Pure functions

1. The function always returns the same result if the same arguments are passed in. It does not depend on any state, or data, change during a program's execution. It must only depend on its input arguments.
2. The function does not produce any observable side effects such as network requests, input and output devices, or data mutation.

Pure Function Example In JavaScript

Example of a pure function that calculates the price of a product including tax (UK tax is 20%):

```
function priceAfterTax(productPrice) {
 return (productPrice * 0.20) + productPrice;
}
```

It doesn't depend on any external input, it doesn't mutate any data and it doesn't have any side effects.

If you run this function with the same input , it will **always produce the same result**.

Impure Function In JavaScript

function example in JavaScript:

```
var tax = 20;function calculateTax(productPrice) {
 return (productPrice * (tax/100)) + productPrice;
}
```

If you said it’s because the function depends on an external tax variable you’d be right! A pure function can not depend on outside variables. It fails one of the requirements thus this function is impure.

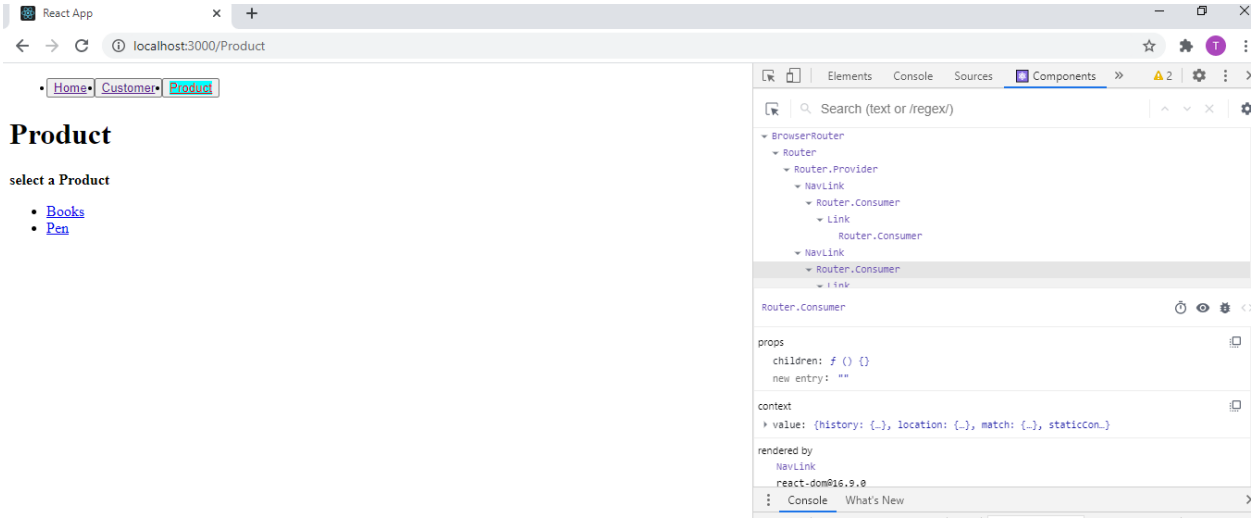
React Developer Tools Chrome Extension

Debugging is one of the most useful skills a developer can possess. It allows you to properly navigate and spot errors in your code quickly and efficiently. In the modern web, this is made possible by leveraging various tools and techniques.

React is one of the fastest-growing front-end frameworks. It makes creating complex and interactive UIs painless. Like other frameworks, it has a debugging toolset called React development tools.

React development tools (React DevTools) is a browser extension available for Chrome, Firefox, and as a standalone app that allows you to inspect the React component hierarchy in the Chrome developer tools. It provides an extra set of React-specific inspection widgets

Installing the new React DevTools
React DevTools is available as an extension for Chrome and Firefox. If you have already installed the extension, it should update automatically.



Lists and keys

React List and Keys

Lists are an important aspect within your app. Every application is bound to make use of lists in some form or the other. You could have a list of tasks like a calendar app, list of pictures like Instagram, list of items to shop in a shopping cart and so on.

Imagine an app with a huge list of videos or pictures and you keep getting thousands more, as you scroll. That could take a toll on the app’s performance.

Because performance is an important aspect, when you are using lists you need to make sure they are designed for optimal performance.

Did you know that in React, when you use lists, each list item needs a unique key?

