A Declarative Front End Framework from Scratch

Peter Crona
petercrona89@gmail.com

ABSTRACT

This paper is a report from a personal learning project. In order to practice front end coding skills, a simple front end application was built using TypeScript. The application was developed using a highly declarative style, and without the use of any libraries or frameworks. The application structure is heavily inspired by functional programming. The initial goal was to build a simple to-do application in at most one MTU (1500B), however, this evolved to trying different ideas with regards to front end application structure. Ultimately, I ended up building my own little front end framework.

Code: https://github.com/petercrona/frontend-framework-from-scratch

1 APPLICATION STRUCTURE

For a functional programmer, at least someone into Haskell, it all starts with types. While a lot of code may be needed to realise an idea, the types can often express most of the story very concisely. For brevity, pseudo notation inspired by Haskell is used for expressing types in this paper, types and code will also routinely be simplified to not distract from the main points. The main two types of this project are:

```
Component :: Context ->
             Promise Application


Application :: {Node, Controllers}
```

Experienced functional programmers likely can already tell that I'm essentially using something along the lines of a "PromiseReader" monad, which is just another way of saying that my type declares that to run my code you must provide a context, and you will get a Promise back (which gives me the power of doing things such as loading data from external APIs). Another noteworthy observation is that I am not only returning a Node, but have some kind of additional state (Controllers). So perhaps "PromiseReaderState" would be another way to describe what is going on here. But, we're not in Haskell world, nor are we "formally" working with monads, so perhaps we should stop trying to name it as if we were. And, if this is confusing, don't worry, let's get to the fun and continue with some examples.

2 HELLO WORLD EXAMPLE

Given the types presented above, how would we go about to create a simple application which just prints Hello World in the browser?

Our Component type tells us that we need to produce a function which given a context produces the Promise of an application. This does not sound too hard, let us give it a shot:

```
HelloWorld = (): Promise<Application>
 => {
      node = document.createElement("p");
      node.textContent = "Hello World";
      Promise.resolve(
          {node, controllers: {}}
      );
    }
```

We're defining a function which takes no argument and returns an Application with a node containing "Hello World". To make Hello World appear in the DOM, we can run our component:

```
HelloWorld()
  .then(({node}) =>
    document.body.appendChild(node)
  )
```

Note that our HelloWorld function, or Component, isn't taking a Context. But, this is just a matter of perception, without any code modification, we can claim that it is taking a Context, just that it isn't using it:

```
HelloWorld: Component =
  () => {[SAME AS BEFORE]}
```

Note that all our examples are simplified, for instance, in reality we'd want to specify things such as what type of Node a Component returns. These specifications would simply be type arguments, for example:

```
Component<HTMLDivElement, {}>
```

However, we can mostly relieve ourselves of the burden to write these specifications by using utility functions when constructing our Component. With such utility functions, our Hello World becomes:

```
HelloWorld = pipe(
  ofElement(
    "p",
    (e) => (e.textContent = "Hello World")
  )
)({});
```

Alternatively, if we want to pass in "document.createElement" via the context:

```
HelloWorld = pipe(
  ofElement(
    "p",
    (e) => (e.textContent = "Hello World")
  ),
)({ mkElem:
      document.createElement.bind(document)
});
```

where "ofElement" creates a component which requires "mkElem" to be in the Context. The Component creates and sets the text content of our HTML element.

Note that "pipe" here is a function that given a value (the initial argument) passes it through the following functions. "pipe" is actually not needed here, but will come in handy later. I first saw "pipe" in FP-TS [6], but it is also similar to "flow" in Lodash [2]. One implementation is:

```
((value: any, ...fns: Fn<any, any>[]) =>
  fns.reduce(
    (result, fn) => fn(result),
    value
  );
)
```

In TypeScript, a function like "pipe" can be useful since it helps with type inference while writing the code. However, additional type specification is needed for this. A brief example of using "pipe":

```
pipe(5, x => x + 1)
```

In the code snippet, x is inferred to be a number, and the result of the whole expression is also inferred to be a number.

