Tech (/category/tech.html) · 04 April 2016 · 12 min read

A Functional Front-End with React

One trend that has started gaining some serious traction in the past couple of years is functional programming. Unlike an object-oriented approach, which encourages decomposition of a program into 'objects' which relate to a particular domain, a functional approach guides the developer to decompose a program into small functions, which are then combined to form an application. Once confined to the halls of academia, it has recently seen a resurgence as a popular tool to solve a number of the problems that we face in modern development, especially complex applications and distributed systems.

Trailing along behind the functional cheerleaders are a whole host of super-powered functional languages. These include such venerated languages as Haskell, Lisp and Erlang, as well as more modern interpretations including F# (a .NET language), Scala and Clojure (on the JVM), Rust (a server-side language) and Elm (a language based on Haskell for building web front-ends).

While it is wonderful to have such a varied ecosystem of functional languages, the reality is that for day-to-day work we are often restricted in our choice of languages and frameworks. I've been taking a look at some of the characteristics that define the functional paradigm and in this post I'm hoping to show you that you don't have to use a 'pure' functional language to benefit from some insights that the functional approach gives us. By way of demonstration I'll be building a small, functional front-end app in React and JavaScript*.

**I'll actually be using JSX instead of Plain Old JavaScript, as it makes building React components really easy. Most of the concepts will work with standard ES5/ES6*

Before we dive in to building our front-end, I want to touch on a couple of important functional concepts that we can use to make our app more robust.

Pure Functions

Michael Tinning (/mtinning)

@mttinning (https://twitter.com/mttinn

Categories

We all know about functions - they are the building blocks of any program. What makes functional programming so special? Pure functions are part of the answer. A pure function is a function where the return value is determined *only* by its input values, and which does not alter any external state. This has two important implications:

1. For a given input the function will always return the same result, no matter how many times it is called. This means that our application is completely predictable - if we know what inputs it will receive, we can be confident of the outputs.
2. Calling the function will not result in any observable side-effects. We can be certain that we will not experience any undesirable 'emergent behaviour' arising from the interactions between different functions.

Building an application from pure functions enables us to more easily reason about the behaviour of an application.

Immutability

An immutable object's state cannot be changed once it is created. If we want to change the object, we can't simply set a property on the object - we have to return a *whole new object* with that one property changed. This might seem arduous, but it comes with big advantages.

One of the advantages of immutability is in asynchronous programming: if a thread is operating on an immutable object, it can be sure that it is dealing with the latest version of the object. This allows multiple threads to operate simultaneously without worrying about what the other threads are up to.

While the advantages in parallel processing may not be entirely appreciated in JavaScript running in the browser, immutability can still give us big gains. The most apparent in a React application is that we can use simple equality checks to determine if a component needs to be updated: if two variables reference the same object, we can be sure that all data contained within the two objects is identical; if a property is different, the two variables will reference different objects. This is both efficient computationally and cognitively (`===` is always more readable than a deep-equals!).

Another advantage to immutability is that it encourages us to keep our functions pure - if we can't modify an object from inside a function, we can't modify state external to the function. This means that we can be sure of making the most of the very tangible benefits of pure functions.

Currying and Point-Free Programming

A curried function is one which *returns a function* for every argument until the last argument is supplied. As an example consider the following code, which adds two numbers:

We can call `add` with each argument provided simultaneously (invoking the function after each argument). Alternatively, we can store a *partial application* of the function - here, we are storing `add3`. This partial application can later be applied to further arguments.

This is a good approach if we are going to re-use the function multiple times, or if we want to be more explicit about the meaning of a function. A good example is the following, which assigns "modulo 2" to "is odd", resulting in more readable code:

This style of assigning partially applied functions is known as *point-free* programming. A good explanation of point-free programming can be found on the Haskell wiki (https://wiki.haskell.org/Pointfree). As the article notes, care should be taken when using point-free programming with higher-order functions (functions composed of functions) as it can easily result in code that is *more* obfuscated and *less* easy to reason about - precisely the opposite of what we want!

