

Presentation by Uplatz

Contact Us: <https://training.uplatz.com/>

Email: info@uplatz.com

Phone: +44 7836 212635

Table Of Contents:

- React and ES6 Workflow with Webpack
- Passing Data Between React Components — Parent, Children, Siblings
- Pure Components
- User interaction(MouseOver)

React and ES6 Workflow with Webpack:

Why Webpack?

- Like JSPM, Webpack is the solution for module management of your frontend application.
- With Webpack, you are able to fully control all the assets of application in a convenient way.

Why is Webpack so popular? There is number of reasons:

- Webpack uses npm as a source of external modules. If you want to add React to your project, just run `npm install react`.
- This is an added advantage because you'll already know how to drop your favourite library into a project.
- You could load almost everything, not only JavaScript.

- **Webpack uses concept named loaders for that purpose.**

Just take a look at this list.

- Webpack comes with a powerful ecosystem of developer tools.
- Things like Hot Module Replacement (more on this later in this article) will change your development workflow dramatically.

Getting started

- Let's start adapting our application from previous parts of series.
- First, we are going to install initial dev.dependencies:

npm install --save-dev webpack@1.x

npm install --save-dev babel-core

npm install --save-dev babel-preset-es2015 babel-preset-react babel-preset-stage-0

In the above list, webpack package should be self-explanatory.

- Babel is used for translating of JavaScript version 2015 (ES2015 or ES6) to JavaScript version 5 (should be already familiar to you if you've read other articles in "React and ES6" series).
- Since babel version 6 you have to install separate package for each additional language feature. These packages are called presets.
- We install es2015 preset, react preset and stage-0 preset (for things like class properties).

For more information about babel 6, you could refer to my another article.

- Next, install non-dev dependencies (react and react-dom packages):

npm install --save react react-dom

- Now the most important step in any project based on Webpack.

Create the file named webpack.config.dev.js in the root folder of your project.

- This file will be used by Webpack to pack all your JavaScript (in the majority of project it won't be only JavaScript) files in one bundle (or in multiple bundles if you would like to) that will be loaded into user's browser.

The contents of our **webpack.config.dev.js** is as follows:

```
var path = require('path');
var webpack = require('webpack');
var config = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    './app.js'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/dist/'
  },
  plugins: [
    new webpack.NoErrorsPlugin()
  ]
};
```

module.exports = config;

Main highlights in the above code:

- **Line 5.** We have a choice between various strategies of enhancing debugging of the application
- You could read more about strategy cheap-module-eval-source-map by clicking this link.
- **Lines 6-8.** Here we define that app.js will be the main JavaScript file of an application.
- **Lines 9-13.** This configuration specifies that Webpack will pack all modules under one file bundle.js and place it in dist/ directory.

Webpack loaders

- It's possible to load almost everything into your code with Webpack (take a look at this list).

- The concept that Webpack uses for that, is called Webpack loaders.
- You could specify the file extensions to be associated with particular loader.
- In our example, we'll use babel-loader to produce ES5 JavaScript out of ES2015 / ES6 code.
- First, we need to install corresponding npm package:

npm install --save-dev babel-loader

Then, adjust the configuration file `webpack.config.dev.js` by adding new loaders key to the exported object:

var config = {

... add the below code as object key ...

```
module: {  
  loaders: [  
    {  
      test: /\.js$/,  
      loaders: ['babel'],  
      exclude: /node_modules/  
    }  
  ]  
}  
};
```

```
module.exports = config;
```

- Important to note here is that we don't allow Webpack to parse files inside node_modules directory by using exclude key.
- This is just slow and useless.
- The finishing move will be adding file called .babelrc to the root of the project:

```
{  
  "presets": ["react", "es2015", "stage-0"]  
}
```

- This file configures babel to use react, es2015 and stage-0 presets added previously.
- Now, whenever Webpack encounters ie. import CartItem from './cartItem.js', it will load this file and transpile ES6 code inside it to ES5.

