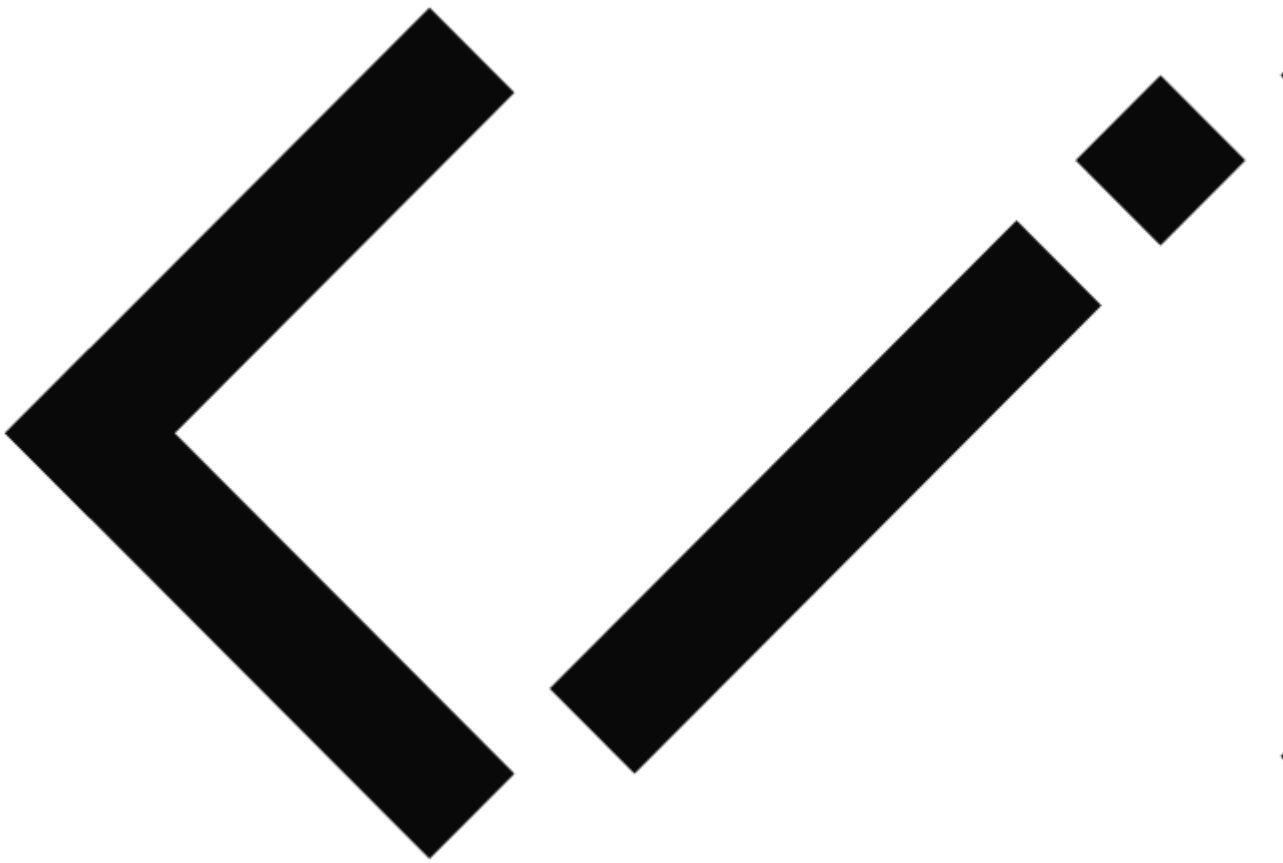
IDOM - It's React, but in Python

IDOM is a new declarative Python package for building highly interactive user interfaces.