To end the Hello World example, let us consider how we could test our application. There is not much to test, but, to highlight a benefit of computing with an explicit context, let us try to test it in a Node, as opposed to JSDOM, environment. That is, we want to test it in an environment where there is no DOM. The example is using Jest [1].

```
describe("HelloWorld", () => {
  test("node has text hello world", () => {
    const mkElem = jest.fn()
      .mockImplementation((tag) => ({
        tag,
      }));
    HelloWorld({ mkElem })
      .then(({ node }) => {
        expect(node.textContent)
          .toBe("Hello World");
      });
  });
});
```

As shown, we can easily test our simple Hello World component without any DOM functionality. This example was mainly for showing the power of using an explicit Context, and not to discourage taking advantage of JSDOM.

Hello World is a good starting point, but, given how trivial it is, you may wonder, why not just use HTML or JavaScript/TypeScript without any additional abstractions. Let us now look at a slightly more complex example, namely, a to-do application.

3 TO-DO APPLICATION EXAMPLE

A to-do application is a much more complex example, therefore, the description will be kept at a high-level. If you are interested, feel free to check out the code on Github for a more complete and complex example.

A good starting point is to think about what components our to-do application needs. We surely need a component to add a new to-do item, we need a list displaying our to-do items, the to-do items themselves of course, and finally something to put it all together, perhaps we can call that TodosApp. Let us write it out:

- TodoAdd
- TodoList
- ListItem
- TodosApp

In the spirit of functional programming, let us now map these components into their implementation, starting with TodoAdd:

```
TodoAdd = () => pipe(
  ofElement("form"),
  addChild(TextInput(), "todoValue"),
  addChild(Button("add")),
  run(({ node, controllers }, util) => {
    const textInput = util.getController(
      controllers.TextInput,
      "todoValue"
    );

    Promise.resolve({
      name: "TodoAdd",
      onAdd: (fn) => {
        node.onSubmit((ev) => {
          ev.preventDefault();
          fn(textInput.getValue());
          textInput.clear();
        });
      },
    });
  })
);
```

A not previously presented function here is "addChild", it has the type:

```
addChild :: Component
            -> String
            -> Component
            -> Component
```

Given a component and id, it adds the provided component to the later supplied component as a child, and returns the resulting component.

In the background, "addChild" also takes care of other useful things, for instance, making sure that the parent component can access the controllers of the child. You see an example of this in the "run" function, where we use the controller of the "textInput" to get the value. We also "export" a controller called "TodoAdd" which has the function "onAdd", which will come in handy when we connect everything. Let us now continue with the next component, namely "TodoList":

```
TodoList = () => pipe(
  ofElement("ul"),
  tellController<GetController<ListItem>>(),
  run((app, _, context) =>
    Promise.resolve({
      name: "TodoList",
      addTodo: (value: string) => {
        setChildren(app, context, "APPEND")
                    ([ListItem(value)]);
      },
      getTodos: () => {
        return app.node
          .querySelectorAll("li")
          .map(node =>
            app.controllers.ListItem
                           .get(node)
          )
          .map(controller => ({
            value: controller.getValue(),
            checked: controller.isChecked(),
          }));
      },
    });
  )
);
```

A new function/modifier here is "tellController". The implementation is identity, but, it does update the type to make our component aware of that ListItem's controller will be accessible via our component. Since ListItem is dynamically added, we do not get this for free as when using "addChild". "ListItem" is added using "setChildren" in our "addTodo" function. We export "addTodo" via our controller (so that anyone who does "addChild(TodoList())" can access it.

Also, note that we have an example of loading all children (li elements) and getting controllers. The "util.getController" function that we used elsewhere is not the only way to get hold of a controller. In fact, "app.controllers" is a "WeakMap<Node, Controller>", so, all you need is the right node.