Adding Webpack Development Server


- In order to run an application, we'll need to run our files on some server.
- Luckily, Webpack eco-system already provides all the needed pieces to get you started.
- You could use either Webpack Development Server or Webpack Development Middleware for Express.js.
- Advantages include faster serving due to handling files in memory instead of file system and watch mode for live reloading.

Let's install corresponding npm modules:

`npm install --save-dev webpack-dev-middleware`
`express`

Next, add file named `server.js` to the root:

```
var path = require('path');  
var express = require('express');  
var webpack = require('webpack');  
var config = require('./webpack.config.dev');  
var port = 3000;  
var app = express();  
var compiler = webpack(config);  
app.use(require('webpack-dev-  
middleware')(compiler, {  
  noInfo: true,  
  publicPath: config.output.publicPath  
}));  
app.use(require('webpack-hot-  
middleware')(compiler));
```

```
app.get('*', function (req, res) {  
  res.sendFile(path.join(__dirname, 'index.html'));  
});  
app.listen(port, function onAppListening(err) {  
  if (err) {  
    console.error(err);  
  } else {  
    console.info('==>  Webpack development  
server listening on port %s', port);  
  }  
});
```

Adding Hot Module Reloading

- Webpack Dev Middleware already includes live reloading feature.

- Whenever you change the code of the file, it reloads the page.
- Though, Webpack has some more powerful feature called Hot Module Reloading.
- It reloads not the entire page, but only the part that was changed.
- This is handy when you are trying to fix the specific bug revealed after 10 subsequent user actions.
- You don't have to manually re-do these 10 user actions when the page is reloaded.
- Just because only the part of the page (
- corresponding to your changes) was reloaded.

In order to activate Hot Module Reloading, first install necessary npm packages:

npm install --save-dev webpack-hot-middleware

Then adjust entry and plugins fields in your **webpack.config.dev.js**:

```
var config = {  
  entry: [  
    './app.js',  
    'webpack-hot-middleware/client'  
  ],  
  plugins: [  
    new webpack.HotModuleReplacementPlugin(),  
    new webpack.NoErrorsPlugin()  
  ],  
};  
module.exports = config;
```


To further use module reloading specifically for React application there are various ways.

- One of the easiest is to install special babel-preset-react-hmre babel preset:

npm install --save-dev babel-preset-react-hmre

1. And adjust .babelrc file:

```
{  
  "presets": ["react", "es2015", "stage-0"],  
  "env": {  
    "development": {  
      "presets": ["react-hmre"]  
    }  
  }  
}
```

➤ **Adjusting package.json**

- Now it's time to combine all the previous parts in one working piece of software.
- We'll do it by adding some scripts into package.json scripts section:

```
{  
  "name": "awesome-application",  
  "version": "1.0.0",  
  ...  
  "scripts": {  
    "start": "node server.js"  
  },  
}
```

Running the application

- Run **npm start** (this script just runs server.js which in sequence runs Webpack we've setup previously), open up **http://localhost:3000** in you browser.
- Grab the file webpack.config.prod.js from the repository and drop it into the project.
- It's similar to Webpack configuration for development mode, but with following differences:
Hot reloading mode is disabled, just because it's not needed in production.
- JavaScript bundle is compressed by UglifyJs using the corresponding webpack.optimize.UglifyJsPlugin.
- Environment variable NODE_ENV is set to production.

- This is required to suppress warnings coming from React in development mode (and could be potentially used for other purposes).
- Next, update scripts section of your package.json file:

```
{ ...  
  "scripts": {  
    "start": "node server.js"  
    "clean": "rimraf dist",  
    "build:webpack": "NODE_ENV=production  
webpack --progress --colors --config  
webpack.config.prod.js",  
    "build": "npm run clean && npm run  
build:webpack",  
  },
```

- By now, if you run `npm run build`, compressed file `bundle.js` file will be created under `dist/` directory.

