ViewDOM

Paul Everitt

Mar 16, 2022

CONTENTS

ViewDOM brings modern frontend templating patterns to Python:

- tagged to have language-centered templating (like JS tagged templates)
- htm to generate virtual DOM structures from a template run (like the `htm` JS package)
- ViewDOM for components which render a VDOM to a markup string, along with other modern machinery
- Optionally, Hopscotch for a component registry with dependency injection

CHAPTER

ONE

FEATURES

- Component-driven development.
- Intermediate VDOM.
- Pass in data either via props (simple) or DI (rich).
- Emphasis on modern Python dev practices: explicit, type hinting, static analysis, testing, docs, linting, editors.

CHAPTER

TWO

REQUIREMENTS

- Python 3.9+.
- viewdom
- tagged
- htm.py
- Markupsafe

CHAPTER

THREE

INSTALLATION

You can install ViewDOM via pip from PyPI:

```
$ pip install viewdom
```

CHAPTER

FOUR

QUICK EXAMPLES

CHAPTER

FIVE

CONTRIBUTING

Contributions are very welcome. To learn more, see the *contributor's guide*.

CHAPTER

SIX

LICENSE

Distributed under the terms of the MIT license, ViewDOM is free and open source software.

CHAPTER

SEVEN

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

EIGHT

CREDITS

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

8.1 Examples

Let's take a look at some common templating patterns, implemented in `viewdom`.

8.1.1 Static String

Let's look at some non-dynamic uses of templating to learn the basics.

Render a String Literal

Let's start with the simplest form of templating: just a string, no tags, no attributes: For the purposes of illustration, we do the VDOM in one step and the rendering in a second.

```
from viewdom import html
from viewdom import render


def main() -> str:
    """Main entry point."""
    vdom = html("Hello World")
    result = render(vdom)
    return result
```

We start by importing the `html` function from `viewdom`. This is, essentially, `htm.py` in action. It takes a "tagged" template string and returns a VDOM. The `render` function, imported from `vdom`, does the rendering.

Simple Render

Same thing, but in a : nothing dynamic, just "template" a string of HTML, but done in one step:

```
def main() -> str:
    """Main entry point."""
    result = render(html("<div>Hello World</div>"))
    return result
```

We get back a VDOM – an optimized dataclass – with:

- The name of the "tag" ( )
- The properties passed to that tag (in this case, an empty dict)
- The children of this tag (in this case, a text node of Hello World)

The second item in the VDOM tuple – the props dictionary – now has a key of 'class' with the assigned class name value.

Show the VDOM Itself

Let's take a look at that VDOM structure. This time, we'll return the VDOM rather than rendering it to a string:

```
def main() -> str:
    """Main entry point."""
    result = html('<div class="container">Hello World</div>')
    return result
```

In our test we see that we got back a VDOM – an optimized dataclass:

```
    assert main() == VDOMNode(
        tag="div", props={"class": "container"}, children=["Hello World"]
    )
```

What does it look like?

- The name of the "tag" ( )
- The properties passed to that tag (in this case, an empty dict)
- The children of this tag (in this case, a text node of Hello World)

The second item in the VDOM tuple – the props dictionary – now has a key of 'class' with the assigned class name value.

Expressions as Attribute Values

We can go one step further with this and use a little bit of "expressions". Let's pass in a Python symbol as part of the template, inside curly braces:

```
def main() -> str:
    """Main entry point."""
    vdom = html('<div class="container{1}">Hello World</div>')
    result = render(vdom)
    return result
```

Everything is the same, except the value of the `class` prop now has a Python `int` included in the string. If it looks like Python `f-strings`, well, that's the point. We did an expression *inside* that prop value, using a Python expression that evaluated to just the number 1.

Shorthand Syntax

As a shorthand, when the entire attribute value is an expression, you can just use curly braces instead of putting in double quotes:

```
def main() -> str:
    """Main entry point."""
    vdom = html('<div class={"Container1".lower()}>Hello World</div>')
    result = render(vdom)
    return result
```

The VDOM's third item contains the "children".

Child Nodes in VDOM

Let's look at what more nesting would look like:

```
def main() -> VDOMNode:
    """Main entry point."""
    vdom = html("<div>Hello <span>World<em>!</em></span></div>")
    return vdom
```

Over in the test, we see what this looks like:

```
    assert main() == VDOMNode(
        tag="div",
        props={},
        children=[
            "Hello ",
            VDOMNode(
                tag="span",
                props={},
                children=["World", VDOMNode(tag="em", props={}, children=["!"])],
            ),
        ],
    )
```

It's a nested Python datastructure – pretty simple to look at.

