What I’ve learned so far about React

Abhishek Tiwari

Citation: *A. Tiwari*, ”What I’ve learned so far about React”, Abhishek Tiwari, 2017. doi:10.59350/a4vzh-dhy95

Published on: September 25, 2017

Over the weekend, I took a shot to build something new in React. React is a JavaScript library to build user interfaces. React was open sourced by Facebook and since then it has gained popularity over other frontend frameworks such as Angular, Vue, etc. To start, I have a good experience in JavaScript and I have built several applications using Node, Express, and Angular. Unfortunately, React landscape is way too complex. It is quite a different ecosystem and a difficult one for beginners.

Things became very complicated when decided to use an existing React admin theme as the foundation for my project. I know what you are thinking, I really got myself in a mess. But that admin template taught me few things which otherwise I have not learned as a React beginner.

Knowledge map for React

In this post, I want to cover few things which will help you to develop beyond hello world React application.

ECMAScript 2015

ECMAScript 2015 (ES2015 also known as ES6) should be the obvious choice for writing React applications. Unfortunately, ES2015 is not fully supported by the browser so you will need to use plugins such as Babel or Traceur. These plugins allow you to use new ES2015 syntax, right now without waiting for browser support. They achieve this by compiling or transforming ES6 syntax to ES5. In addition, they offer polyfill functionality to support new globals such as Promise or new native methods like String.padStart. They can also support ES.next features i.e. proposals such as decorators, async-await, and static class properties. You can read a detailed overview of various ECMAScript 2015 features here. In my opinion, when it comes to developing React applications Babel is more fit for purpose and popular compared to Traceur. Babel also includes React specific batteries i.e. React presets to work with JSX and Flow. Babel also plays really well with your existing JavaScript build systems including Grunt, RequireJS, Gulp, and Webpack.

You can try various ECMAScript 2015 features in the browser using JSBin without any installation.

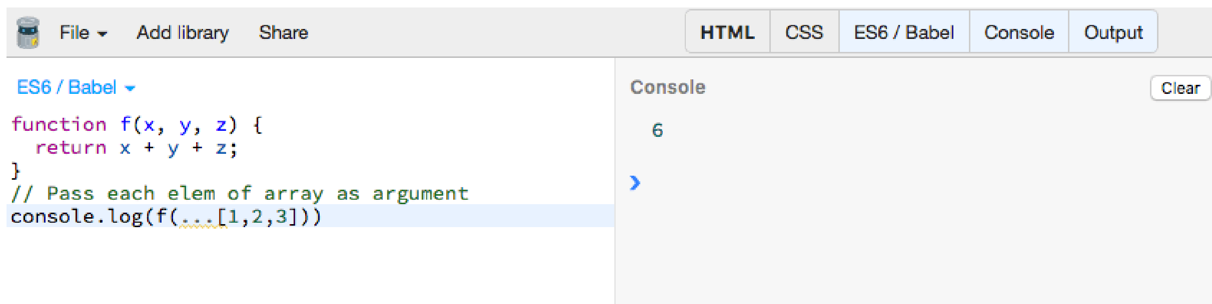

Figure 1: Try ECMAScript 2015 at JSBin without any installation. You can select Babel or Traceur as you JS compiler.

UI as Function of State

React takes a functional approach to UI. As described by Alexander Beletsky, React gives us the ability to treat UI as a function of the state. In this case, function F is a stateless functional component and state S represents the immutable state of data models, DOM, and side-effects.

```
UI = F(S)
```

In addition, Redux represents a function that takes current state S and action as input and produces next state Snext as output. This function is called *Reducer*.

```
Snext = R(S, action)
```

In other words, when we combine React with Redux, with each action the whole UI is re-rendered.

```
Given UI = F(S) and Snext = R(S, action)

UI = F(R(S, action));
```

You may want to watch this video which describe the first principle of Redux - state of your whole application is stored in an object tree within a single store.

> everything that changes in your application, including the data and the UI state, is contained in a single object, we call the state or the state tree

A state is read-only and immutable. The only way you can change the state of the store is by dispatching an action on it. It is important to understand the distinction between Redux's store and React's state. React state is local and ephemeral [1] [2].

[1] How to choose between Redux store and React state?
[2] Where to Hold React Component Data

> Redux store is great for keeping application state rather than UI state. UI state and transitory data are best served by React state.

Components and Props

React components are the independent but reusable piece of functionality. Props are input. On a high-level, a component transforms props into UI. In React, you can describe components three different ways,

- Using `React.createClass` (deprecated hence not recommended but it supports mixins to components, Mixins aren't supported in ES6 classes [3])
- Extending `React.Component` or `React.PureComponent` (compatible with ES6, have a local state and lifecycle methods [4])
- Stateless functional components (only has a render function with optional argument of props, no state [5])

So following class-based component ~~~ import React from 'react';

class App extends React.Component { render() { return

Hello World

;}}

export default App;

```
is same as below stateless functional component[^5] [^6],
```

import React from 'react';

const App = () =>

Hello World Stateless

;

export default App;

```
```

[3] React.createClass versus extends React.Component
[4] State and Lifecycle
[5] How do you decide, how do you choose between these three based on the purpose/size/props/behavior of our components?

```
More importantly, whether you use class-based components or stateless
    functional components, it should never modify its own props. Such as
    function is called *pure* because they do not attempt to change their
    inputs. Pure functions are predictable and they always return the same
    output for the same inputs. On the opposite, *impure* functions may
    have some observable side effects [^8].

As described above React offers a powerful composition model. It is highly
     recommended to use components as composable units instead of
    inheritance. Components can be stateless. If we are given three
    components `C1(S)`, `C2(S)`, and `C3()` where C3 stateless function, as
     described below we can compose them in many different ways.
```