Passing Data Between React Components — Parent, Children, Siblings

- React is a JavaScript library created by Facebook.
- Data handling in React could be a bit tricky, but not as complicated as it might seem.
- I have currently compiled three methods of Data Handling in React :-

From Parent to Child using Props

From Child to Parent using Callbacks

Between Siblings :

(i) Combine above two methods

(ii) Using Redux

(iii) Using React's Context API

From Parent to Child Using Props

- Let us consider our directory structure to be such that the Parent Component renders child components in the Application.

```
App
├── Parent
│   ├── Child1
│   └── Child2
```

- This is the easiest direction of data flow in React and the most basic one.

```
class Parent extends React.Component {  
  state = { data : "Hello World" }  
  render() {  
  
    return (  
      <div>  
        <Child1 />           //no data to send  
        <Child2 dataFromParent = {this.state.data} />  
      </div>  
    );  
  }  
}
```

- //It is no compulsion to use the data to send as a state, simple vars or const variables could also be used to send data from Parent to Child.
- Simply, use `this.props.dataFromParent` (just a variable used for sending props) to access the data sent from Parent to Child.

```
class Child2 extends React.Component {  
  render() {  
    return (  
      <div>  
        The data from parent  
is:{this.props.dataFromParent}  
      </div>  
    );  
  }  
}
```


From Child to Parent Using Callbacks

- **Step 1:** Define a callback function that takes in a parameter which we consider having accessed from the child in the Parent.js
- **Step 2:** Also, send the defined callback function as a props to the Child1.js

```
class Parent extends React.Component {  
  state = { message: "" }  
  callbackFunction = (childData) => {  
    this.setState({message: childData})  
  },  
  render() {  
    return (  
      <div>
```

```
<Child1 parentCallback = {this.callbackFunction}/>
    <p> {this.state.message} </p>
</div>
);
}}
```

➤ **Step 3:** In Child1.js send the data using
this.props.callback(dataToParent)

```
class Child1 extends React.Component{
  sendData = () => {
    this.props.parentCallback("Hey Popsie, How's it
going?");
  },
  render() {
```

//you can call function sendData whenever you'd like to send data from child component to Parent component.

```
}
```

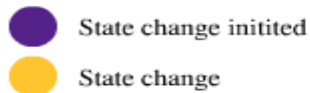
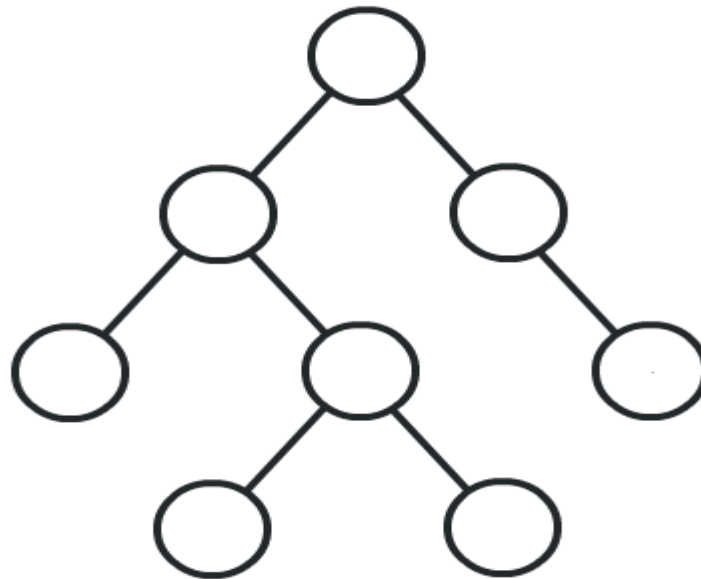
```
};
```

