Future UI

Chris Biscardi

Contents

Chapter 1

What is this book?

This book will help you explore the state of cutting edge UI/UX development. We will focus on React, Webpack, both Flux and "Flux-inspired" architectures, Routing and Universal Applications.

1.1 Tools

The TC39 categorises proposals into 4 stages:

- Stage 0 - Strawman
- Stage 1 - Proposal
- Stage 2 - Draft
- Stage 3 - Candidate
- Stage 4 - Finished

All of the code in this book is written using a stage 0 babel transpiler. This means we have access to future ECMAScript features all the way up to the Strawman stage (currently considered "ES7"). We do this because this is a book about writing UI in the future and to explore that, we will use what the cutting edge looks like today as our starting point.

1.2 Code examples

The simple code examples often use @gaearon's react-transform-boilerplate which provides a simple base to jump in and explore example code.

Chapter 2

Rendering

In this book, we will be focusing on React for controlling the rendering of our applications. According to the React site, React is:

> A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES

Notice that there is no mention of the DOM because React is not tied to a specific render target. In practice, this means we can render to DOM, SVG, Canvas and even native iOS components. Most importantly, React is a component model for abstractions.

2.1 Virtual DOM

The abstraction used which allows us to render to multiple targets is referred to as the Virtual DOM. Simply speaking, the Virtual DOM is a tree structure which allows us to diff against previous versions of our rendered application.

In practice one of the benefits is that we have to worry less about DOM insertion optimizations such as batching updates. A spec-compliant synthetic event system means that bubbling and capturing works the same across browsers.

The React docs explain the implications well:

> **Event delegation:** React doesn't actually attach event handlers to the nodes themselves. When React starts up, it starts listening for all events at the top level using a single event listener. When a component is mounted or unmounted, the event handlers are simply added or removed from an internal mapping. When an event occurs, React knows how to dispatch it using this mapping. When there are no event handlers left in the mapping, React's event handlers are simple no-ops. To learn more about why this is fast, see David Walsh's excellent blog post.

2.2 Basics

2.2.1 A Simple Component

The following component renders a heading. We can then use `react-dom#render` to attach to the body of the `document`.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

export default class Simple extends Component {
  render() {
    return <p>Hello World</p>;
  }
}
```

```
render(<Simple/>, document.body);
```

To see this component in action, run the following:

```
git clone git@github.com:future-ui/basics.git
cd basics && npm install
npm start
```

then go to `localhost:3000`

The example is hot-reloadable, which means that you can edit the `src/App.js` file and see the changes in the browser immediately. You should play around a bit and see what happens.

2.2.2 A More Complex Example

We want to display a counter which we can adjust using various buttons.

```
import React, { Component } from 'react';

export default class Counter extends Component {

  state = {
    count: 0
  }

  increment = (e) => {
    const { count } = this.state;
    this.setState({
      count: count + 1
    });
  }

  decrement = (e) => {
    const { count } = this.state;
    this.setState({
      count: count - 1
    });
  }

  render() {
    return (
      <div>
        {this.renderControls()}
        <section>
          <h1>My Awesome Counter!</h1>
          <p>Counters are an integral part of counting things!</p>
        </section>
      </div>
    )
  }

  renderControls = () => {
    const { count } = this.state;
    return (
      <section>
        <button onClick={this.decrement}>-</button>
        <span>{count}</span>
        <button onClick={this.increment}>+</button>
      </section>
    );
  }
}
```

If you take the above example and paste it in to the App.js file from before, you will see it render roughly as:

```
<div data-reactid=".0">
  <section data-reactid=".0.0">
    <button data-reactid=".0.0.0">-</button><span data-reactid=
    ".0.0.1">7</span><button data-reactid=".0.0.2">+</button>
  </section>

  <section data-reactid=".0.1">
```

```
    <h1 data-reactid=".0.1.0">My Awesome Counter!</h1>

    <p data-reactid=".0.1.1">Counters are an integral part of counting
    things!</p>
  </section>
</div>
```

Each of the `data-reactid`s is a node in the tree.

2.2.3 render

render is the most common function to be written on a React Element, partially because it is a `required` function. A `render()` function uses `props` and `state` to return a virtual representation of a node.

In the following case, we ignore `this.props` and `this.state` and instead render a constant result node representing a paragraph tag with some text.