C = f(C1(S), C2(S), C3())

where f can be a higher-order component transform. ~~~

React-Router

React-Router offers declarative routing for React applications. It keeps your application UI in sync with the URL by invoking right kind of components. React-Router v4 is a complete rewrite and everything is now *just components* [6]. React-Router supports dynamic routing i.e. routing that takes place as your React application is rendering, not in a configuration or convention outside of a running application. React-Router also plays really nice with Redux.

dva framework

dva is a lightweight ELM-style front-end framework based on Redux, Redux-Saga, and React-Router. At first instance, *dva* as framework looks quite promising but it has its own share of problems. First of all, it was developed by engineering teams at Alibaba in China. So most of the documentation is either Chinese or not very complete. Because it's a wrapper on top of popular React libraries, you are at the mercy of developer to update them. For instance, *dva* is still stuck at React-Router v3.

[6] A detailed comparison of React-Router v4 against v2/v3

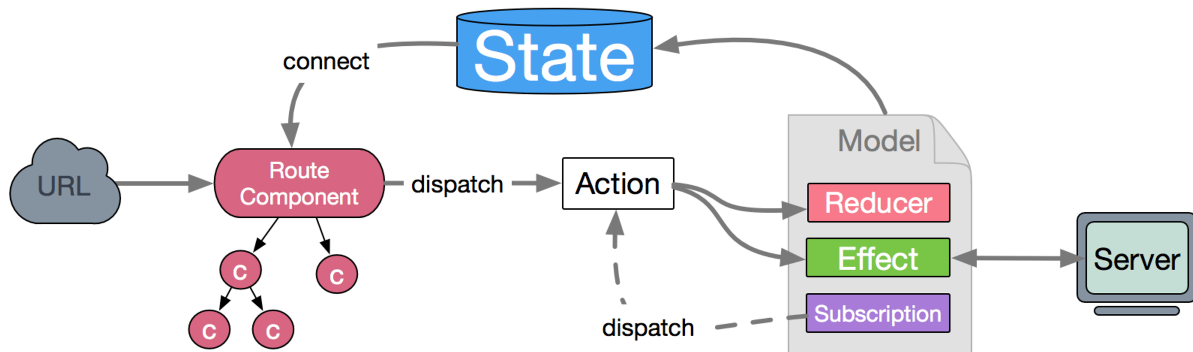

Figure 2: dva concepts and how they connect together. Some of these concepts are ELM inspired.

Overall, *dva* is easy to use. The companion tool *dva-cli* is quite nifty and it can not only generate application skeleton but also various application building blocks such as routes, models and components.

You start by defining the models. When you create model using *dva-cli* it generates a placeholder model. Then you design and implement the components. You update models with reducers, effects and subscriptions. Finally using *react-redux* connect functionality you wire models and components.

After connect, component can use the data from model, and model can receive actions dispatched from component. One last thing, define router to connect URL with UI components.

Container/component Architecture

Container/component architecture is one simple but powerful pattern [7] [8]. This is particularly useful if you are using Redux. Container components are responsible for how things work. Often they are stateful. Presentational components are dumb React components responsible for how things look. They depend on container components for data and behavior, hence generally stateless. When implementing this pattern, we can either separate these component types in two different folders (`container` and `components`) or use a naming convention which reflects the connection between two type of corresponding components.

```
StockWidgetContainer => StockWidget
TagCloudContainer => TagCloud
```

[7] Presentational and Container Components
[8] Container Components

React UI frameworks

There are several reusable React UI frameworks freely available on the web[9]. These frameworks provide reusable component libraries enabling you to build your React application. For those who are familiar with Bootstrap, React-Bootstrap offers reusable UI components with look-and-feel of Bootstrap, albeit with much cleaner code. If you don't like Bootstrap then you can try Ant Design, Semantic UI React, or Material-UI. Ant Design offers built-in internationalization for components and works really well with *dva* framework. Obviously, with React-Bootstrap, Material-UI and Semantic UI React you have to use an internationalization library such as react-intl.

Styled components vs. Style Loaders

This was an eye-opener for me. Traditionally React developers have used various style loaders to implement styling. With style loaders, you write your component style code using CSS, Sass or Less in a file and then load them using your build system.

```
// Button.css
.danger {
  background-color: red;
}

// Button.js
import React from 'react';
import styles from './Button.css';

class Button extends React.Component {
  render() {
    return <button className={styles.danger}>Click me</button>;
  }
}
```

Styled components basically allows you to write actual CSS inside your JavaScript. It removes the mapping between styles and components.

```
import React from 'react';
import styled from 'styled-components';

// Create a <Title> react component that renders an <h1> which is
// centered, palevioletred and sized at 1.5em
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;
```

[9] Best UI Frameworks for your new React.js App

```
// Create a <Wrapper> react component that renders a <section> with
// some padding and a papayawhip background
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;

// Use them like any other React component – except they're styled!
class Button extends React.Component {
  render() {
    return (
      <Wrapper>
        <Title>Hello {this.props.name}, your first styled component!</
           Title>
        ...
      </Wrapper>
    );
  }
}
```

React boilerplate

Finally, if you want to create skeleton React application, you can use create-react-app. This will create React application without build configuration. You need to create your own build configuration which in my opinion a lot of work. One of the reasons I liked *dva* was its ability to generate React application skeleton with build configuration as well various building blocks of the application. Alternatively, you can use react-boilerplate. Using *react-boilerplate* can create components, containers, routes, selectors, sagas, corresponding tests - right from the CLI. It comes with baked-in build configuration. I definitely recommend *react-boilerplate* only if you are starting a big React project. Otherwise, for small React applications use *create-react-app*.