Between Siblings

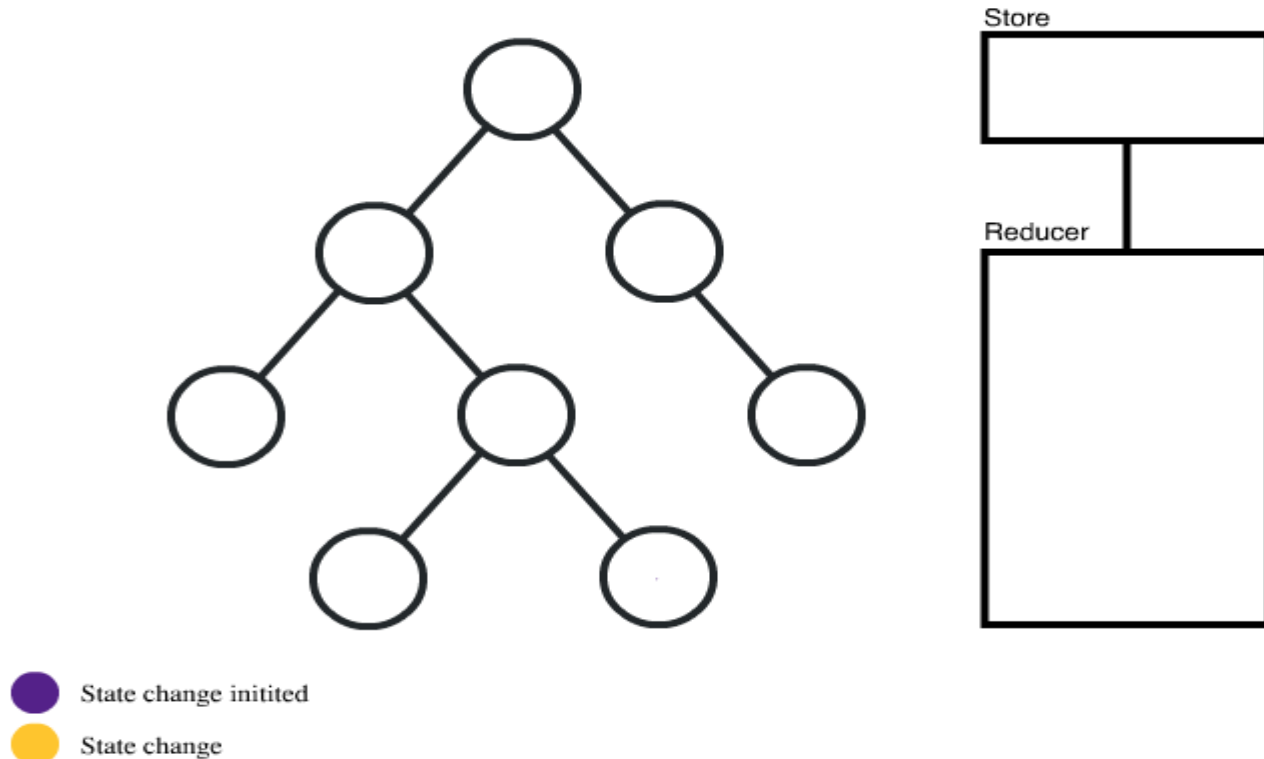
- When I was a beginner, it took me a hard time deciding which method to choose to share data between siblings, there are three methods known to me yet to share data between siblings and all of them have their own perks and cons.

Method 1: Combine the above two methods of sharing data.

- This method however, will not work for complicated directory structures as one will have to write large bits of code for sending data between components at far levels from each other.
- The data, then will have to be pushed and pulled through each middle level.