Rendering a simple List component

```
function ListComponent(props) {
  const listItems = myList.map((item) =>
    <li>{item}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}const myList = ["apple", "orange", "strawberry", "blueberry", "avocado"];
ReactDOM.render(
  <ListComponent myList={myList} />,
  document.getElementById('root')
);
```

The code above shows a *ListComponent* that renders a list of items that are passed to it as *props.* In the *render()* method we have invoked the *ListComponent* and passed to it a list *myList* as the *props.* This code will generate the following output:

- apple
- orange
- strawberry
- avocado

How do you use Keys in Lists?

Keys help React identify which items have changed (added/removed/re-ordered). To give a unique identity to every element inside the array, a key is required.

To better understand this, let’s refactor the code snippet we saw earlier, to now include keys.

```
function ListComponent(props) {
  const listItems = myList.map((item) =>
    <li key={item.id}>
       {item.value}
     </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}const myList = [{id: 'a', value: 'apple'},
                 {id: 'b', value: 'orange'},
                 {id: 'c', value: 'strawberry'},
                 {id: 'd', value: 'blueberry'},
                 {id: 'e', value: 'avocado'}];
ReactDOM.render(
  <ListComponent myList={myList} />,
```

```
  document.getElementById('root')
);
```

In the code snippet above you can notice that we have included a key to each list item. Observe, that we have updated original list to be a value-id pair. Each item in the array has an id associated with it. Hence, this is the id that is assigned as a key for each item. This is the best approach to assign unique keys for items on a list.

Type checking with proptypes

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like Flow or TypeScript to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special propTypes property:

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
```

```
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

Controlled components

A Controlled Component is one that takes its current value through props and notifies changes through callbacks like onChange . A parent component "controls" it by handling the callback and managing its own state and passing the new **values** as props to the controlled component.

UNCONTROLLED VS. CONTROLLED COMPONENT

The next example adds state management with React Hooks to our function component:

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

const App = () => {
  const [value, setValue] = useState('');

  const handleChange = event => setValue(event.target.value);

  return (
    <div>
      <label>
        My still uncontrolled Input:
        <input type="text" onChange={handleChange} />
      </label>

      <p>
        <strong>Output:</strong> {value}
      </p>
    </div>
  );
};
```

```
export default App;
```

FROM UNCONTROLLED TO CONTROLLED COMPONENT

You can change the input from uncontrolled to controlled by controlling its value yourself. For instance, in this case the input field offers a value attribute:

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

const App = () => {
  const [value, setValue] = useState('Hello React');

  const handleChange = event => setValue(event.target.value);

  return (
    <div>
      <label>
        My controlled Input:
        <input type="text" value={value} onChange={handleChange} />
      </label>

      <p>
        <strong>Output:</strong> {value}
      </p>
    </div>
  );
};

export default App;
```

Now the initial state should be seen for the input field and for the output paragraph once you start the application. Also when you type something in the input field, both input field and output paragraph are synchronized by React's state. The input field has become a controlled element and the App component a controlled component. You are in charge what is displayed in your UI. You can see different input elements implemented as controlled

MobX

The philosophy behind MobX is very simple:
Anything that can be derived from the application state, should be derived. Automatically.

In trying to solve the problem of decoupling the state from the individual components as well as sharing the data between them, MobX is similar to Redux. Both rely on the concept of the store for data. However, with MobX you can have multiple mutable stores; the data can be updated directly there. As it is based on the Observable Pattern:

Emitters

Event Emitter is useful library in React apps

Event Emitter in sample how to sharing state in clean way

Assume we have three parts of view: header, sidebar and main. In main view we have input text element and button. And after click we want to pass value from input element to sidebar and header. Additionally header react only on first click. Finally, new value in header and sidebar presents on view.

We can use the event emitter with the life cycle of the component.

refs and context

Refs

Essentially, React provides a special ref attribute that can be applied to any element (JSX or HTML).

Refs provide a way to access DOM nodes or **React** elements created in the render method.