IDOM takes inspiration from React, and wherever possible, attempts to achieve parity with the features it cop
than 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 d
of times a button has been clicked:

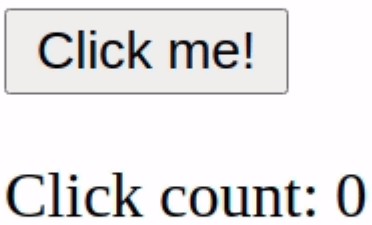
```
import React, { useState } from "react";
import ReactDOM from "react-dom";

function Counter() {
  const [count, setCount] = useState(0);
```

```
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Click me!</button>
      <p>{`Click count: ${count}`}</p>
    </div>
  );
}

ReactDOM.render(<Counter />, document.getElementById("root"));
```

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

```
import idom

@idom.component
def Counter():
    count, set_count = idom.hooks.use_state(0)
    return idom.html.div(
        idom.html.button(
            {"onClick": lambda event: set_count(count + 1)},
            "Click me!"
        ),
        idom.html.p(f"Click count: {count}")
    )

idom.run(Counter)
```

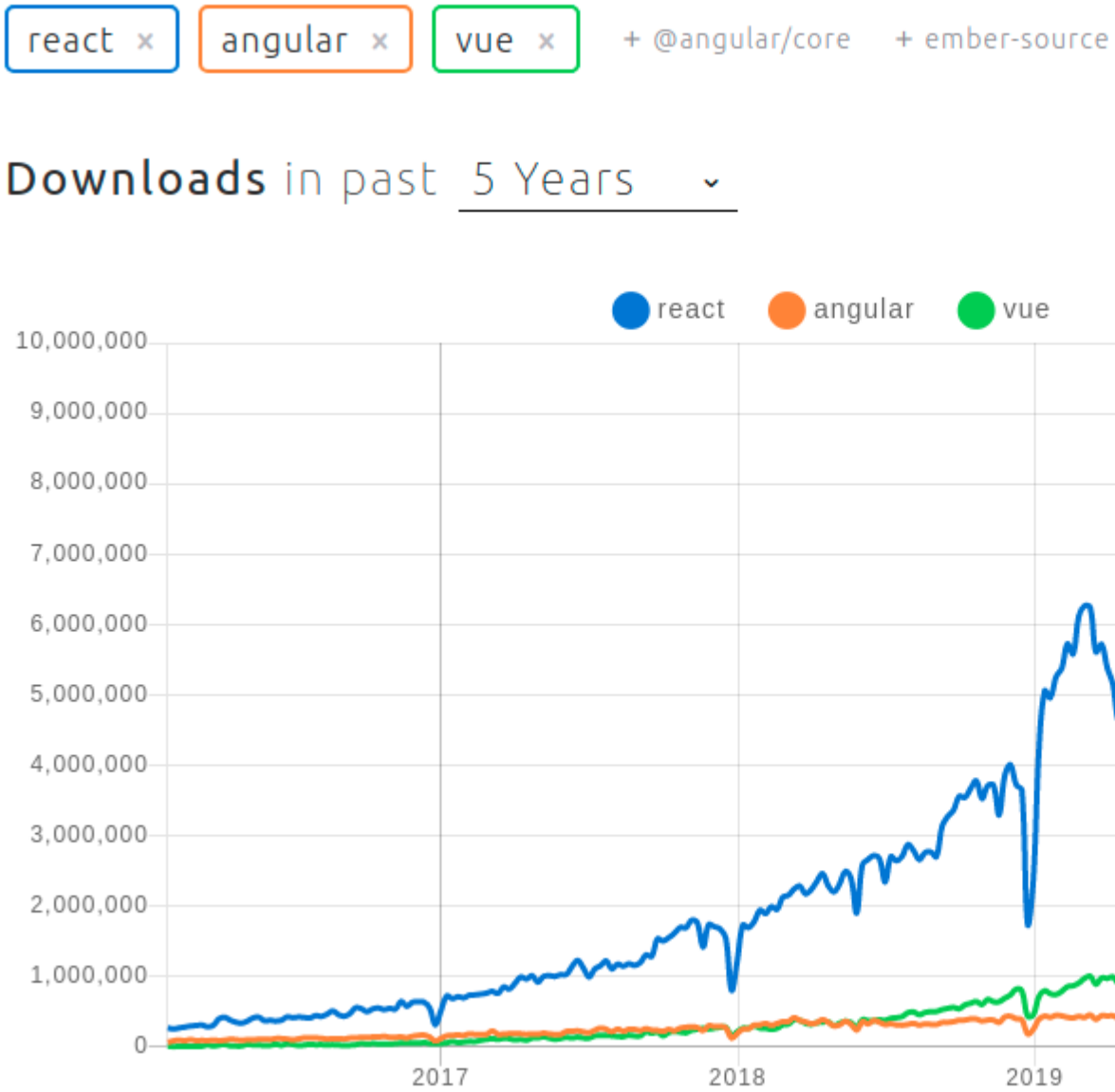
Which, when displayed in your browser, should look something like this:



Why Do We Need IDOM?