Expressing the Doctype

One last point: the HTML doctype is a tricky one to get into the template itself as its syntax is brackety like tags. Instead, define it as a variable using `markupsafe.Markup` and insert the variable into the template:

```
def main() -> str:
    """Main entry point."""
    doctype = Markup("<!DOCTYPE html>\n")
    vdom = html("{doctype}<div>Hello World</div>")

    result = render(vdom)
    return result
```

Reducing Boolean Attribute Values

The renderer also knows to collapse truthy-y values into simplified HTML attributes. Thus, instead of `editable="1"` you just get the attribute *name* without a *value*:

```
def main() -> str:
    """Main entry point."""
    vdom = html("<div editable={True}>Hello World</div>")
    result = render(vdom)
    return result
```

8.1.2 Variables

Inserting a variable into a template mimics what you would expect from a Python f-string.

Insert Value Into Template

In this case, the template is in a function, and `name` comes from that scope:

```
def main() -> str:
    """Main entry point."""
    name = "viewdom"
    result = render(html("<div>Hello {name}</div>"))
    return result
```

Value From Import

In this third case, `name` symbol is imported from another module:

```
def Hello():
    """A simple hello component."""
    return render(html("<div>Hello {name}</div>"))


def main() -> str:
    """Main entry point."""
```

(continues on next page)

(continued from previous page)

```
    result = Hello()
    return result
```

Passed-In Prop

Of course, the function could get the symbol as an argument. This style is known as "props":

```
def Hello(name):
    """A simple hello component."""
    return render(html("<div>Hello {name}</div>"))


def main() -> str:
    """Main entry point."""
    result = Hello(name="viewdom")
    return result
```

Default Value

The function (i.e. the "component") could make passing the argument optional by providing a default:

```
def Hello(name="viewdom"):
    """A simple hello component."""
    return render(html("<div>Hello {name}</div>"))


def main() -> str:
    """Main entry point."""
    result = Hello()
    return result
```

8.1.3 Expressions

In Python f-strings, the curly brackets can take not just variable names, but also Python "expressions".

Same is true in `viewdom`.

Python Operation

Let's use an expression which adds two numbers together:

```
def main() -> str:
    """Main entry point."""
    name = "viewdom"
    result = render(html("<div>Hello {name.upper()}</div>"))
    return result
```

Simple Arithmetic

Let's use an expression which adds two numbers together:

```
def main() -> str:
    """Main entry point."""
    name = "viewdom"
    result = render(html("<div>Hello {1 + 3}</div>"))
    return result
```

Python Operation

The expression can do a bit more. For example, call a method on a string to uppercase it:

```
def main() -> str:
    """Main entry point."""
    name = "viewdom"
    result = render(html("<div>Hello {name.upper()}</div>"))
    return result
```

Call a Function

But it's Python and f-strings-ish, so you can do even more. For example, call an in-scope function with an argument, which does some work, and insert the result:

```
def make_bigly(name: str) -> str:
    """A function returning a string, rather than a component."""
    return f"BIGLY: {name.upper()}"


def main() -> str:
    """Main entry point."""
    name = "viewdom"
    result = render(html("<div>Hello {make_bigly(name)}</div>"))
    return result
```

8.1.4 Conditionals

It's a common pattern in templating: return one chunk of HTML most of the time, but under certain conditions, return a different chunk.

Thus, conditionals are a common part of templating. They're also a common part of Python f-strings, because... well, Python has conditionals. Here's a simple example using a Python "ternary":

```
def main() -> str:
    """Main entry point."""
    message = "Say Howdy"
    not_message = "So Sad"
    show_message = True
    result = render(
        html(
```

(continues on next page)

(continued from previous page)

```
            """
        <h1>Show?</h1>
        {message if show_message else not_message}
    """
        )
    )
    return result
```

8.1.5 Looping

It's common in templating to format a list of items, for example, a <ul> list. Many Python template languages invent a Python-like grammar to do `for` loops and the like.