```
class SimpleRender extends Component {
  render() {
    return (
      <p>Some Content</p>
    )
  }
}
```

Moving up in complexity, we can use both `this.props` and `this.state` to return a more complex element.

```
class ComplexRender extends Component {

  render() {
    const { name } = this.props;
    const { age } = this.state;
    return (
      <div>
        <h3>{name}</h3>
        <p>{name} is {age} years old.</p>
     </div>
    )
  }

}
```

As shown in the code above, the usage of state and props is incredibly similar. The difference between the two is how (and where) the data is managed. We go into this more in the chapter on State Management.

2.2.4 JSX

JSX is an XML-like extension to ECMAScript which is currently a Draft Speficiation. JSX looks a lot like HTML, but it has some crucial differences. For one, JSX transforms to JavaScript. An example of this transform, given by the React docs is shown below.

```
// Input (JSX)
var app = <Nav color="blue" />;
```

```
// Output (JS):
var app = React.createElement(Nav, {color:"blue"});
```

Our more complex example from before (`ComplexRender`) transforms into the following render function when using babel for the transform.

```
function render() {
  var name = this.props.name;
  var age = this.state.age;

  return React.createElement(
    "div",
    null,
    React.createElement(
      "h3",
      null,
      name
      ),
    React.createElement(
      "p",
      null,
      name,
      " is ",
      age,
      " years old."
    )
  );
}
```

Knowing that JSX is transformed into JavaScript enables us to understand one of the restrictions of our render function: A render function can only have a single JSX root node as the return value. This is because of the fact that JSX transforms into JS, which means returning multiple root nodes would mean the same thing as returning multiple functions.

2.2.5 Accepting Props

To illustrate the role of Props in React's render(), we will revisit the Counter example from earlier in this chapter. In the following example, we initialize the state of our counter to 0 and provide two functions to increment and decrement the counter. We also write a second Element, whose job is to render a number in "human" form as well as a third element, whose job is to render and dispatch state changes.

```
import React, { Component, PropTypes } from 'react';
const { number, func } = PropTypes;
import numeral from 'numeral.js';

class HumaneNumber extends Component {

  static propTypes = {
    num: number.isRequired
  }

  humanize(num) {
    return numeral(num).format('0,0');
  }

  render() {
    const { num } = this.props;
    return (
      <span>{this.humanize(num)}</span>
    )
  }
}

export class CounterController extends Component {

  static propTypes = {
    increment: func.isRequired,
    decrement: func.isRequired
  }

  render() {
    const { increment, decrement } = this.props;
    return (
      <section>
        <button onClick={decrement}>-</button>
        <button onClick={increment}>+</button>
      </section>
    );
  }
}

export default class Counter extends Component {

  state = {
    count: 0
  }

  increment = (e) => {
    const { count } = this.state;
    this.setState({
      count: count + 1
    });
  }

  decrement = (e) => {
    const { count } = this.state;
    this.setState({
      count: count - 1
    });
  }

  render() {
    return (
      <div>
        <CounterController increment={this.increment}
                           decrement={this.decrement} />
        <p>Clicked <HumaneNumber number={this.state.count} /> Times</p>
      </div>
    )
  }

