Transcrypt Hyperapp Hydrate Documentation

Paul Everitt <pauleveritt@me.com>

Dec 01, 2018

Contents:

Transcrypt is fantastically interesting: write advanced JavaScript from advanced Python. Hyperapp is fantastically interesting: modern browser applications in a tiny amount of code.

But the JS toolchain? Ugh. My GatsbyJS `node_modules` is 500 Mb. Isn't there a better way? Python, and Transcrypt, to the rescue. Write your JS in Python, where it can execute both in a Python web application and later in the browser. Same code.

Let's do an example, in tutorial format, of explaining these two, individually and in combination. Let's then explore some further topics:

- Server-Side-Rendering(SSR), where the initial page's HTML is generated and delivered, then replaced with client-side rendering if JS is supported (aka "hydration")

Other topics in the future:

- Transcrypt's static type analysis
- Client-side routing, where links between pages are intercepted and handled with a JS router
- Doing server-side "templating" and client-side "templating" with the same code
- Link pre-fetching
- Offline-first

CHAPTER 1

Setup

We need to do the normal Python dance: make a virtual environment and install our dependencies. We also need to download the two files from Hyperapp.

We want to show going an extra step and get the packages for a development environment that matches some of the webpack-devserver auto-build/reload.

1.1 Virtual Environment

This one is obvious:

```
$ python3 -m venv .env
$ .env/bin/pip install --upgrade pip setuptools
```

1.2 Dependencies

We need Transcrypt, obviously. We're also going to use Bottle connected to the Python `livereload` package to watch for changes and reload the browser:

Note: We're picking a specific livereload version to avoid a bug.

Let's also get our two JS files, Hyperapp itself and a Hyperapp HTML helper package (as an alternative to JSX):

```
$ wget https://unpkg.com/hyperapp@1.2.9/dist/hyperapp.js
$ wget https://unpkg.com/@hyperapp/html@1.1.1/dist/hyperappHtml.js
```

CHAPTER 2

Hello LiveReload

Let's get some bits on the screen. In this step we're going to make a small Bottle application which returns an HTML template and the two script tags to load our Hyperapp files.

2.1 LiveReload

Make a file `run_server.py` with the following:

```
import bottle
from bottle import route, default_app, static_file
from bottle import template
from livereload import Server

bottle.debug(True)


@route('/')
def index():
    return template('index', state=dict(count=0))


@route('/<filepath:path>')
def server_static(filepath):
    return static_file(filepath, root='.')


if __name__ == '__main__':
    app = default_app()
    server = Server(app)
    server.watch('index.css')
    server.watch('counter.js')
    server.watch('index.tpl', ignore=False)
    server.serve()
```

This is a simple Bottle app with two routes:

- / uses a Bottle template at index.tpl to dynamically render a response
- Everything else is served as static files from the root directory.

2.2 Template

Now make index.tpl:

```
<html>
<head>
    <title>Transcrypt Hyperapp</title>
    <link rel="icon" type="image/x-icon"
          href="https://www.python.org/static/favicon.ico">
    <link rel="stylesheet" href="index.css"/>
    <script src="hyperapp.js"></script>
    <script src="hyperappHtml.js"></script>
</head>
<body>
<h1>Transcrypt + Hyperapp Demo</h1>
<p>This sample application has a counter. On first load, the counter
    renders, server-side, the HTML. After that, client-side events
    re-render the counter.</p>
<div id="counter">
    <div>
        <h3>Counter Demo</h3>
        <p>Current Count: {{ state['count'] }}</p>
        <button>-</button>
        <button>+</button>
    </div>
</div>
<script src="counter.js"></script>
</body>
</html>
```

This template is passed in the "state" of the application. It inserts the count value, as well as includes script and link tags for the static assets.

As this shows, when the browser makes a request to the URL, you get "server-side rendering": the current value of count is inserted, as well as the two buttons which will later be made dynamic.