Depending on what type of element you assign it to, the ref provides access to that class instance or DOM element, respectively.

```
// Applying a ref directly to an HTML element
<input
  className="AuthorFilterMenu__filter-input"
  ref={authorFilterInputRef}
  placeholder="Filter by author..."
  value={filterInputValue}
  type="search"
  onInput={event => {
    setFilterInputValue(event.currentTarget.value);
  }}
/>
```

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

const Child = ({ setRef }) => <input type="text" ref={setRef} />;

class Parent extends Component {
    constructor(props) {
        super(props);
        this.setRef = this.setRef.bind(this);
    }
```

```
  componentDidMount() {
    // Calling a function on the Child DOM element
    this.childRef.focus();
  }

  setRef(input) {
    this.childRef = input;
  }

  render() {
    return <Child setRef={this.setRef} />
  }
}
```

Using our Context, we can then pass these refs down to our child components. In the child component, we destructure the appropriate ref off the component's props object and assign it directly to our HTML input:

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between

components without having to explicitly pass a prop through every level of the tree.

React Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

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

function Example() {
  // Declare a new state variable, which we'll call "count"  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Error Boundaries

Introducing Error Boundaries

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a reserve UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {    // Update state so the next render will show the fallback UI.    return { hasError: true };  }
  componentDidCatch(error, errorInfo) {    // You can also log the error to an error reporting service    logErrorToMyService(error, errorInfo);  }
  render() {
    if (this.state.hasError) {      // You can render any custom fallback UI      return <h1>Something went wrong.</h1>;    }
    return this.props.children;
  }
}
```

Then you can use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

Optimizing Performance in React

Optimizing Performance

Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. For many applications, using React will lead to a fast user interface without doing much work to specifically optimize for performance. Nevertheless, there are several ways you can speed up your React application.

Use the Production Build

React apps, make sure you're testing with the minified production build.

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

If you aren't sure whether your build process is set up correctly, you can check it by installing React Developer Tools for Chrome. If you visit a site with React in production mode, the icon will have a dark background:

webpack

Webpack v4+ will minify your code by default in production mode.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

Code Splitting

Code-Splitting

Bundling

Most React apps will have their files “bundled” using tools like Webpack, Rollup or Browserify. Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

We bundle the files in React application using tool such as webpack. Bundling in the end merges the files in the sequence of their imports and creates a single file.

The problem with this approach is that the bundle file gets larger with the increase in files. User may not be sung all the feature components but still bundle is loading them, this could affect the loading of application.

To avoid this, code splitting is used in React.

Example

Example of bundling −

```
// app.js

import { total } from './math.js';

console.log(total(10, 20)); // 42

// math.js

export function total(a, b) {

   return a + b;

}
```

Bundle

```
function total(a, b) {

   return a + b;

}

console.log(total(10, 20)); // 30
```

The code splitting feature uses lazy loading to load only those files which are required . this can improve the performance of app considerably.

Use of dynamic import is simple use case of lazy loading −

Before

```
import { total } from './math';

console.log(total(10, 20));
```

After

```
import("./math").then(math => {

   console.log(math.total(10, 20));
```

```
});
```

Route based code splitting

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import React, { Suspense, lazy } from 'react';

const Customer = lazy(() = > import('./routes/Customer'));

const Admin = lazy(() = > import('./routes/Admin'));

const App = () = > (
  <Router>
    <Suspense fallback = {<div>Loading...</div>}>
      <Switch>
        <Route exact path = "/" component = {Customer}/>
        <Route path = "/admin" component = {Admin}/>
      </Switch>
    </Suspense>
  </Router>
);
```

Suspense

React 16.6 added a <Suspense> component that lets you “wait” for some code to load and declaratively specify a loading state (like a spinner) while we’re waiting:

```
const ProfilePage = React.lazy(() => import('./ProfilePage')); // Lazy-loaded

// Show a spinner while the profile is loading
<Suspense fallback={<Spinner />}>
  <ProfilePage />
</Suspense>
```

Suspense for Data Fetching is a new feature that lets you also use <Suspense> to **declaratively "wait" for anything else, including data.** This page focuses on the data fetching use case, but it can also wait for images, scripts, or other asynchronous work.

Mixin pattern

Reusable Code In React: Inheritance, Composition, Decorators and Mixins

Once you get past the initial complexity it creates by bucking conventions, React is the most developer friendly UI library I have ever used. A big reason for that is the way it uses and encourages a variety of patterns to share codes and concepts across a code base. Let's look at how 4 traditional code re-use patterns are used by React and in the React ecosystem, using a simple counter example to explore why each pattern is used.

Inheritance

Inheritance is a code-reuse technique associated with object oriented programming. If you're like me and learned to program in school using Java, it probably is one of the very first Computer Science concepts you learned. For self-taught JavaScript devs it may be much less familiar. Inheritance is a pattern for extending an existing class or object with additional methods or data, possibly overriding the implementations of other methods in the process. A classic example of inheritance looks like this in JS:

```
class Automobile {
    constructor() {
        this.vehicleName = automobile;
        this.numWheels = null;
    }
    printNumWheels() {
```

```
        console.log(`This ${this.vehicleName} has ${this.numWheels} wheels`);
    }
}

class Car extends Automobile {
    constructor() {
        super(this);
        this.vehicleName = 'car';
        this.numWheels = 4;
    }
}

class Bicycle extends Automobile {
    constructor() {
        super(this);
        this.vehicleName = 'bike';
        this.numWheels = 2;
    }
}

const car = new Car();
const bike = new Bicycle();
car.printNumWheels() // This car has 4 wheels
bike.printNumWheels() // This bike has 2 wheels
```

Inheritance is useful for highly structured objects where you have a lot of shared behavior across similar but slightly different objects and you want to be able to treat all of the objects as interchangeable from an API perspective. This is especially valuable when that interchangeable API is well defined and unlikely to change too much. However it tends to lead to brittle structures when nested across multiple levels, and is a poor fit when dealing with dissimilar objects that need to share a few behaviors.

React Components are objects that we want to treat as interchangeable from an outside API perspective, and they have well defined points where behavior may need to be different between components but don't always need to differ: the lifecycle methods like render, componentDidMount, componentWillUnmount, etc. Thus React relies on a single level of inheritance as the foundation of its class components. Class components inherit from React.Component, and only implement the life cycle methods that they're interested in. But all of them inherit useable setState and forceUpdate functions, and React can verify that by checking the `isReactComponent` property on the object.

Here is a basic counter component. Notice how we don't have to define `setState` on our `Counter` class. Instead it inherits it from React.Component.

```
//counter.js
class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0,
        }
    }
    render() {
        let {count} = this.state;
        let increment = () => this.setState(state => {
            return {
                count: state.count + 1
            };
        });
        return <div className="fancy-layout">
            <div className="fancy-display">
                Current Count: {count}
            </div>
            <button
                className="incrementCounterBtn fancy-btn"
                onClick={increment}
            >Increment</button>
        </div>
    }
}
```

Outside of this single layer of inheritance for defining Class components, inheritance isn't used much in React development. It turns out that better patterns have emerged for sharing application code.

Inheritance Pattern: Used sparingly to share common code across React class components.

Composition

Composition is a code reuse technique where a larger object is created by combining multiple smaller objects. Composition is in some ways the natural pattern of the browser, as HTML uses it. We build a table in html, not by adding a bunch of configuration to a single table object, but by combining a `table` tag with `tr`s, `td`s, and `th`s like this:

```
<table>
    <thead>
        <tr><th>Web Framework</th><th>Language</th></tr>
    </thead>
    <tbody>
        <tr><th>Rails</th><th>Ruby</th></tr>
        <tr><th>Django</th><th>Python</th></tr>
        <tr><th>Flask</th><th>Python</th></tr>
```

```
      <tr><th>Play</th><th>Scala</th></tr>
      <tr><th>Play</th><th>Java</th></tr>
      <tr><th>Express</th><th>JavaScript</th></tr>
   </tbody>
</table>
```

React has fully embraced the HTML style of composing UI components, and it takes thing to the next level by allowing components to be stateful and pass values down into children. For instance, if we wanted to split the counter component above into a separate control and display, perhaps using a standard set of UI components we could do it by extracting FancyLayout, FancyButton and FancyDisplay elements out into their own components and then composing them back into a single counter component. This type of thing is useful for creating and reusing standard UI elements, and might look something like the following:

```
// UI components defined elsewhere
const FancyLayout = ({children}) => <div className="fancy-layout">{children}</div>;
const FancyDisplay = ({children}) => <div className="fancy-display">{children}</div>;
const FancyButton = ({children, className, onClick}) => <button
   className={`fancy-layout ${className}`}
   onClick={onClick}
   >
      {children}
   </button>;


// counter.js
class Counter extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         count: 0,
      }
   }
   render() {
      let {count} = this.state;
       let increment = () => this.setState(state => {
         return {
            count: state.count + 1
         };
      });
      return <FancyLayout>
         <FancyDisplay>
            Current Count: {count}
         </FancyDisplay>
         <FancyButton
            className="incrementCounterBtn"
            onClick={increment}
         >Increment</FancyButton>
      </FancyLayout>;
   }
```