You might have noticed that we used a new component "ListItem", but we haven't defined it, let us do that now:

```
ListItem = (value: string) =>
  pipe(
    ofElement("li"),
    addChild(Checkbox(), "checkbox"),
    addChild(
      ofElement(
        "span",
        (e) => (e.textContent = value)
      )
    ),
    addChild(Button("Remove"), "remove"),
    run((app, util) => {
      buttonRemove = util.getController(
        app.controllers.Button,
        "remove"
      );
      checkbox = util.getController(
        app.controllers.Checkbox,
        "checkbox"
      );

      buttonRemove.onClick(() => {
        app.node.remove();
      });

      return Promise.resolve({
        name: "ListItem",
        isChecked: checkbox.isChecked,
        getValue: () => value,
      });
    })
  );
```

The code represents a list item containing a checkbox, some text, and a remove button. We are not doing anything special here, just the same as we have done before. We're creating an element, adding some children and running some code, using controllers. One noteworthy thing is that the value supplied when creating the "ListItem" component is returned when someone calls "getValue" on the controller. But, for the checkbox, we are reading it from the checkbox controller, which ultimately gets the state from the DOM.

We still need to tie it all together, let us now do that:

```
TodosApp = () => pipe(
  ofElement("div"),
  addChild(TodoAdd, "todoAdd"),
  addChild(TodoList, "todoList"),
  addChild(Button("save"), "save"),
  run((app, util) => {
    todoAdd = util.getController(
      app.controllers.TodoAdd,
      "todoAdd"
    );
    todoList = util.getController(
      app.controllers.TodoList,
      "todoList"
    );
    save = util.getController(
      app.controllers.Button,
      "save"
    );

    todoAdd.onAdd(todoList.addTodo);

    save.onClick(() => {
      console.log(todoList.getTodos());
    });
  })
);
```

Note that we're not doing much at this point, but just adding what we built before, and glueing it together through the well-defined and typed interfaces exposed by the controllers.

Let us now briefly compare what we have built with a typical approach with React. Then we can proceed by looking at some details, such as memory management.

4 BRIEF COMPARISON WITH REACT

React is a versatile library that allows you to structure your application in many different ways. My implementation is very simplistic but reflects the type of structure I have seen many times, but, it is by no means the only way to structure a to-do application, and likely not the best way. With that said, here we go:

```
TodoAdd = ({ onAdd }) => {
  const [todoValue, setTodoValue] =
    useState("");

  const handleSubmit = (ev) => {
    ev.preventDefault();
    onAdd(todoValue);
    setTodoValue("");
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        name="value"
        type="text"
        value={todoValue}
        onChange={(ev) =>
          setTodoValue(ev.target.value)}
      />
      <button>Add</button>
    </form>
  );
}
```

First difference is that we have explicit state in the React version using the "useState" hook. This was added in order to reset the text field on submit. Perhaps this could have been avoided, but, in React components you do fairly often see state, especially if there is a form involved. The non-React to-do application avoided this by letting the browser handle the state, and just offer a way of observing it.

Another notable difference is that in React we're getting the callback ("onAdd") via props, as opposed to the component exposing "onAdd". In React you may allow people to skip supplying "onAdd" (make it nullable), whereas in my application you either use or don't use the controller. Let us continue with the TodoList:

```
TodoList =
  ({ todos, onRemove, onChecked }) => {
    return (
      <ul>
        {todos.map((todo) =>
          <ListItem
            todo={todo}
            key={todo.id}
            onRemove={onRemove(todo.id)}
            onChecked={onChecked(todo.id)}
          />
        )}
      </ul>
    )
  }
```

The "TodoList" component differs quite a bit in that in React, we are simply rendering provided to-dos. The TodoList component

is pure presentation instead of implicitly holding the state. In my application, the TodoList exposes a function "addTodo", as well as "getTodos". In a sense, the TodoList is the owner of the collection of to-dos in my application. In the React application, TodosApp (the top-level component) is the owner of this state. Note that our TodoList doesn't have any explicit state, as in local variables, but since each child, or list item, is associated with a controller, the existence of children becomes the state. This is also why our implementation of removing a to-do item was simply "node.remove()", as in removing the DOM node. Let's continue with "ListItem" now:

```
ListItem =
  ({ todo, onRemove, onChecked }) => {
    return (
      <li>
        <input
          type="checkbox"
          value={todo.checked}
          onChange={
            () => onChecked(!todo.checked)}
        />
        {todo.value}
        <button
          onClick={() => onRemove()}>
          Remove
        </button>
      </li>
    )
  }
```