Simple Looping

You know what's more Python-like? Python. f-strings can do looping in a Python expression using list comprehensions and so can `viewdom`:

```
def main() -> str:
    """Main entry point."""
    message = "Hello"
    names = ["World", "Universe"]
    result = render(
        html(
            """
            <ul title="{message}">
                {[
                    html('<li>{name}</li>')
                    for name in names
                ]}
            </ul>
            """
        )
    )
    return result
```

Rendered Looping

You could also move the generation of the items out of the "parent" template, then use that VDOM result in the next template:

```
def main() -> str:
    """Main entry point."""
    message = "Hello"
    names = ["World", "Universe"]
    items = [html("<li>{label}</li>") for label in names]
    result = render(
        html(
            """
```

(continues on next page)

(continued from previous page)

```
            <ul title="{message}">
              {items}
            </ul>
            """
        )
    )
    return result
```

8.1.6 Components

You often have a snippet of templating that you'd like to re-use. Many existing templating systems have "macros" for this: units of templating that can be re-used and called from other templates.

The whole mechanism, though, is quite magical:

- Where do the macros come from? Multiple layers of context magic and specially-named directories provide the answer.
- What macros are avaiable at the cursor position I'm at in a template? It's hard for an editor or IDE to predict and provide autocomplete.
- What are the macros arguments and what is this template's special syntax for providing them? And can my editor help on autocomplete or telling me when I got it wrong (or the macro changed its signature)?
- How does my current scope interact with the macro's scope, and where does it get other parts of its scope from?

`viewdom`, courtesy of `htm.py`, makes this more Pythonic through the use of "components". Instead of some sorta-callable, a component is a normal Python callable – e.g. a function – with normal Python arguments and return values.

Simple Heading

Here is a component callable – a `Heading` function – which returns a VDOM:

```
def Heading():
    """The default heading."""
    return html("<h1>My Title</h1>")


def main() -> str:
    """Main entry point."""
    vdom = html("<{Heading} />")
    result = render(vdom)
    return result
```

Component in VDOM

The VDOM now has something special in it: a callable as the "tag", rather than a string such as .

```
def Heading():
    """The default heading."""
    return html("<h1>My Title</h1>")


def main() -> VDOMNode:
    """Main entry point."""
    vdom = html("<{Heading} />")
    return vdom
```

What does that vdom node look like? This assertion shows that it is a VDOMNode:

```
    assert main() == VDOMNode(tag=Heading, props={}, children=[])
```

Simple Props

As expected, components can have props, passed in as what looks like HTML attributes. Here we pass a `title` as an argument to `Heading`, using a simple HTML attribute string value:

```
def Heading(title):
    """Default heading."""
    return html("<h1>{title}</h1>")


def main() -> str:
    """Main entry point."""
    result = render(html('<{Heading} title="My Title" />'))
    return result
```

Children As Props

If your template has children inside that tag, your component can ask for them as an argument, then place them as a variable:

```
def Heading(children, title):
    """The default heading."""
    return html("<h1>{title}</h1><div>{children}</div>")


def main() -> str:
    """Main entry point."""
    result = render(html('<{Heading} title="My Title">Child<//>'))
    return result
```

`children` is a keyword argument that is available to components. Note how the component closes with <//> when it contains nested children, as opposed to the self-closing form in the first example.

Expressions as Prop Values

The “prop” can also be a Python symbol, using curly braces as the attribute value:

```
def Heading(title):
    """The default heading."""
    return html("<h1>{title}</h1>")


def main() -> str:
    """Main entry point."""
    result = render(html('<{Heading} title={"My Title"} />'))
    return result
```

Prop Values from Scope Variables

That prop value can also be an in-scope variable:

```
def Heading(title):
    """The default heading."""
    return html("<h1>{title}</h1>")


this_title = "My Title"


def main() -> str:
    """Main entry point."""
    result = render(html("<{Heading} title={this_title} />"))
    return result
```

Optional Props

Since this is typical function-argument stuff, you can have optional props through argument defaults:

```
def Heading(title="My Title"):
    """The default heading."""
    return html("<h1>{title}</h1>")


def main() -> str:
    """Main entry point."""
    result = render(html("<{Heading} />"))
    return result
```

Spread Props

Sometimes you just want to pass everything in a dict as props. In JS, this is known as the "spread operator" and is supported:

```
def Heading(title, this_id):
    """The default heading."""
    return html("<div title={title} id={this_id}>Hello</div>")


def main() -> str:
    """Main entry point."""
    props = dict(title="My Title", this_id="d1")
    result = render(html("<{Heading} ...{props}>Child</>"))
    return result
```

Pass Component as Prop

Here's a useful pattern: you can pass a component as a "prop" to another component. This lets the caller – in this case, the `result` line – do the driving:

```
def DefaultHeading():
    """The default heading."""
    return html("<h1>Default Heading</h1>")


def Body(heading):
    """The body which renders the heading."""
    return html("<body><{heading} /></body>")


def main() -> str:
    """Main entry point."""
    result = render(html("<{Body} heading={DefaultHeading} />"))
    return result
```