```
}
```

Container and Presentation Components

What's a Presentational Component Pattern?

Presentational Component Patterns can best be described as patterns that are primarily concerned with how things look. The primary function of a presentational component is to display data. They rarely handle state and are best written as stateless functional components. The term "presentational component" does not mean that the component is a type of class in the React library, it just implies a practice which programmers have used over time while creating component-based React user interfaces. Examples of presentational components are lists containing information and data. Check out this code block showing a school list as an example:

```
const SchoolList = props =>
  <ol>
    {props.schools.map(s => (
      <li>{s.name} - {s.grade}</li>
    ))}
  </ol>
```

Although best written as stateless functional components, presentational components can be made to have a degree of interactivity via the addition of callbacks. Check out this example of how an input field can be controlled to have a certain limit of characters:

```
const milesLimit = 3
class MilesRun extends React.Component {
  render() {
    return (
```

```
    <input
      type=number
      className="miles-run"
      onChange={this.props.onChange}
      maxLength={this.props.limit || milesLimit}
    />);
  }
}
```

In the example, `MilesRun` fits the description of a presentational component. It displays data using the `<input>` tag, but then it also provides a class which we can use should we want to style our input and an `onChange` callback as well as a `limit` for the amount of numbers users can enter into the field. By providing all this definition in one place, the application is made easier to work with and debug. Let's have a recap on all we have covered so far:

- Presentational components are primarily concerned with how things look.
- Most times they contain no more than a render method.
- Presentational components do not know how to load or alter the data that they render.
- Presentational components rarely have any internally changeable `state` properties.
- Presentational components best written as stateless functional components.

What's a Container Component Pattern?

If presentational components are concerned with how things look, container components are more concerned with how things work. Container components are components that specify the data a presentational component should render. Instances of container components can be generated using higher order components such as `connect()` from Redux or `createContainer()` from Relay. One very important feature of components is their ability to be reused across different parts of an application. Check out the code block below:

```
class CarList extends React.Component {
  this.state = { cars: [] };
```

```
  componentDidMount() {
    getCars(cars =>
      this.setState({ cars: cars }));
  }
  render() {
    return (
      <ul>
        {this.state.cars.map(e => (
          <li>{e.make}: {e.model}</li>
        ))}
      </ul>
    );
  }
}
```

In the example above, we've made the `CarList` component responsible for fetchig and displaying of data but there's a catch, `CarList` cannot be used unless under the same conditions. Let's implement the container component pattern and solve that problem:

```
class CarListContainer extends React.Component {
  state = { cars: [] };
  componentDidMount() {
    getCars(cars =>
      this.setState({ cars: cars }));
  }
  render() {
    return <CarsList cars={this.state.cars} />;
  }
}
```

We then recreate `CarsList` to take `cars` as a prop:

```
const CarsList = props =>
  <ul>
    {props.cars.map(e => (
      <li>{e.make}: {e.model}</li>
    ))}
  </ul>
```

By setting apart our data fetching and rendering operations, we have made our `CarsList` component super reusable. Not only that, we get go understand our application's user interface better and make our components easier to understand.

Higher Order Components

A higher-order component is a function that takes a component and returns a new component. A higher-order component (HOC) is the advanced technique in React.js for reusing a component logic. Higher-Order Components are not part of the React API. They are the pattern that emerges from React's compositional nature. The component transforms props into UI, and a higher-order component converts a component into another component. The examples of HOCs are Redux's connect and Relay's createContainer.

Syntax Overview of HOC

Step 1: Create one React.js project.

```
npm install -g create-react-app
create-react-app my-app

cd my-app
npm start
```

Step 2: Create one new file inside src folder called HOC.js.

```
// HOC.js

import React, {Component} from 'react';

export default function Hoc(HocComponent){
    return class extends Component{
        render(){
            return (
                <div>
                    <HocComponent></HocComponent>
                </div>

            );
        }
    }
}
```

Now, include this function into the App.js file.