}
```

In building this set of elements, we have:

- Pushed event dispatch out to the leaf nodes while maintaining state control in a single location (the parent node). This means we only have one place to look for all of the possible ways state can be altered.
- Encapsulated the display of human-readable numbers into a dumb element.
- Separated the display of the control panel from the definition of state-adjusting logic.
- Clearly defined APIs for our dumb elements.

The parent node is specifying "This is what you can do to the state of the application". Our child (dumb) elements are specifying their own API contract using PropTypes, which details their requirements. The child nodes are responsible for rendering the UI and defining how the user interacts with the rendered UI.

2.2.6 Render Targets

As we said earlier, part of React's power comes from being an abstraction layer. One of the most interesting facets of this is the ability to abstract render targets.

TODO: Talk briefly about constructing the previous examples for iOS.

2.3 Lifecycle Methods

2.3.1 Mounting

2.3.1.1 componentWillMount

2.3.1.2 componentDidMount

2.3.2 Updating

2.3.2.1 componentWillReceiveProps

2.3.2.2 shouldComponentUpdate

2.3.2.3 componentWillUpdate

2.3.2.4 componentDidUpdate

2.3.3 Unmounting

2.3.3.1 componentWillUnmount

2.4 Advanced

2.4.1 Autobinding

2.4.2 refs

2.4.3 statics

2.4.4 context

2.4.5 keys

Chapter 3

State Management and Abstraction

When building UI, one of the most complex pieces is the state of the application. React provides two core ways to manage State, `this.state` and `this.props`. We will also talk about a third way to manage state: Redux. It turns out that the way elements abstract functionality has a large implication for how state is managed.

As we mentioned before, one of the key differences between `this.props` and `this.state` is where the data gets managed. `this.props` gets passed in to an element from "outside". This means that props aren't controlled by the element displaying them.

3.1 Dumb Elements and propTypes

To see why the concept of props is so useful, we are going to explore the concept of Dumb Elements. In accordance with @dan_abramov's definition of Dumb Elements. They are elements which:

- Have no dependencies on the rest of the app, e.g. Flux actions or stores.
- Often allow containment via this.props.children.
- Receive data and callbacks exclusively via props.
- Have a CSS file associated with them.
- Rarely have their own state.
- Might use other dumb components.
- Examples: Page, Sidebar, Story, UserInfo, List.

Talking briefly about our `HumaneNumber` example, we have created an element which abstracts the choice of "Which format do I use to represent numbers in my application?". This has a number of benefits including reducing the opportunity to use the wrong format and reducing the mental burden of writing the UI. Reducing mental burdens, even in this simple way, is critically important to opening up opportunity to focus on business logic.

propTypes define the API supported by a dumb element. For example, by looking at the following PropTypes, we can determine a fair bit of information about the element we're using.

```
static propTypes = {
  title: string.isRequired,
  isEligible: bool,
  age: number.isRequired
}
static defaultProps = {
  isEligible: false
}
```

The above element takes three possible attributes. `title` is a `string`, and we have no good default value for a `title` so it's required. We also have an `age` attribute, which is a required `number`, and an `isEligible` attribute, which is an option `boolean`. Notice how we're using `defaultProps` to fill in the gaps for those properties which we don't require to be passed in when using the element. `defaultProps` are a great way to reduce the noise on the consumer's side when defining an element's API.

3.1.0.1 Using PropTypes to encapsulate third party libraries

`HumaneNumber` is nice because it provides an opinionated way to display numbers in an application, but we could use it to greater effect if we encapsulated Numeral.js in it's own element. (note that we've skipped some of the Numeral API and focused solely on rendering formatted strings).

```
// HumaneNumber.js
import React, { Component, PropTypes } from 'react';
const { number, oneOf } = PropTypes;
import numeral from 'numeral.js';

export class Numeral extends Component {

  static propTypes = {
    format: string,
    num: number.isRequired
  }

  static defaultProps = {
    format: '0,0'
  }

  mkNumeral = () => {
    const { num, format } = this.props;
    return numeral(num).format(format);
  }

  render() {
    return (
      <span>{this.mkNumeral()}</span>
    )
  }
}

export default class HumaneNumber extends Component {

  static propTypes = {
    num: number.isRequired
  }

  render() {
    const { num } = this.props;
    return (
      <Numeral num={num} format='0,0' />
    )
  }
}
```

We can now import and use the simplified `HumaneNumber` or the more powerful `Numeral` element in another place in our application.