Default Component for Prop

As a variation, let the caller do the driving but make the prop default to a default component if none was provided:

```
def DefaultHeading():  # pragma: nocover
    """The default heading."""
    return html("<h1>Default Heading</h1>")


def OtherHeading():
    """Another heading used in another condition."""
    return html("<h1>Other Heading</h1>")


def Body(heading=DefaultHeading):
```

(continues on next page)

(continued from previous page)

```
    """Render the body with a heading based on which is passed in."""
    return html("<body><{heading} /></body>")


def main() -> str:
    """Main entry point."""
    result = render(html("<{Body} heading={OtherHeading}/>"))
    return result
```

Conditional Default

One final variation for passing a component as a prop… move the "default or passed-in" decision into the template itself:

```
def DefaultHeading():
    """The default heading."""
    return html("<h1>Default Heading</h1>")


def OtherHeading():
    """Another heading used in another condition."""
    return html("<h1>Other Heading</h1>")


def Body(heading=None):
    """Render the body with a heading based on which is passed in."""
    return html("<body>{heading if heading else DefaultHeading}</body>")


def main() -> str:
    """Main entry point."""
    result = render(html("<{Body} heading={OtherHeading}/>"))
    return result
```

Children as Prop

You can combine different props and arguments. In this case, `title` is a prop. `children` is another argument, but is provided automatically by `render`.

```
def Heading(children, title):
    """The default heading."""
    return html("<h1>{title}</h1><div>{children}</div>")


def main() -> str:
    """Main entry point."""
    result = render(html('<{Heading} title="My Title">Child<//>'))
    return result
```

Generators as Components

You can also have components that act as generators. For example, imagine you have a todo list. There might be a lot of todos, so you want to generate them in a memory-efficient way:

```
def Todos():
    """A sequence of li items."""
    for todo in ["First", "Second"]:  # noqa B007
        yield html("<li>{todo}</li>")


def main() -> str:
    """Main entry point."""
    result = render(html("<ul><{Todos}/></ul>"))
    return result
```

Subcomponents

Subcomponents are also feasible. They make up part of both the VDOM and the rendering:

```
def Todo(label):
    """An individual to do component."""
    return html("<li>{label}</li>")


def TodoList(todos):
    """A to do list component."""
    return html("<ul>{[Todo(label) for label in todos]}</ul>")


def main() -> str:
    """Main entry point."""
    todos = ["first"]
    return render(
        html(
            """
      <h1>{title}</h1>
      <{TodoList} todos={todos} />
    """
        )
    )
```

Architectural Note

Components and subcomponents are a useful feature to users of some UI layer, as well as creators of that layer. They are also, though, an interesting architectural plug point.

As `render` walks through a VDOM, a usage of a component pops up with the props and children. But it isn't yet the *rendered* component. The callable…hasn't been called.

It's the job of `render` to do so. If you look at the code for `render` and the utilities it uses, it's not a lot of code. It's reasonable to write your own, which is what some of the integrations have done.

This becomes an interesting place to experiment with your own component policies. Want a cache layer? Want to log each call? Maybe type validation? Want to (like the `wired` integration) write a custom DI system that gets argument values from special locations (e.g. database queries)?

Lots of possibilities, especially since the surface area is small enough and easy enough to play around.

8.1.7 Registry and Injection

The ViewDOM component system can be used as "better templating": smaller, testable, Pythonic, re-usable units. But it really shines when combined with Hopscotch for replaceable, injectable components.

Simple Registered Heading

We'll start with the `Heading` component from the first components example. In this case, our component is a dataclass, rather than a function:

```
@injectable()
@dataclass
class Heading:
    """The default heading."""

    def __call__(self) -> VDOM:
        """Render the component."""
        return html("<h1>My Title</h1>")
```

This time our main function simulates making a registry and processing a request:

```
from .app import Heading  # noqa: F401
from viewdom import html
from viewdom import render


def main() -> str:
    """Main entry point."""
    # At startup
    registry = Registry()
    registry.scan()

    # Per request
    vdom = html("<{Heading} />")
    result = render(vdom, registry=registry)
    return result
```

Replacement

In the previous example, the component shipped with a pluggable app's package. But a local site might want to register a replacement for that component. Here is a `site.py` which does just this – replace `app.Heading` with a different implementation:

```
@injectable(kind=Heading)
@dataclass
class SiteHeading:
    """A heading customized to the site."""

    def __call__(self) -> VDOM:
        """Render the component."""
        return html("<h1>Local Site Title</h1>")
```

This time the decorator said `kind=Heading`.