Functional Programming in JavaScript

Although JavaScript inherits its syntax from C-like languages, this is quite misleading. Douglas Crockford (author of *JavaScript: The Good Parts*) notes that (http://www.crockford.com/javascript/javascript.html) "JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java". JavaScript makes it easy to write in a functional style.

In this post, I'm going to be using Ramda (http://ramdajs.com/0.20.0/index.html). Ramda is a functional library that makes it even easier to use a functional approach with ordinary JavaScript. All Ramda functions are curried by default, making it straightforward to use a point-free approach where appropriate, with clean code and no weird-looking syntax (https://youtu.be/m3svKOdZijA?t=218). Ramda also provides functions which will help us to keep our data immutable. Ramda doesn't *guarantee* immutability, as it operates on plain JavaScript objects. Alternatives exist that do guarantee immutability, for example ImmutableJS (https://facebook.github.io/immutable-js/) - Colin Eberhardt (http://blog.scottlogic.com/ceberhardt/) (prolific blogger and CTO of Scott Logic) recently wrote about using ImmutableJS with Angular2 (http://blog.scottlogic.com/2016/01/05/angular2-

with-immutablejs.html). For the purposes of this small project I like Ramda's balance between a clean API and immutable data, and it doesn't play nicely out-of the box with ImmutableJS (though it can certainly be made to (https://medium.com/@drboolean/lenses-with-immutable-js-9bda85674780#.pdy6cc28n)).

Why React?

React is a great tool for building the view layer in an MVC-style web application. Rather than providing a full single page application framework like Angular, it focuses simply on rendering data, and it is *very* good at doing this. This is not an introduction to React, so I won't go into too much depth here (the React Docs (https://facebook.github.io/react/docs/getting-started.html) are an excellent place to start learning about React). Instead, I want to focus on what makes React an excellent choice for our functional front-end.

React applications are separated into components. These components should encapsulate a particular piece of functionality, and can be combined into more complex components and eventually into a whole application. The job of most components is to simply render data that it is given (for example, to display a single comment in a forum), or manage some small interaction (the click of a button, entry of text into a form). This sounds a lot like functional programming!

In fact, React makes it really easy to build components out of functions. Here's the canonical "Hello, world!" example as a React component:

Here our component - `HelloWorld` - is defined as a *pure function*. React can then render it just as it would any other component - we do this by calling `ReactDOM.render` with our component and a DOM element (which we have given the id `'app'`).

That html-looking syntax there is actually JSX. `<div></div>` returns a React element that indicates that an html `div` element should be rendered here. We might also want to render another React component - we can reference our HelloWorld component using `<HelloWorld />`. JSX makes it easy to compose React views inside JavaScript - for a more in-depth explanation, check out the docs (https://facebook.github.io/react/docs/jsx-in-depth.html).

What if we wanted to re-use a component for displaying different data? In the following example, our component greets someone by name:

We've defined a component that can say hello to anyone by name. Our component accepts some properties (`props`) that have been set by its parent component - you

can see this inside the `ReactDOM.render` method - we are passing the prop `name="Bob"` to the `GreetSomeone` component.

In general, this is how components are built in React - a parent component will define some props that are passed down to its children. When `ReactDOM.render` is called React builds a virtual representation of the DOM and renders to the real DOM. It looks like every time we call `ReactDOM.render` the entire real DOM is re-rendered, but this is not the case: behind the scenes React builds a new virtual representation of the DOM and does some very clever comparisons to work out what changes to make to the real DOM. Again, the docs (https://facebook.github.io/react/docs/reconciliation.html) have an excellent explanation of how this comparison - normally $O(n^3)$ - can be turned into an $O(n)$ problem with a couple of assumptions. This makes React's approach of re-rendering the whole virtual DOM practical.