```
import React, { Component, PropTypes } from 'react';
const { number } = PropTypes;
import HumaneNumber, { Numeral } from './HumaneNumber';

class TimeAndMoney extends Component {

  static propTypes = {
    days: number.isRequired,
    money: number.isRequired
  }

  render() {
    return (
      <div>
        <p>In <HumaneNumber num={days} /> days; you've made
        <Numeral num={money} format='($ 0.00 a)' /> dollars</p>
      </div>
    )
  }
}
```

If we rendered money in enough places in our application, it might also make sense to build a `<Currency currency='usd'/>` element to display different countries' currency. For now, the raw ability of the `<Numerical/>` element serves any additional number-rendering need that may crop up in our application.

Notice that the `<Numeral/>` element will behave exactly the same as the `<HumaneNumber/>` element if you leave out the `format` because the default format for `<Numeral/>` is the same. This brings up an interesting design decision when building dumb elements. How much of the API should you expose?

`<HumaneNumber/>` has a much more restrictive API. The benefits of this include always knowing how a `<HumaneNumber/>` will treat a `num` attribute and a smaller API surface area.

`<Numeral/>` provides more power, but has a larger API surface area and is easily modified to show numbers differently.

Often, I'll opt for a more opinionated API such as `<HumaneNumber/>` because it clearly expresses the intent I had when I built the UI. There are no questions about whether I forgot a comma, etc in the formatting string because of the restrictive API. More powerful APIs such as `<Numeral/>` are *always* nice to have in case an unexpected situation arises. One can even build more opinionated elements from the more powerful elements. The API can be optionally restricted for each additional element while the more powerful or harder-to-use API is hidden for advanced users.

3.1.1 PropTypes

It is a good idea to declare `propTypes` for every element which takes props. There are a number of benefits to doing this including documentation and the fact that if you specify the types correctly, React will warn you of data which is passed in that doesn't match the expected signature.

One real-world and slightly messy application of propType warnings is as a canary for changing APIs. If the propTypes match up with the values that are expected to be received from an API, and that API changes: The propTypes can serve as a late warning that the data is not in the structure the application expects.

3.2 Smart Elements

3.2.1 State

3.3 Redux

Chapter 4

Routing

React Router is a project which grew up with the React Community. As a result, it is one of the best Routers to use in conjunction with React.

4.1 Link

4.2 Universal

4.3 Animations

4.4 Transitions

Chapter 5

CSS

We can not talk about Future UI without talking about the leaps the CSS world has been taking lately.

5.1 Cascading Out of Control

One of the biggest sources of complexity in large CSS codebases is the cascade. To that end, we should eliminate it. This means only styling classes, not tags or IDs.

5.2 Naming Things

CSS has a single namespace. This leads to a large amount of time spent focusing on just naming classes. BEM, OOCSS and SUITCSS are approaches that have been developed to solve this problem but we can do better. We can eliminate namespace conflicts with `css-modules`.

5.2.1 css-modules

CSS modules is (TODO: insert quote from css-modules github)

TODO: explain how css-modules avoids namespace conflicts.

5.2.1.1 ICSS

TODO: briefly touch on what ICSS is and how it relates to css-modules

5.3 PostCSS

PostCSS is the next step in CSS preprocessors. (TODO: insert website quote here).

5.3.1 cssnext

A pack of PostCSS plugins enabling quickly getting up to speed.

5.3.2 postcss-local-constants

5.3.3 postcss-nested

5.3.3.1 When should CSS be nested?

TODO: expand this; Almost never. Only in advanced use cases such as “container” elements or when encapsulating something like `<Markdown/>` using highlight.js which requires global classes.

Chapter 6

Redux

6.1 Basics

6.1.1 actions

6.1.2 reducers

6.1.3 store

6.1.4 middleware

6.2 Higher Orders

6.2.1 Stores

6.2.2 Components

6.3 Dumb Components

6.4 Smart Components

6.5 Normalizr

Chapter 7

Utilities

7.1 keymirror

keymirror is used to create an object with values equal to its key names.

7.2 invariant

invariant is a mirror of Facebook's invariant, which is used in React and flux.

7.3 classnames

classnames is a simple javascript utility for conditionally joining classNames together.

7.4 humps

7.4.1 camelizeKeys

> hump is an underscore-to-camelCase converter (and vice versa) for strings and object keys.

– mention usage in tandum with normalizr/APIs

Chapter 8

Reference

8.1 PropTypes

8.2 Lifecycle Methods

8.3 Redux