2.3 JS

As a placeholder for the next step, create counter.js:

```
console.log('Loaded')
```

2.4 CSS

Not much in index.css:

```
body {
    font-family: "Helvetica Neue", sans-serif;
    margin: 2em 20em 2em 2em;
}

#counter {
    margin: 2em;
    padding: 1em;
    border: solid gray 1px;
}
```

2.5 Running

With that in place, let's run the server:

```
$ python ./run_server.py
```

Open `http://127.0.0.1:5500/` in a browser, preferably with the dev console open. You get the first render.

Now try editing the `run_server.py` and changing the initial count. The application restarts and you see the new value.

Now edit `index.tpl` and `index.css`. The changes update much faster, as the Python process doesn't restart. Instead, the server tells the browser to reload.

CHAPTER 3

Hello Hyperapp

Lots of choices for JavaScript frontend frameworks, with React leading the way. But most of those choices end up meaning a *ginormous* toolchain of ever-changing, ever-breaking tools to transpile the universe.

Hyperapp is the impossibly-small, yet feature-rich JS framework. Small as in, under 1 KB. Small enough to not actually need minification and bundling…and in browsers which support ES Modules you still get import semantics without tools like webpack.

Let's wire up our counter.

```
const {app} = window.hyperapp;
const html = window.hyperappHtml;
const {h3, p, div, button} = html;

const state = {
  count: 0
}

const actions = {
  down: () => state => ({ count: state.count - 1 }),
  up: () => state => ({ count: state.count + 1 })
}

const view = (state, actions) =>
  div([
    h3('Counter Demo'),
    p('Current Count: ' + state.count),
    button({ onclick: actions.down }, "-"),
    button({ onclick: actions.up }, "+")
  ])

app(state, actions, view, document.getElementById("counter"))
```

Note: Because we aren't bundling, we get `hyperapp` and `hyperappHtml` from the `window` object.

What's in this simple counter?

- A section at the top that is the moral-equivalent of ES6 imports
- The initial state of the counter
- Two *actions*, which are arrow functions which Hyperapp calls, providing the state
- The *view*, which is usually JSX, but we are doing using the Hyperapp `html` helper package. We don't want JSX because we don't want a JavaScript toolchain to have to process it into JS. We have another reason, which we'll reveal in the next section.
- The actual `app` which wires the pieces together and taks over the `counter` node in the markup

As a recap:

- On the first load, Bottle runs the template and generates – server-side in Python – the initial HTML
- JavaScript then kicks in and "hydrates" the existing DOM element, rendering the view into it…keeping the existing nodes which are the same, replacing changed ones, and adding new ones. This is the mythical "VDOM" that you may have heard of.

To emphasize: we are doing SSR, but with Python, not a NodeJS framework.

CHAPTER 4

Hello Transcrypt

This step is gonna get freaky.

Perhaps you noticed that we wrote the counter in JS. Perhaps you noticed that there will therefore be a duplication in implementation logic: Python for the first render and JavaScript for once it is hydrated.

Let's write the counter once, in Python, and have Transcrypt generate the JS for Hyperapp.

4.1 Counter, In Python

Make a file called `counter.py`:

```
from hyperappHtml import button, div, h3, p

state = dict(count=10)

actions = dict(
    down=lambda value: lambda s: dict(count=s.count - value),
    up=lambda value: lambda s: dict(count=s.count + value),
)


def view(st, ac):
    return div({}, [
        h3({}, 'Welcome Counter'),
        p({}, 'Current Counter: ' + str(st.count)),
        button(dict(onclick=lambda: ac.down(1)), "-"),
        button(dict(onclick=lambda: ac.up(1)), "+")
    ])


try:
    window.hyperapp.app(
        state,
```

(continues on next page)

(continued from previous page)

```
        actions,
        view,
        document.getElementById("counter")
    )
except NameError:
    pass
```

This is a Python file that will be run through Transcrypt to generate the moral equivalent of `counter.js` which we just saw. If you use a smart editor, you'll see red squigglies complaining that `window` and `document` are undefined.

And that's true. Those are symbols which don't exist in Python. They spring into existence when run in JavaScript.

Note: Need to find a way to teach the editor that those symbols can exist and give them a structure.

In the first line, you see a Python import of `hyperappHtml`. What's that? The `hyperapp/html` "package" is really a single, simple file. This tutorial just converted it to Python, to allow us to work towards a single "template" which works both on initial render (in Python) and browser-side (in JS).

