following:

```

layout = Layout()

def and_gate():
    state = {"input_1": False, "input_2": False}
    output_text = html.pre()
    update_and_gate_output_text(output_text, state)

    def toggle_input(index):

```

```

state[f"input_{index}"] = not state[f"input_{index}"]
update_and_gate_output_text(output_text, state)

return html.div(
    html.input(
        {"type": "checkbox", "onClick": lambda event: toggle_input(1)}
    ),
    html.input(
        {"type": "checkbox", "onClick": lambda event: toggle_input(2)}
    ),
    output_text
)

def update_and_gate_output_text(text, state):
    text.update(
        children="{input_1} AND {input_2} = {output}".format(
            input_1=state["input_1"],
            input_2=state["input_2"],
            output=state["input_1"] and state["input_2"],
        )
    )

layout.add_element(and_gate())
layout.run()

```

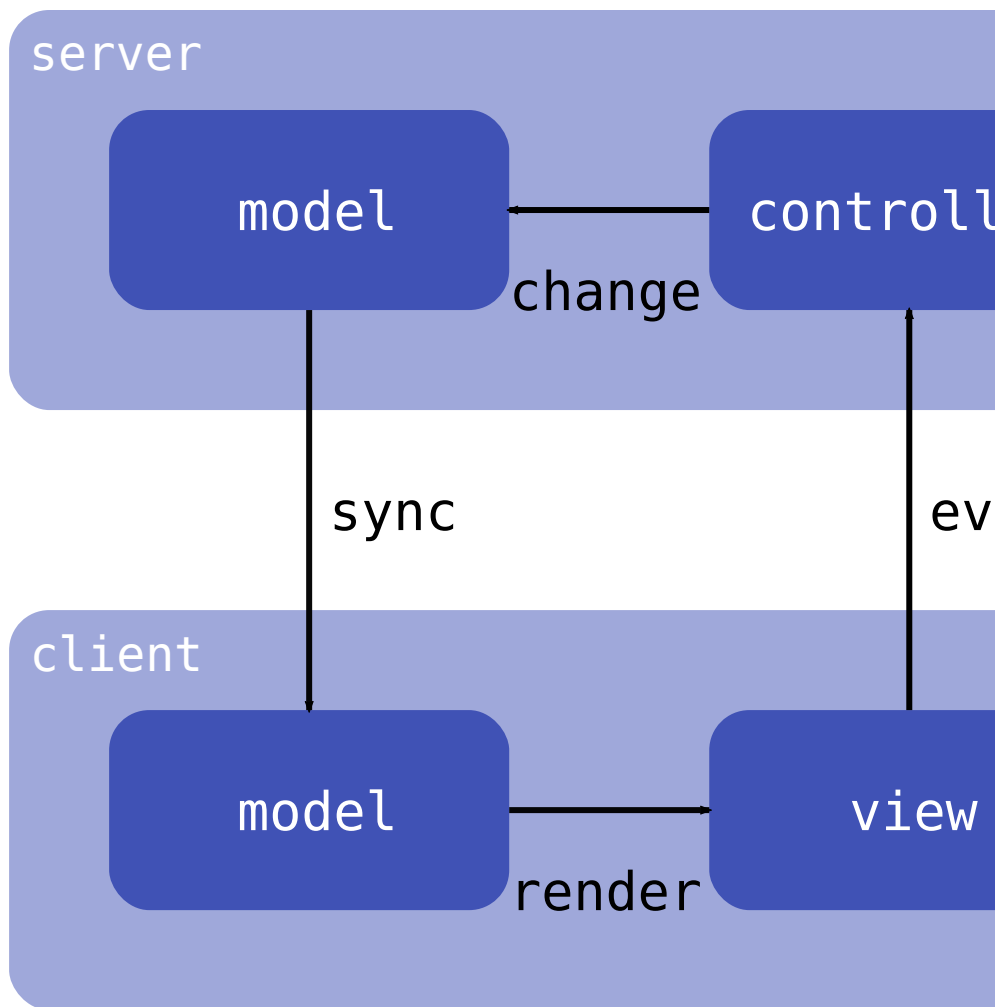
In this imperative incarnation there are several disadvantages:

1. **Refactoring is difficult** - Functions are much more specialized to their particular usages in `and_gate`. By its implementation could be applicable to any scenario where boolean indicators should be toggled on and off.
2. **No clear static relations** - There is no one section of code in which one can discern the basic structure and flow, as exemplified by the fact that we must call `update_and_gate_output_text` from two different locations. Once in the body of the callback `toggle_input`. This means that, to understand what the `output_text` might contain, one must follow the logic that surrounds it.
3. **Referential links cause complexity** - To evolve the view, various callbacks must hold references to all the state, this makes writing programs difficult since elements must be passed up and down the call stack wherever they are used, though, it also means that a function layers down in the call stack can accidentally or intentionally impact the program.

## Virtual Document Object Model

To communicate between their back-end Python servers and Javascript clients, IDOM's peers take an approach called the [View-Controller](#) design pattern - the controller lives server-side (though not always), the model is what's synthesized, and the view is run client-side in Javascript. To draw it out might look something like this:





By contrast, IDOM uses something called a Virtual Document Object Model ([VDOM](#)) to construct a representation of the DOM on the Python side by components then, as it evolves, IDOM's layout computes VDOM-diffs and wires them to its



This process, in addition to drastically reducing complexity, means that Python developers with just a little bit of knowledge can create elaborate interfaces because they have complete control over the view. Of course many users probably don't create low-level components, but for those who do it's easy to distribute their creations for others to use.

## Custom Javascript Components

Now, if you're thinking about IDOM use of a VDOM critically you may have thought...

Isn't wiring a representation of the view to the client, even if it's diffed, expensive?

And yes, while the performance of IDOM is sufficient for most use cases, there are inevitably scenarios where it's not. Just like its peers, IDOM makes it possible to seamlessly integrate [Javascript components](#). They can be custom-built to leverage the existing Javascript ecosystem without any extra work:

```
import json
import idom

material_ui = idom.install("@material-ui/core", fallback="loading...")

@idom.component
def DisplaySliderEvents():
```

```

event, set_event = idom.hooks.use_state(None)
return idom.html.div(
    material_ui.Slider(
        {
            "color": "primary",
            "step": 10,
            "min": 0,
            "max": 100,
            "defaultValue": 50,
            "valueLabelDisplay": "auto",
            "onChange": lambda *event: set_event(event),
        }
    ),
    idom.html.pre(json.dumps(event, indent=2)),
)

idom.run(DisplaySliderEvents)

```



```

[
    {
        "isTrusted": true
    },
    0
]

```

## 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, even within an existing web application.

To learn more check out:

- [installation instructions](#)
- [where to get started](#)
- [interactive examples](#)
- [and much more!](#)