Over the past 5 years front-end developers seem to have concluded that programs written with a declarative
understand and maintain than those done imperatively. Put more simply, mutable state in programs can quic

trend is largely evidenced by the rise of Javascript frameworks like Vue and React which describe the logic of control flow.



So what does this have to do with Python and IDOM? Well, because browsers are the de facto "operating syst like Python have had to figure out clever ways to integrate with them. While standard REST APIs are well suit 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 produces 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 complex, undocumented, or are structurally inaccesible.
2. **Imperative paradigm** - IPyWidgets and Bokeh have not embraced the same declarative design principles p 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 visualizatio create 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'l problems above.

Ecosystem Independence

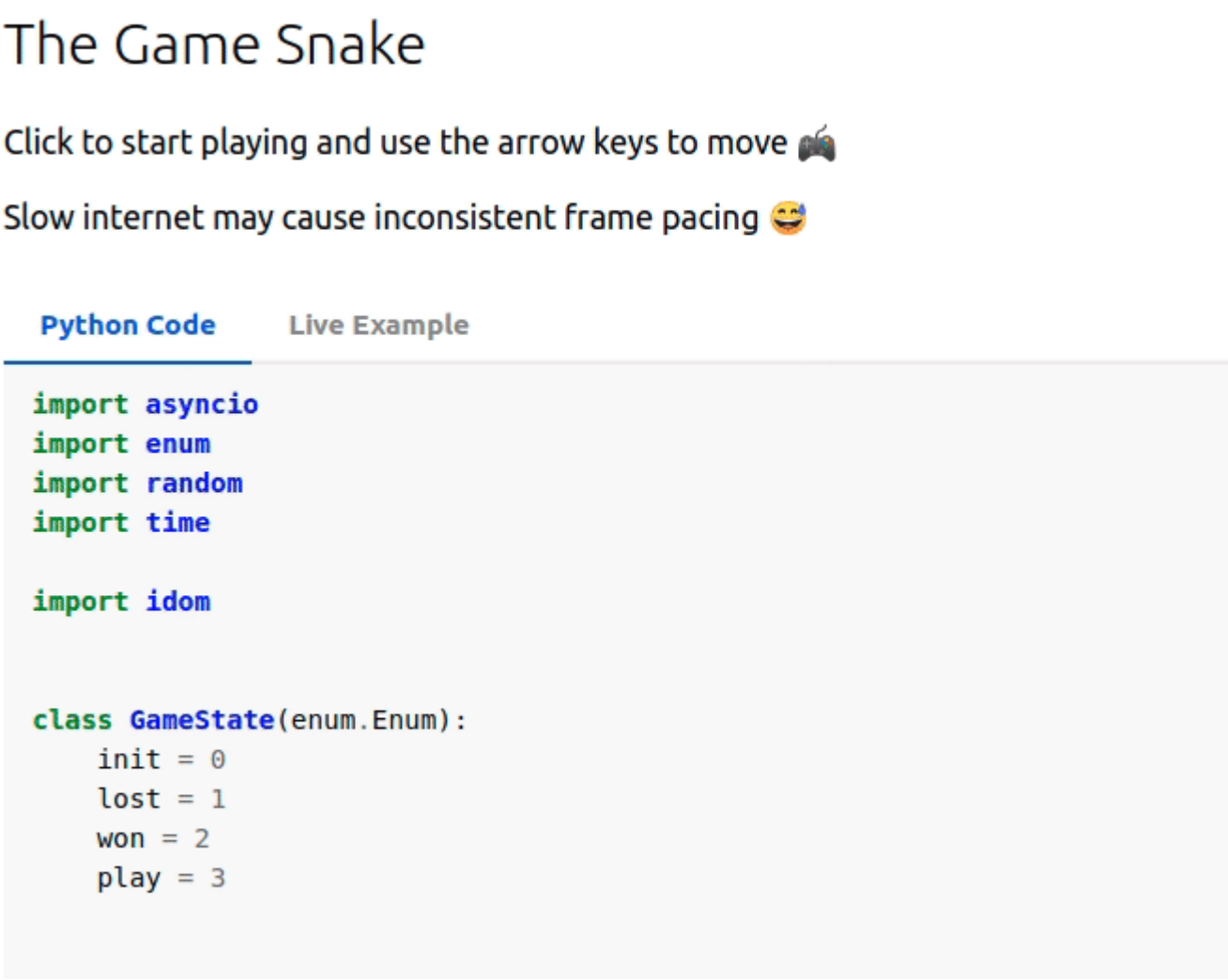
IDOM has a flexible set of core abstractions that allow it to interface with its peers. At the time of writing, bot Streamlit and Bokeh are in the works:

- idom-jupyter (try it now with Binder)
- idom-dash

By providing well defined interfaces and straighforward protocols, IDOM makes it easy to swap out any part you want to. For example, if you need a different web server for your application, IDOM already has 3 options your own:

- Sanic
- Flask
- Tornado

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



Declarative Components

IDOM, by adopting the hook design pattern from React, inherits many of its aesthetic and functional character
interfaces are composed of basic HTML elements that are constructed and returned by special functions call
hooks, those component functions can be made to have state. Consider the component below which displays

```
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)
```

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

In the code above, consider the fact that it never explicitely describes how to evolve the frontend view when particular state, this is how the view should look. It's then IDOM's responsibility to figure out how to bring tha defining outcomes without stating the means by which to achieve them is what makes components in IDOM hypothetical, and a more imperative approach to defining the same interface might look similar to the follow

```
layout = Layout()

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

    def toggle_input(index):
```

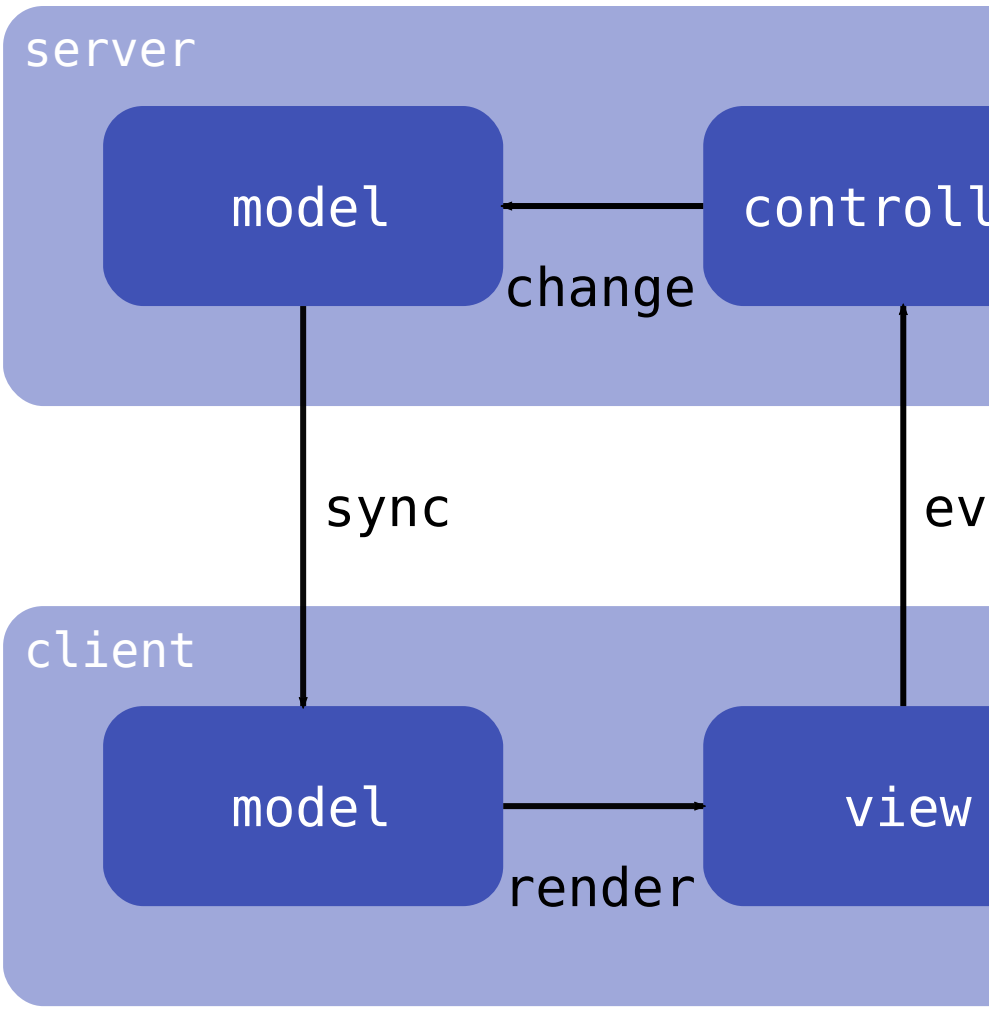
```
        state[f"input_{index}"] = not state[f"input_{index}"]
        update_output_text(output_text, state)

    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_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(make_and_gate())
layout.run()
```