Nothing changed in `__init__.py`, and yet, a different class was selected and `Local Site Title` is the result. How did that work, without monkey-patching?

It's because ViewDOM can *tranparently* look up components using a registry. In that model, `app.Heading` isn't a dataclass. It's a "kind"…sort of like an interface, or a base type, or a protocol. The registry has two implementations, and the second one was added last, so it was used.

That's *replacement*. But what if you want *both* implementations, but used in different contexts?

Variants

First, let's imagine our app had a concept of a `Customer`:

```
@dataclass
class Customer:
    """The person to greet, stored as the registry context."""

    first_name: str
```

We could then tell our "replacement" in our local site to *only* replace in certain cases. Namely, when the "context" is a `Customer`:

```
@injectable(kind=Heading, context=Customer)
@dataclass
class SiteHeading:
    """A heading customized to the site."""

    def __call__(self) -> VDOM:
        """Render the component."""
        return html("<h1>Local Site Title</h1>")
```

What is the "context"? It's a special aspect of a registry. In this case, a per-request *child registry*:

```
from .app import Customer
from .app import Heading  # noqa: F401
from viewdom import html
from viewdom import render
```

(continues on next page)

(continued from previous page)

```
def main() -> tuple[str, str]:
    """Main entry point."""
    # At startup
    registry = Registry()
    registry.scan()

    # First request, no customer, context=None
    request_registry0 = Registry(parent=registry, context=None)
    vdom = html("<{Heading} />")
    result0 = render(vdom, registry=request_registry0)

    # Second request, context=customer
    customer = Customer(first_name="Marie")
    request_registry1 = Registry(parent=registry, context=customer)
    vdom = html("<{Heading} />")
    result1 = render(vdom, registry=request_registry1)

    return result0, result1
```

As you can see in the test, each "request" renders a different output:

```
    assert main() == ("<h1>My Title</h1>", "<h1>Local Site Title</h1>")
```

From a component developer's perspective, this is all transparent. But what if I want my component to get access to that `Customer`?

Simple Injection

We could arrange for the caller to pass in the context into the component. But if the component is way down the component tree, it will have to be passed by all the parents. What if the component could just ask the registry – that is, the registry's *injector* – to provide the `context`

```
@injectable(kind=Heading, context=Customer)
@dataclass
class SiteHeading:
    """A heading customized to the site."""

    customer: Customer = context()

    def __call__(self) -> VDOM:
        """Render the component."""
        first_name = self.customer.first_name
        return html("<h1>Hello {first_name}</h1>")
```

Our component can then get the `first_name` off the context, because we said it was a `Customer` in the type hint.

What is `context()`? It is a Hopscotch "operator": something which acts like a `dataclasses.field` but during injection, does extra work. You can write your own operators, but the built-in ones have some extra features.

Only Inject What You Need

Our component says it needs the entire `Customer`. That's a broad surface area. Let's change it, to ask the injector to get just the piece of information we need – the `first_name` – as part of injection:

```
@injectable(kind=Heading, context=Customer)
@dataclass
class SiteHeading:
    """A heading customized to the site."""

    customer_name: str = context(attr="first_name")

    def __call__(self) -> VDOM:
        """Render the component."""
        return html("<h1>Hello {self.customer_name}</h1>")
```

Note: Broke The Contract You'll note that `app.Heading` and `site.SiteHeading` have deviated in "props".

This has some powerful consequences. First, you can customize a component by passing `first_name` in manually as a prop, which will be used instead of injection. For example, `html('<{Heading} first_name="Bob" />')`.

In a larger sense, your components could become observable. During registration, Hopscotch could record that you need that value. Then during injection, it could persist the `Customer` and this `SiteHeading` instance, which a relationship. If the particular customer's `first_name` changed, you could rebuild just that component instance.

8.1.8 Context

Warning: Currently Disabled The code for context is still there but unused. I plan to re-enable when switching away from threadlocal to maybe asyncio ContextVar.

When you really embrace components and subcomponents, you end up with deeply nested component trees. It can be frustrating, not to mention brittle, to pass data all the way from the top to the bottom. Intermediate components don't use the data. Why should they have to depend on them?

Pass-Through

This is the harder, manual way.