The "ListItem" is similar to "TodoList" in terms of differences with the non-React application. In my application the "ListItem" could remove itself, whereas in React it can tell the "parent" that it wants to be removed. In my application the state ultimately lives in the DOM, for instance whether the item is checked or not. This is a somewhat fundamental difference. In React state tends to live higher up and lower-level components are simply presenting it. And when state changes, parent components are informed and we re-render a sub tree. In my application, the "ListItem" is simply giving a means of inspecting the value, it exposes "isChecked" and "getValue". Rather than the "ListItem" telling the parent component what's happening, the parent component need to ask "ListItem". Finally, let us look at the top-level component, namely "TodosApp":

```
TodosApp = () => {
  [todos, setTodos] = useState([]);

  handleTodoAdd = (todo) => {
    setTodos([
      ...todos,
      {
        id: Math.random(),
        value: todo,
        checked: false
      }]);
  };

  handleTodoCheck = (id) => (updated) => {
    setTodos(todos.map((curr) =>
      curr.id === id
        ? { ...curr, checked: updated }
        : curr
    ));
  }

  handleTodoRemove = (id) => () => {
    setTodos(todos.filter(
      (todo) => todo.id !== id)
    );
  }

  handleSave = () => {
    console.log(todos);
  }

  return (
    <div>
      <TodoAdd onAdd={handleTodoAdd} />
      <TodoList
        onChecked={handleTodoCheck}
        onRemove={handleTodoRemove}
        todos={todos} />
      <button onClick={handleSave}>
        Save
      </button>
    </div>
  );
}
```

The "TodosApp" is perhaps the most telling in how React and my application differs. In React we have state and the components are there to render the state. We control the state and sub components can merely inform us about events based on which we update the state and re-render. This is indeed quite beautiful in a sense, as other components can remain simple and contain very little logic. If we would re-model our state to be indexed by id, we would only need to update the "TodosApp" component, since it owns the state.

In my application the logic is spread out. As an example, the "TodoList" component is in charge of the collection of to-dos, so we can add or get to-dos to/from it. The top-level component that glues everything together is instead devoid of logic, a bit similar to sub components in React. For example, it contains logic such as: "When TodoAdd calls onAdd, add a to-do item to the TodoList component". It simply glues together the interfaces exposed by its children. In the just mentioned example, we're gluing things together almost literary, as the code looks like:

```
todoAdd.onAdd(todoList.addTodo)
```

As always, there are pros and cons with different structures. React's concentration of logic further up the component tree, and simple components further down, can make it easier to reason about state. But my application structure can through its clear interfaces support composition and a structure where many fairly simple parts are put together to form something complex. Depending on how you look at it, my application structure also reduces the amount of state your code deals with, by offloading more to the browser.

Another noteworthy difference is bundle size. After minification and Brotli compression, the React app weighs in at 40590B, whereas my to-do application is only 1000B. While this is an unfair comparison given React being a very mature library, it did surprise me, since despite being an experienced React developer, I didn't feel that I was missing that much that can't fairly easily be added. But, of course, I'd still recommend going with React et al. over a home cooked framework for serious projects. In fact, if bundle size is important, there are alternatives such as Preact, Solid and Svelte [4]. My work here was done as a learning project.

Let us continue the fun and look at some more details of my little framework.

5 COMPONENT LIFETIME AND RENDER

We already covered this a bit when discussing the difference with React. My framework never implicitly re-renders, in fact, the concept of re-render doesn't exist. Sure, one can accomplish it using the "run" method, but, the life-time of a component is generally consistent with its time in the DOM. For instance, a to-do item will only "run" once, and any additional updates will need to happen via calls using its controller, or by it itself listening to events and reacting to them. I believe that a benefit of this "never re-render" concept is that it allows you to use ordinary variables to represent local state. As an example, to implement a button which increments every time you click on it, you'd do:

```
ButtonIncrement = (start: number) =>
  pipe(
    ofElement(
      "button",
      (e) => e.textContent =
        `start value: ${start}`
    ),
    run(({ node }) => {
      let nr = start;
      node.addEventListener("click", () => {
        node.textContent =
          `Clicked ${++nr} time`
          + `${nr !== 1 ? "s" : ""}`;
      });
    })
  );
```

Note the use of "let" as opposed to for instance "useState". However, also note the manual update of "textContent", a price you pay for this simplistic approach.