This is all great stuff, as it allows us to build *really simple* components that are *easy to reason about*.

So we've built a component using just functions - let's see what happens if we stick a few together!

Functions as Components

I'll be building a simple forum app which displays comments. As far as possible I'll try to *keep it functional* - that is, use pure functions with immutable data. We'll start by breaking down our app into pure functional components that will display our comments, and then we'll plug in some data.

Let's start with the top-level component - `App` . Just as before, we can define our component as a pure function. Our inputs will be the title and subject of our forum discussion, and our list of comments. On top of that, we'll also display our title, subject and list of posts using `Title` , `Subject` and `PostList` components like good React programmers.

Pretty straightforward, right? It's really easy to see what's going on here. Our `App` component is rendering `Title` , `Subject` and `Post` components inside a `div` element. It's passing down the `title` to the `Title` component, the `subject` to the `Subject` component and the `posts` to the `PostList` component.

So far, so easy. Let's take a look at `PostList` - it's calling `postComponents` to generate the `Post` components from the array of posts. What does this `postComponents` look like? We can use currying here:

This uses Ramda's `map` (http://ramdajs.com/0.20.1/docs/#map) function to partially define a function that maps the given expression (a conversion from post to `Post` component) over an array of posts.

For those unfamiliar with the functional style, this can be quite difficult to follow so I'll break down what it's doing. We want our `postComponents` function to convert an array of posts into an array of components. A first pass at a function that does this might start with an empty list and add components to it one at a time:

This is fine, but it is quite verbose and not very readable. We need to follow the flow of the function through variable initialisation and loop to determine that it will result in an array of `Post` components.

As a second pass, we might realise that JavaScript's `Array.prototype.map` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) is up to the job:

This is better, however it is still verbose. We need to define a whole function in order to define our customisation of the `map` function. This is where currying can help us - it allows us to store references to *partially defined* functions that we can use later.

We could just use the `posts.map` function directly inside our `PostList` component, however storing a reference to the function promotes code re-use and allows us to keep our code very readable.

Now that we have our list of `Post` components, let's take a look at the `Post` component itself. We need to display an author and a comment - so let's break those up into components too.

Take a look at that - they're all pure functions! The `Post` component has an `author` and a `comment` in its `props`, which it passes down to the `Author` and `Comment` components respectively.

Our `Author` component is doing something similar to the `PostList` component - it's calling the `authorComponents` function. This function should return an array of components to display all properties that are present on the `author` object. This is defined as follows:

Here, we're using a few JavaScript language features to make our code concise and readable.

The ES6 arrow function (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) syntax replaces the `function` declaration. This makes it

semantically clearer that "`author` maps to the following array" or "`author` maps to the following element".

Our use of the ternary operator (?:) (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operc makes it easy to shorten `if-else` blocks in order to make our code more readable.

Finally we recognise that when a property is present on an object, the first argument to the ternary expression will evaluate to true; if it is missing, it will evaluate to false. This makes it easy to conditionally include or exclude React elements based on properties in our object.

Let's recap - we now can now display a forum and thread of posts, with each post displaying some information about the author and the comment. And we've done all of this using nothing but functions. Not only that, but we haven't modified any state outside of our functions. This has profound implications for testing - it means that we can write tests for our functions and be sure that no external interference will change their behaviour - that's pretty impressive.

Generating the Sample Data

For this example I'm going to simulate a few comments from a few authors.

Here, we have a couple of interesting functions to generate sample data. `nthWrapped` uses Ramda's `pipe` (http://ramdajs.com/0.20.1/docs/#pipe) function to build a function that takes the modulo of the input with the length of the array (using the handy `flip` (http://ramdajs.com/0.20.1/docs/#flip) function to swap the order of the first two arguments of `mathMod`) and then returns the element at the resulting index. `nthPost` takes an index and merges the result of calling `nthWrapped` on both the `authors` and `comments` arrays into a single result. We can use this as an argument to Ramda's `map` function to generate another function that returns posts in a given range.