In this imperative incarnation there are several disadvantages:

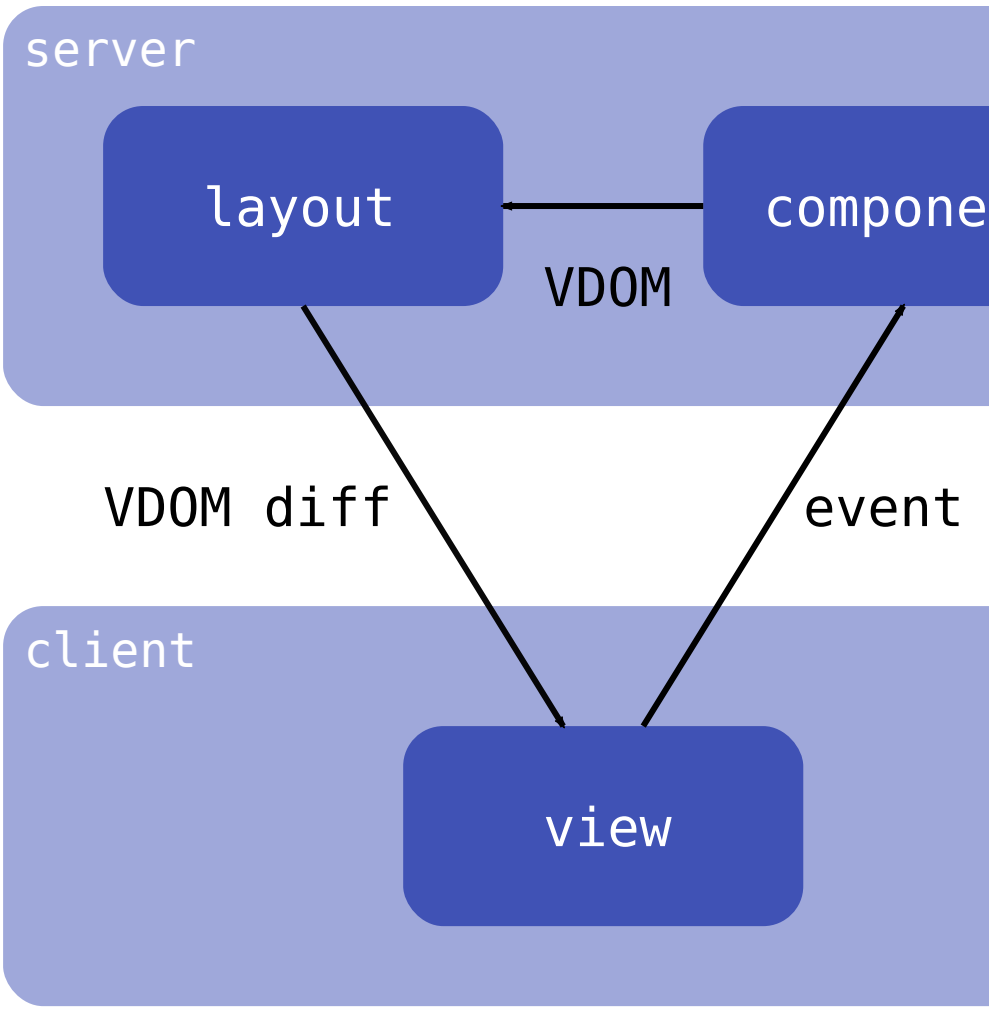
1. **Refactoring is difficult** - Functions are much more specialized to their particular usages in `make_and_gat` comparison, `use_toggle` from the declarative implementation could be applicable to any scenario where
2. **No clear static relations** - There is no one section of code through which one can discern the basic struc exemplified by the fact that we must call `update_output_text` from two different locations. Once in the of the callback `toggle_input`. This means that, to understand what the `output_text` might contain, we that surrounds it.
3. **Referential links cause complexity** - To evolve the view, various callbacks must hold references to all the this makes writing programs difficult since elements must be passed up and down the call stack wherev though, it also means that a function layers down in the call stack can accidentally or intentionally impac the program.