And here's that file.

4.2 HTML Helper

Create `hyperappHtml.py` with the following:

```
try:
    h = window.hyperapp.h
except NameError:
    def h(name, attributes, children):
        # TODO Implement h in Python
        pass


def vnode(name):
    def f(attributes, children):
        return h(name, attributes, children)

    return f


def h3(attributes, children):
    return vnode("h1")(attributes, children)


def h4(attributes, children):
    return vnode("h1")(attributes, children)


def button(attributes, children):
    return vnode("button")(attributes, children)


def div(attributes, children):
    return vnode("div")(attributes, children)
```

(continues on next page)

(continued from previous page)

```
def p(attributes, children):
    return vnode("div")(attributes, children)
```

As you can see, this is just a proof-of-concept:

- It doesn't include all the tags
- It can't be used 100% in Python, because it still needs the `h` function from Hyperapp (only 60 or so lines, but dense in concepts)

Still, it allows us to use Python tooling (e.g. type hinting) when writing our views, rather than rely on some alien symbol representing a JavaScript function.

4.3 Loading the Generated JS

Transcrypt is going to read our Python and generate JS. We need to load that generated JS. Transcrypt generated ESM-friendly JS, so we can do an import. Change `index.tpl` to (a) remove loading `hyperHtml.js`, (b) remove loading `counter.js`, and (c) add the new `<script`:

```
<html>
<head>
    <title>Transcrypt Hyperapp</title>
    <link rel="icon" type="image/x-icon"
          href="https://www.python.org/static/favicon.ico">
    <link rel="stylesheet" href="index.css"/>
    <script src="hyperapp.js"></script>
    <script src="hyperappHtml.js"></script>
</head>
<body>
<h1>Transcrypt + Hyperapp Demo</h1>
<p>This sample application has a counter. On first load, the counter
    renders, server-side, the HTML. After that, client-side events
    re-render the counter.</p>
<div id="counter">
    <div>
        <h3>Counter Demo</h3>
        <p>Current Count: {{ state['count'] }}</p>
        <button>-</button>
        <button>+</button>
    </div>
</div>
<script src="counter.js"></script>
</body>
</html>
```

4.4 Run Transcrypt

Now it's time to get Transcrypt to do its magic. Let's have `run_server.py` handle it, so we can run it any time `counter.py` changes. Even better, the browser will reload with our changes.

```
import bottle
from bottle import route, default_app, static_file
from bottle import template
from livereload import Server, shell

from counter import state

bottle.debug(True)

# TODO Instead of forking and running Python, extract
#   stuff from transcrypt.__main__.main
transpile = 'env/bin/transcrypt -b -m -n counter.py'


@route('/')
def index():
    return template('index', state=state)


@route('/<filepath:path>')
def server_static(filepath):
    return static_file(filepath, root='.')


if __name__ == '__main__':
    app = default_app()
    server = Server(app)
    server.watch('counter.py', shell(transpile))
    server.watch('hydrate/__target__/*', delay=2)
    server.watch('index.css')
    server.watch('counter.js')
    server.watch('index.tpl', ignore=False)
    server.serve()
```

In the previous step, the initial-rendering (in Python) used a different state than the hydrated count, which was in JS. Now that the counter is in Python, we can import the initial value and hand it to the template.

4.5 Future: One Template

This is all a little freaky. It's hard to remember some times which side of the fence is governing which of the executions (initial in Python, later in JS.)

Of course, it can get a lot freakier. For example, we still have two templates:

- The `index.tpl` Bottle template
- The `hyperapp/html` (converted to Python) in the Hyperapp view

Wouldn't it be great if there was just one template? That way the initial render could use the same template that was used later.

A full-port of `hyperapp/html` to Python is half the equation. The other is harder: porting the "h" function is also needed. It's short but complicated. Some of the complexity could go, as a diff isn't needed: this is the first render.

CHAPTER 5

Next Steps

Lots more to do to match the capability of SSR with React etc. Foremost, client-side routing. With this, clicking on links doesn't go back to the server.

Also, as systems like Gatsby show, you can do progressive-web apps that are offline-first.