In this example the `site` object is passed through the `App` component, then `Nav`, to get it to `NavHeading`. Neither `App` nor `Nav` need anything from `site`. They are just transits.

```
def NavHeading(title):
    """A navigation heading component."""
    return html("<h1>{title}</h1>")


def Nav(site):
    """A navigation component."""
    return html('<nav><{NavHeading} title={site["title"]}/></nav>')
```

(continues on next page)

(continued from previous page)

```
def PageHeading(title):
    """Title block for a page."""
    return html("<h2>{title}</h2>")


def Main(page):
    """Full page layout."""
    return html('<{PageHeading} title={page["title"]}/>')


def App(site, page):
    """An app for a site."""
    return html(
        """
        <{Nav} site={site}/>
        <{Main} page={page}/>
    """
    )


def main():
    """Main entry point."""
    site = dict(title="My Site")
    page = dict(title="My Page")

    return render(
        html(
            """
        <{App} site={site} page={page} />
"""
        )
    )
```

Context

This is the easier way.

React has addressed this with a number of technologies over the years, including Context and Hooks. `viewdom` has a similar "context" construct, where you can wrap a part of the component tree, shove values into the rendering, and pluck them out later on. We can use this to make `site` available anywhere in the tree:

```
def NavHeading():
    """A navigation heading component."""
    site = use_context("site")
    title = site["title"]
    return html("<h1>{title}</h1>")


def Nav():
    """A navigation component."""
```

(continues on next page)

(continued from previous page)

```
    return html("<nav><{NavHeading}/></nav>")


def PageHeading(title):
    """Title block for a page."""
    return html("<h2>{title}</h2>")


def Main(page):
    """Full page layout."""
    return html('<{PageHeading} title={page["title"]}/>')


def App(page):
    """An app for a site."""
    return html(
        """
        <{Nav}/>
        <{Main} page={page}/>
    """
    )


def main():
    """Main entry point."""
    site = dict(title="My Site")
    page = dict(title="My Page")

    return render(
        html(
            """
        <{Context} site={site}>
            <{App} page={page} />
        <//>
"""
        )
    )
```

As you can imagine, a React Hooks approach could also be implemented.

8.2 Reference

`ViewDOM` only has a few public symbols to be used by other packages. Here's the API.

ViewDOM.

`viewdom.`**`Context`**(*children=None*, ***kwargs*)
Like the React Conext API.

`class viewdom.`**`VDOMNode`**(*tag*, *props*, *children*)
Implementation of a node with three slots.

Parameters

- **tag** (*str*) –
- **props** (*Mapping*) –
- **children** (*List[Union[str, viewdom.VDOMNode]]*) –

Return type None

viewdom.**encode_prop**(*k*, *v*)
If possible, reduce an attribute to just the name.

viewdom.**flatten**(*value*)
Reduce a sequence.

viewdom.**htm**(*func=None*, ***, *cache_maxsize=128*)
The callable function to act as decorator.

Return type *Callable*[[str], *Union*[*Sequence*[*viewdom.VDOMNode*], *viewdom.VDOMNode*]]

viewdom.**relaxed_call**(*callable_*, *registry=None*, *children=None*, *props=None*)
Get the correct implementation and call it to produce a vdom.

Parameters

- **registry** (*Optional[hopscotch.registry.Registry]*) –
- **props** (*Optional[dict[str, Any]]*) –

Return type *Union*[*Sequence*[*viewdom.VDOMNode*], *viewdom.VDOMNode*]

viewdom.**render**(*value*, *registry=None*, ***kwargs*)
Render a VDOM to a string.

Parameters

- **value** (*Union[Sequence[viewdom.VDOMNode], viewdom.VDOMNode]*) –
- **registry** (*Optional[hopscotch.registry.Registry]*) –

Return type str

viewdom.**render_gen**(*value*, *registry=None*, *children=None*)
Render as a generator.

Parameters **registry** (*Optional[hopscotch.registry.Registry]*) –

viewdom.**use_context**(*key*, *default=None*)
Similar to the React use context API.

8.3 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

8.3.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

8.3.2 How to request a feature

Request features on the Issue Tracker.

8.3.3 How to set up your development environment

You need Python 3.6+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run viewdom
```

8.3.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

8.3.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before commiting your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

8.4 Contributor Covenant Code of Conduct

8.4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

8.4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

8.4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

8.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at pauleveritt@me.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

8.4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

8.5 MIT License

PYTHON MODULE INDEX

V

INDEX