Virtual Document Object Model

To communicate between their back-end Python servers and Javascript clients, IDOM's peers take an approac View-Controller design pattern - the controller lives server-side (though not always), the model is what's syn 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 representa
the Python side by components then, as it evolves, IDOM's layout computes VDOM-diffs and wires them to its
displayed:



This process, in addition to drastically reducing complexity, means that Python developers with just a little bi
elabortate interfaces because they have complete control over the view. Of course many users probably don
level components, but for those who do, it's easy to distribute their creations for others to use in Python pac

Custom Javascript Components

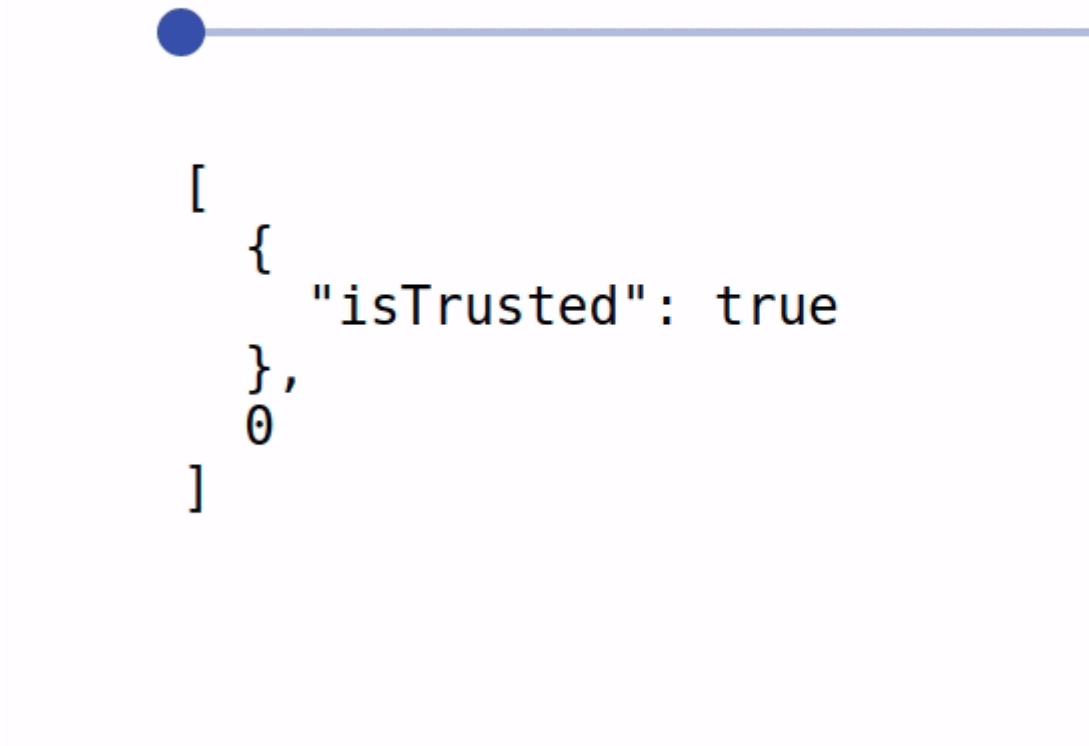
If you're thinking critically about IDOM's use of a virtual DOM, you may have thought...

> Isn't wiring a virtual representation of the view to the client, even if its diffed, expensive?

And yes, while the performance of IDOM is sufficient for most use cases, there are inevitably scenarios wher
just like its peers, IDOM makes it possible to seemlesly integrate Javascript components. They can be custom
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)
```



Conclusion

Building highly interactive web applications as a Python developer has historically been a great challenge. Ho
HTML, CSS, and Python, you can make everything from slideshows to dashboards and use it wherever you ne
an existing web application.

To learn more check out:

- the source code
- installation instructions
- how to make your first component
- interactive examples
- and much more!