```
// App.js

import React, { Component } from 'react';
import Hoc from './HOC';

class App extends Component {

  render() {
    return (
      <div>
        Higher-Order Component Tutorial
      </div>
    )
  }
}
App = Hoc(App);
export default App;
```

Explanation

First, we have made one function that is Hoc inside HOC.js file.

That function accepts one argument as a component. In our case that component is App.

App = Hoc(App);

Composition vs Inheritance and comparison of paradigms

Composition vs Inheritance

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

In this section, we will consider a few problems where developers new to React often reach for inheritance, and show how we can solve them with composition.

Containment

Some components don’t know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic “boxes”.
We recommend that such components use the special children prop to pass children elements directly into their output:

```
function FancyBorder(props) {
```

```
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}    </div>
  );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">        Welcome      </h1>      <p className="Dialog-
message">        Thank you for visiting our spacecraft!      </p>    </FancyBorder>
  );
}
```

Try it on CodePen

Anything inside the <FancyBorder> JSX tag gets passed into the FancyBorder component as a children prop.

Since FancyBorder renders {props.children} inside a , the passed elements appear in the final output.

While this is less common, sometimes you might need multiple “holes” in a component. In such cases you may come up with your own convention instead of using children:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}      </div>
      <div className="SplitPane-right">
        {props.right}      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />      }
      right={
        <Chat />      } />
```

```
  );
}
```

Try it on CodePen

React elements like <Contacts /> and <Chat /> are just objects, so you can pass them as props like any other data. This approach may remind you of "slots" in other libraries but there are no limitations on what you can pass as props in React.

Composition works equally well for components defined as classes:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
              message="How should we refer to you?">
        <input value={this.state.login}              onChange={this.handleChange} />        <button
onClick={this.handleSignUp}>          Sign Me Up!        </button>      </Dialog>
    );
  }

  handleChange(e) {
```

```
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

Anti-patterns

1Why Is This An "Anti-Pattern" in React????2Throw Out Your React State-Management Tools...5 more parts...6Hacking React Hooks: Shared Global State7Synchronous State With React Hooks

A few days ago, I wrote a post about a workaround/hack that I've been using in React to pass around components' state variables and functions. I knew that my approach was by-no-means perfect, so I openly solicited feedback from the community - and they delivered.

What I'm going to discuss/illustrate here is (IMHO) a far better approach to shared state in React. This approach does *not* use any third-party or bolt-on state-management libraries. It uses React's core constructs to address the "challenge" of prop drilling. Specifically, I'm talking about React's Context API.

Integrating 3rd party components

Third Party Integration

Mixing 3rd party integrations/libraries with React

In this example we'll see how to mix React and jQuery's UI plugin. We pick tag-it jQuery plugin for the example. It transforms an unordered list to input field for managing tags:

```
<ul>
  <li>JavaScript</li>
  <li>CSS</li>
</ul>
```

To make it work we have to include jQuery, jQuery UI and the tag-it plugin code. It works like that:

```
$('<dom element selector>').tagit();
```

We select a DOM element and call tagit().

The very first thing that we have to do is to force a single-render of the Tags component. That's because when React adds the elements in the actual DOM we want to pass the control of them to jQuery. If we skip this both React and jQuery will work on same DOM elements without knowing for each other. To achieve a single-render we have to use the lifecycle method `shouldComponentUpdate`
Let's say that we want to programmatically add a new tag to the already running tag-it field. Such action will be triggered by the React component and needs to use the jQuery API. We have to find a way to communicate data to Tags component but still keep the single-render approach. To illustrate the whole process we will add an input field to the App class and a button which if clicked will pass a string to Tags component.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this._addNewTag = this._addNewTag.bind(this);
    this.state = {
      tags: ['JavaScript', 'CSS'],
      newTag: null
    };
  }
```

```
  _addNewTag() {
    this.setState({newTag: this.refs.field.value});
  }

  render() {
    return (
      <div>
        <p>Add new tag:</p>
        <div>
          <input type='text' ref='field'/>
          <button onClick={ this._addNewTag }>Add</button>
        </div>
        <Tags tags={ this.state.tags } newTag={ this.state.newTag }/>
      </div>
    );
  }
}
```

We use the internal state as a data storage for the value of the newly added field. Every time when we click the button we update the state and trigger re-rendering of Tags component. However, because of shouldComponentUpdate we update nothing. The only one change is that we get a value of the newTag prop which may be captured via another lifecycle method - componentWillReceiveProps

```
class Tags extends React.Component {
  componentDidMount() {
    this.list = $(this.refs.list);
    this.list.tagit();
  }

  shouldComponentUpdate() {
    return false;
  }

  componentWillReceiveProps(newProps) {
    this.list.tagit('createTag', newProps.newTag);
  }

  render() {
    return (
      <ul ref='list'>
        { this.props.tags.map((tag, i) => <li key={ i }>{ tag } </li>) }
      </ul>
    );
  }
}
```

React-select case study

Installation and usage

The easiest way to use react-select is to install it from npm and build it into your app with Webpack.