6 MEMORY MANAGEMENT

One fun and useful experience when building from scratch is that you have to deal with memory management. Initially, I did create some memory leaks here and there. The solution I ended up with was to center everything around the DOM node. This can make it a bit cumbersome to do certain things, for instance to get a controller I must do: "getController(controllers.Type, "myId)", which will load the Node and get the controller (if one exists) from a WeakMap. However, centering everything around the Node essentially eliminated the need for memory management by pushing the work to the garbage collector.

While it was not shown in this paper, there is also an "EventBus", or perhaps it would be more appropriately called "Store". It allows nodes to register and listen for events regardless where in the component tree they reside. For instance, nodes can listen to events from their siblings. This is implemented by storing weak references to nodes. However, I must still store the weak references in a list, which means that the reference itself will not automatically be garbage collected. Luckily, the code to clean up any garbage collected references was trivial.

7 PERFORMANCE AND MEMORY

No deeper analysis has been done, so I can't say much about this. However, using the Chrome Dev Tool's profiler, I did consistently get "scripting" time of 2-3ms, and with the React to-do application 24-29ms. I measured scripting time when just reloading the page. My framework does have the drawback of nesting functions, which might lead to problems. Although, while playing around with it, I was never able to spot any memory or performance issues. I only hit an issue, related to nested functions, when adding a very large number of children at once, but this is likely not a very hard problem to solve.

For memory, I measured using Chrome Dev Tools in an incognito window by looking at the JavaScript VM instance memory

consumption. It stayed around 862KB to 2.1MB for my to-do application. The React application lingered around between 2MB and 2.8MB. Memory consumption was measured while quickly adding and removing nodes manually.

8 WHY BUILD FROM SCRATCH

The goal of this project was learning and having fun. I achieved both! As a developer, it is easy to get very deep into the technology you are using at work. While it is great at work since you can take advantage of that others solved a lot of the tricky problems for you, you also lose out. My understanding is that many new developers are starting by learning something like React, and perhaps never attempt to build something from scratch. I'm convinced that having to think about things like how to model components, how to implement routing, how to make the back button work, how to avoid memory leaks, will make you a better developer even when you get back to the real world and use something battle tested like React et al. Of course, there are many ways to hone one's skills, and building from scratch is just one type of deliberate practice that I think is effective. Regardless if you are senior or junior, I encourage you to try to build a web application in a MTU, and see where it takes you. [5] is another noteworthy example of doing this challenge and ending up with interesting results.

9 FUTURE WORK

On top of just cleaning up the code, in particular trying to express the TypeScript types in a nicer way, I think data fetching would be a fun challenge to work on. Given that the Component already returns a Promise, it is fairly easy to add as a "modifier". However, on top of just loading data, Suspense [3] functionality would be useful. A fallback component that can be shown while data is loading. I believe this can be added in a very lightweight way by piggybacking on DOM nodes and their event bubbling functionality. It could also estimate load time by counting number of requests or using additional load time estimates supplied by the framework user.

A more significant piece of future work would be to redo the whole thing, or something similar, without nested functions. Perhaps in some way inspired by a deep embedding DSL. Alternatively, or in addition, I'd love to try it out in a runtime supporting Tail Call Optimization, and make sure everything works flawlessly with that. Or perhaps see whether something like Fext [7] would work and have an impact.

ACKNOWLEDGMENTS

To my wife, for taking care of the kids while I was playing around with this.

REFERENCES

[1] Jest (https://jestjs.io/).
[2] Lodash (https://lodash.com/).
[3] Suspense (https://react.dev/reference/react/suspense).
[4] Carniato, R. Javascript framework todomvc size comparison, Oct 2021.
[5] Charalampidis, I. Fitting a webapp in a single mtu, Nov 2019.
[6] Gcanti. Gcanti/fp-ts: Functional programming in typescript, 2019.
[7] Lathoud, G. Glathoud/fext: Fast explicit tail calls. in today's javascript!