We have effectively provided a "stub" version of the `postsInRange` function - we could easily swap this out for a real version. This is one of the great advantages of functional programming - because each of the functions are small and simple, and because they do not modify anything external to the function itself, they are easy to replace.

Rendering the App

Once we have our data, we can simply call `ReactDOM.render`, as we saw in the examples. We can use Ramda's `range`

(http://ramdajs.com/0.20.1/docs/#range) function. I'll then render them into the html element with `id="app"`. Here's what that looks like:

Bonus - RxJS Stream

Great - we've got an app that renders comments! But we can do better than that - we might be looking at a really popular post, and we want to view comments as they come in. Wouldn't it be nice if we could render a stream of incoming comments? Thanks to the functional style in which we have written our app, we can!

RxJS (https://github.com/Reactive-Extensions/RxJS) is a library that makes it easy to handle streams of data in JavaScript. It is written in a functional style and treats streams of data in a way analogous to arrays of data. This makes it very easy to plug in to our existing implementation.

Instead of rendering posts in a given range as in the previous example, we want to call `render` every time a new post is available from our stream. We can let React do the heavy lifting to ensure that a minimal set of updates is applied to the DOM.

Here's the code:

We set up a source for our posts. This is simulating a post coming in every two seconds, using the same `nthPost` function that we defined in our previous example. This could be swapped out for real data in a production app.

We then set up a subscription to our event source. This is doing a couple of interesting things. `scan` (https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/scan.md) is a method on the `Rx.Observable` `postSource`. It acts very much like the `reduce` function - we provide an initial value and an accumulator, and it returns the result of calling the accumulator over all of the elements in the stream. The difference between `reduce` and `scan` in RxJS is that `scan` will return intermediate values - it will return every time the stream emits an item - whereas `reduce` will wait until the stream has completed. Our accumulator here is simply appending the next value to our current accumulated value.

The result after `scan` is called is an observable that emits an array containing all the posts emitted so far, every time a new post is added.

`map` is then called to return a React element that wraps our posts - here defined as the `app` function. This also returns an observable, which we can subscribe to in order to render our posts.

This amazingly small amount of code achieves something quite complex - we now have a dynamic application!

Is Functional Programming The Way Forward?

I think that we can learn a lot from the functional paradigm and that it can result in cleaner, more readable and more robust code that is also easier to maintain. Though this post is only meant as an introduction, hopefully it will inspire you to look into functional programming and perhaps use some of the approaches outlined here.

Where to go next? If you're using React, then redux (https://github.com/reactjs/redux) provides a functional approach to managing state for your apps. It has a large and growing ecosystem (https://github.com/xgrommx/awesome-redux) of tools and middleware and comes with features such as time-travel and undo-redo practically for free - thanks to its use of immutable states. Colin's written a post on that too! (http://blog.scottlogic.com/2016/01/25/angular2-time-travel-with-redux.html).

There are frameworks out there that take a much stricter approach to building front-ends in a functional style - elm (http://elm-lang.org/) takes its cues from Haskell and provides a pure functional experience for programming in the browser, while Reagent (https://reagent-project.github.io/) and re-frame (https://github.com/Day8/re-frame) allow you to build front-ends with ClojureScript (https://github.com/clojure/clojurescript).

For now, I am very happy that it is so easy to strike a balance between practicality and functional style in JavaScript. Pure function components in React make it really easy to put together simple, easily understandable applications. The approaches here do not answer every problem - stateful components and lifecycle hooks, for example, will still require fully fledged components at some level (for e.g. optimising efficiency, manipulating DOM elements). I'd like to explore this more in the future, and see what other examples we can take from functional programming.

You can check out a version of this app on jsbin (https://jsbin.com/gejenacuzi/edit?html,js,output).

Read more

View more by Michael Tinning (/mtinning)

View all Tech articles (/category/tech.html)