```
yarn add react-select
```

Then use it in your app:

```
import React from 'react';

import Select from 'react-select';



const options = [

  { value: 'chocolate', label: 'Chocolate' },

  { value: 'strawberry', label: 'Strawberry' },

  { value: 'vanilla', label: 'Vanilla' },

];



class App extends React.Component {

  state = {
```

```
    selectedOption: null,
  };
  handleChange = selectedOption => {
    this.setState({ selectedOption });
    console.log(`Option selected:`, selectedOption);
  };
  render() {
    const { selectedOption } = this.state;

    return (
      <Select
        value={selectedOption}
        onChange={this.handleChange}
        options={options}
      />
    );
  }
}
```

React data grid

ag-Grid is the industry standard for React Enterprise Applications. Developers using ag-Grid are building applications that would not be possible if ag-Grid did not exist.

Quick Look Code Example

- index.js
- index.html

```
import React, { useState } from 'react';
import { render } from 'react-dom';
import { AgGridColumn, AgGridReact } from 'ag-grid-react';

import 'ag-grid-enterprise';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

const App = () => {
    const [gridApi, setGridApi] = useState(null);
    const [gridColumnApi, setGridColumnApi] = useState(null);

    const [rowData, setRowData] = useState([
        {make: "Toyota", model: "Celica", price: 35000},
        {make: "Ford", model: "Mondeo", price: 32000},
        {make: "Porsche", model: "Boxter", price: 72000}
    ]);

    function onGridReady(params) {
        setGridApi(params.api);
```

```
        setGridColumnApi(params.columnApi);
    }

    return (
        <div className="ag-theme-alpine" style={ { height: 400, width: 600 } }>
            <AgGridReact
                onGridReady={onGridReady}
                rowData={rowData}>
                <AgGridColumn field="make"></AgGridColumn>
                <AgGridColumn field="model"></AgGridColumn>
                <AgGridColumn field="price"></AgGridColumn>
            </AgGridReact>
        </div>
    );
};

render(<App />, document.getElementById('root'));
```

State management

Need for state management

Redux (without React)

React with Redux

Mobx

Comparison of different state management solutions

React Testing

React Test Utilities

ReactTestUtils makes it easy to test React components in the testing framework of your choice. At Facebook we use Jest for painless JavaScript testing Importing

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // ES5 with npm
```

React Testing Library

React Testing Library builds on top of DOM Testing Library by adding APIs for working with React components.

Projects created with Create React App have out of the box support for **React Testing Library**. If that is not the case, you can add it via npm like so: Copy

```
npm install --save-dev @testing-library/react
```

Shallow rendering

Shallow Renderer

Importing

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 with npm
```

Overview

When writing unit tests for React, shallow rendering can be helpful. Shallow rendering lets you render a component “one level deep” and assert facts about what its render method returns, without worrying about the behavior

of child components, which are not instantiated or rendered. This does not require a DOM.

For example, if you have the following component:

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

Then you can assert:

```
import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing currently has some limitations, namely not supporting refs.

Jest

Jest is a delightful JavaScript Testing Framework with a focus on simplicity.

It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!

Enzyme

Enzyme is a JavaScript Testing utility for React that makes it easier to test your React Components' output. You can also manipulate, traverse, and in some ways simulate runtime given the output.

Enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal.

Upgrading from Enzyme 2.x or React < 16

Are you here to check whether or not Enzyme is compatible with React 16? Are you currently using Enzyme 2.x? Great! Check out our migration guide for help moving on to Enzyme v3 where React 16 is supported.

Installation

To get started with enzyme, you can simply install it via npm. You will need to install enzyme along with an Adapter corresponding to the version of react (or other UI Component library) you are using. For instance, if you are using enzyme with React 16, you can run:

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

Mocha

Using enzyme with Mocha

enzyme was originally designed to work with Mocha, so getting it up and running with Mocha should be no problem at all. Simply install it and start using it:

```
npm i --save-dev enzyme
import React from 'react';
import { expect } from 'chai';
import { mount } from 'enzyme';
import { spy } from 'sinon';
import Foo from './src/Foo';
```

```
spy(Foo.prototype, 'componentDidMount');

describe('<Foo />', () => {
  it('calls componentDidMount', () => {
    const wrapper = mount(<Foo />);
    expect(Foo.prototype.componentDidMount).to.have.property('callCount', 1);
  });
});
```