Table of Contents

Philosophy

Focus on Desired User Interface, not how to get there

To quote from the react documentation

> In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

So, instead of thinking in terms of changes

```
function updateClock() {
  const clockEl = document.getElementById("clock");
  clockEl.innerHTML = new Date().toLocaleTimeString();
}
```

We just describe the clock:

```
function tick() {
  const element = <span id="clock">{new Date().toLocaleTimeString()}.</span>;
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

Everything is nested Components

In React, everything that appears on screen is a Component.

Components are composed of other components. A good example.

Each component should be stand-alone. That is to say, a component knows about the DOM elements and components that make it up, but it doesn't know anything about the contex in which it is being rendered.

Components always have a render function, which describes (using JSX) how the component should be rendered based upon the components properties.

JSX

Multi-line JSX Expressions should be wrapped in `()`'s

Though optional in some specific cases, it's best to always include these parenthesis

```
const element = (
  <h1>Hello, world!</h1>
  <p>I'd like to welcome you to the world of React</p>
);
```

`{}`'s can take any JavaScript expression

From the simple

```
const name = "Ian"
const element = <h1>Hello, {name}</h1>;
```

To the complex

```
const firstName = "Ian";
const lastName = "Bentley";

const element = (
  <h1>
    Hello, {firstName} {lastName === undefined ? "Smith" : lastName}
  </h1>
);
```

Gotchas

- Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

```
// Yes!
const element = <div tabIndex="0"></div>;

// Yes!
const element = <img src={user.avatarUrl}></img>;

// No!
const element = <img src="{user.avatarUrl}"></img>;
```

- By default, React DOM escapes any values embedded in JSX before rendering them

Components

State vs Prop

Every component by default has props - the simplest function based components are defined as:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

> State is similar to props, but it is private and fully controlled by the component. `props` don't change over time, but `state` does.

In order to use a component with state, it needs to be a `class` component. You can define state by initializing a state object in the component's constructor.

```
class Clock extends React.Component {
  constructor(props) { // Notice that `props` is still here!
    super(props);
    this.state = {date: new Date()};
  }
  render() {
    return <h2>It is {this.state.date.toLocaleTimeString()}.</h2>;
  }
}
```

While props are *immutable*, state changes over time. With *every* change to state, react will re-render the component.

`props`, `state`, `render`, as well as the lifecycle callbacks are reserved words on `React.Component` classes, but feel free to add other methods to support more advanced functionality in your react component classes.

State Gotchas

Never modify state directly

With the exception of setting `this.state` in your constructorm, you should never modify `state` directly. Doing so will not result in a render, and it's a bad practise. Instead use `this.setState({key: value});` to update one or more state variables.

Be careful if you're using the current `state` value in a call to `setState`

```
// Bad
this.setState({
  counter: this.state.counter + 1,
});

// Good
this.setState((state, props) => ({
  counter: state.counter + 1
}));
```

The reason for this gotcha is more subtle. `setState` is a mysterious beast, and react may change the values of `this.state` asynchronously underneath you.

Only the current component can see it's state

Other instances of the same component will have isolated state's, the parent of a component don't have access to a child component's state, etc.

Components that render a list

Utilize `map` in order to create an array of elements, which you can then reference in the full JSX.

Make sure that you assign a key to each element in the list. This key is like a `primaryKey`, and allows react to know which elements require re-rendering.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => (
    <li key={number.toString()}>
          {number}
    </li>
    );
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

Gotchas

1. Keys only make sense in the context of the surrounding array. For example, if you extract a `ListItem` component, you should keep the key on the `<ListItem />` elements in the array rather than on the `<li>` element in the ListItem itself.

2. Keys must be unique among their siblings, but do not need to be globally unique.

Component Lifecycle Callbacks

The two most important lifecycle callbacks are:

componentDidMount

This is your setup function - the appropriate place to make async calls, setup timers, etc.

```
  componentDidMount() {
    // setup
    this.interval = setInterval(() => console.log('tick'), 1000);
  }
```

componentWillUnmount

This is your teardown function. Clean up any leaks that you may have introduced in `componentDidMount`

```
  componentWillUnmount() {
    // teardown
    clearInterval(this.interval);
  }
```

Event handling

In react, you generally don't use `addEventListener`, instead just provide an inline listener in your JSX.

In vanilla JavaScript, we might do:

```
const button = document.getElementById("button");
button.addEventListener("click", handleClick);
```

While in react we would do:

```
class MyComponent {
  function render() {
      return <button id="button" onClick={this.handleClick}>myButton</button>;
  }
}
```

Gotcha - Context binding

There are three ways to ensure that your functions have your component bound as `this`, when they are called from a react template (or anywhere that context is lost).

In the Constructor:

```
class MyComponent {
  function constructor(props) {
    super(props);

    this.handleClick = this.handleClick.bind(this);
  }
}
```

Wherever you're using it:

```
class MyComponent {
  function render() {
      return <button id="button" onClick={this.handleClick.bind(this)}>myButton</button>;
  }
}
```

Using the fat-arrow syntax:

```
class MyComponent {
  handleClick = () => {
    // No need to bind anything, this will be bound correctly.
  }
}
```

Form Handling

Some form elements don't work the same in JSX and in HTML

In HTML a `textarea` element specifies it's value by populating it's `innerText` :

```
<textarea>
  This child content is equivalent to the _value_ attribute of a normal input.
</textarea>
```

In another exception, `select` elements specify their value by setting the `selected` attribute on an `option` element:

```
<select>
  <option value="visa">Visa</option>
  <option value="mc" selected>MasterCard</option>
  <option value="amex">AmericanExpress</option>
</select>
```

React makes this behaviour more consistent, by supporting a `value` attribute on `select` and `textarea` fields, so:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay.' // default value
    };
  }

  render() {
    return (
      <form>
        <label>
          Essay:
          <textarea value={this.state.value} />
        </label>
      </form>
    );
  }
}
```

and

```
class PaymentForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'amex'};
  }

  render() {
    return (
      <select value={this.state.value}>
        <option value="visa">Visa</option>
        <option value="mc">MasterCard</option>
        <option value="amex">AmericanExpress</option>
      </select>
    );
  }
}
```

Read more about Controlled Components to see how to keep this state in sync.

React Router

You will use `react-router-dom` as your routing library. This will allow you to control what components to display using the browser location.

react-router-dom main components

```
import {
  BrowserRouter
  Switch,
  Route,
  NavLink
} from "react-router-dom";
```

`<BrowserRouter></BrowserRouter>` enables the use of the other react-router-dom components and passes routing information to all its descendant components. All other Browser related components must be children of the `BrowserRouter` component.

`<Switch></Switch>` wraps several `<Route .../>` components, rendering just the first matched Route, and no others.

`<Route path="/about" component={About}/>` connects specific URLs to specific components to render.

`<NavLink to="/about" activeClassName="selected">About</NavLink>` works like an anchor tag, updating the URL in the browser. It adds an additional styling attribute when the current URL matches the path.

A simple example

```
const App = props => {
  return (
    <BrowserRouter>
      <div>
        <nav>
          <ul>
            <li><NavLink to="/about">About</NavLink></li>
            <li><NavLink to="/blog">Blog</NavLink></li>
          </ul>
          <Switch>
            <Route path="/about" component={About}/>
            <Route path="/blog" component={Blog}/>
          </Switch>
        </nav>
      </div>
    </BrowserRouter>
  )
}
```

Routes with url parameters

```
<Route path="/users/:userId" component={UserShow}/>
```

When a route is matched, the values passed through the url parameters (`:userId` for example) are added to the component props. They are accessible in props.match.params.

```
const UserShow = props = {
  return (
    <h1>Hello User {props.match.params.userId}</h1>
  );
}
```

React Context

Context Providers and Consumers

React Context's act like `topic` queues, in that any number of consumers can register to receive properties from the single provider. That is to say, any `props` that are passed to the Provider's `value` will be available to all Consumers of the context.

Defining a Provider

This will create a Context wrapper, all properties passed to AppWithContext will be accessable to any consumers of the `AppContext` context.

`AppContext.js`

```
const AppContext = React.createContext({});
```

`AppWithContext.js`

```
import AppContext from './AppContext'

export default class AppWithContext extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      ...props
    }
  }


  render() {
    return (
      <AppContext.Provider value={this.state}>
        <App />
      </AppContext.Provider>
    );
  }
}
```

Defining a Consumer

In order to allow your component to utilize the properties passed through the context, wrap it with `Context.Consumer`.

Create a new Component to which you will pass the `Context` through the props.

Note: `Context.Consumer` accepts a *function* who's first parameter will be the value passed to the `Context.Provider`.

`Component.js`

```
import AppContext from './AppContext'
import ComponentWithContext from './ComponentWithContext'

const Component = (props) => {

  return (
    <AppContext.Consumer>
      { value =>
        <ComponentWithContext {...value}/>
      }
    </AppContext.Consumer>
  );
}
```

`ComponentWithContext.js`

```
const ComponentWithContext = (props) => {
  return (
    <h1>{/* Do something with provided props! */}</h1>
  );
}
```