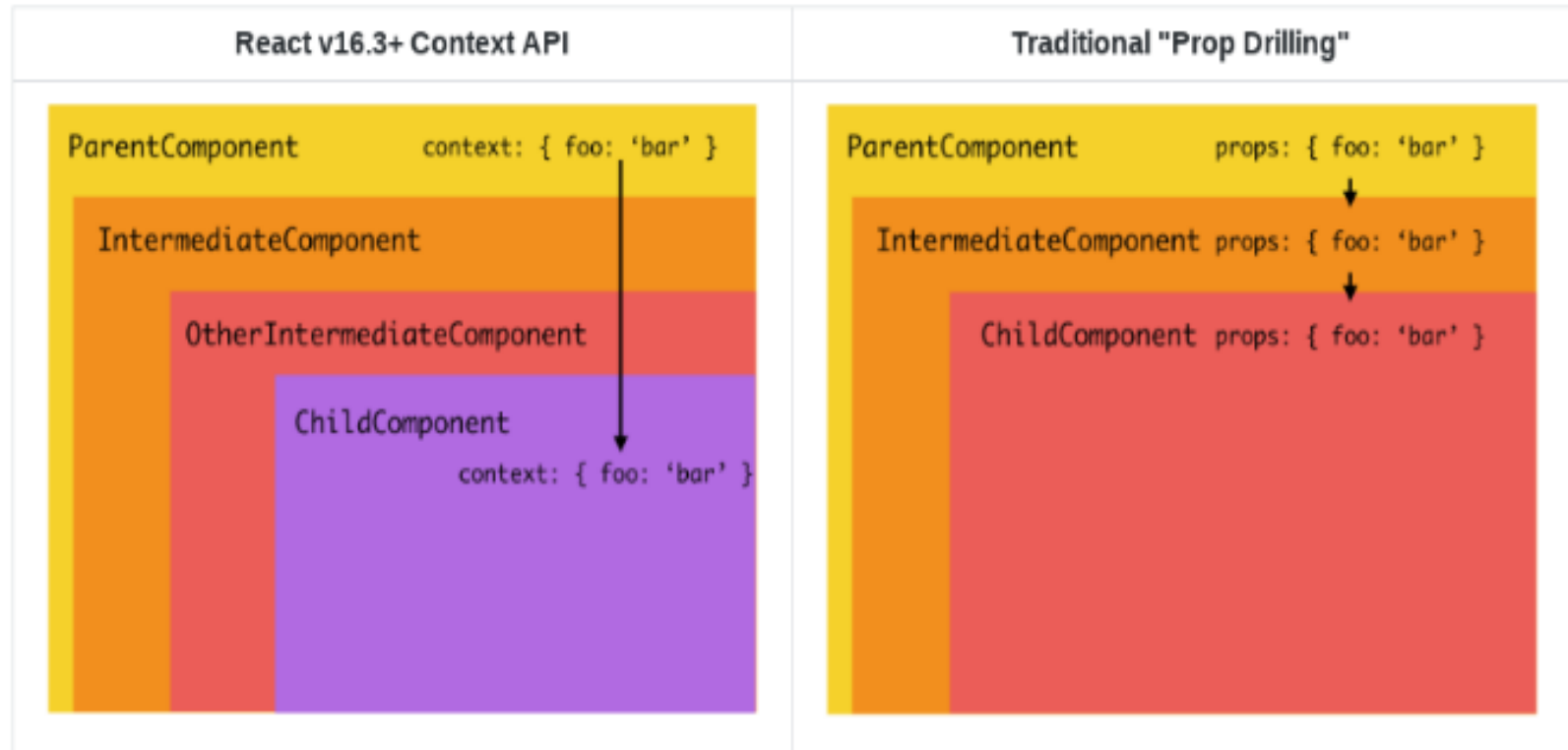


Method 2: Use a global store maintaining the states of all child components which are needed to interact and consume the data required from the store — Redux



Method 3: Use React's Context API

- There are tons of articles and blogs already regarding why React upgraded to Context API and which one is better in what terms I have used this method and already have a slight inclination towards using this one over Redux.
- The major advantage of Context API is that it saves the developer from Prop-Drilling.
- (Prop-drilling refers to the technique of passing down variables to sub components.
- The main idea is functional programming where you pass the parameters to the next function and so on)



- Consider the directory structure and we need to pass data between Child1 and Child2.
- [Child1 has to send message — “SSup brother??” to Child2]

- We implement this in the following method using Context API:

App



Step1: Create a Provider Component for the two children.

- This Provider maintains the state (data to be used by both components and some callback used to manipulate the states) and returns a `contextObject.Provider` JSX component
- **Step 2:** Pass the state and the callback function as props to all children inside the Provider Component.


```
export const MContext = React.createContext();  
//exporting context object  
class MyProvider extends Component {  
  state = {message: ""}  
  render() {  
    return (  
      <MContext.Provider value={  
        { state: this.state,  
          setMessage: (value) => this.setState({  
            message: value })}}>  
        {this.props.children} //this indicates that the  
global store is accessible to all the child tags with  
MyProvider as Parent  
      </MContext.Provider>  
    )  
  }  
}
```

- The provider is the boss for its children (the global store of all the states and callback functions to manipulate those states).
- Who-ever needs anything has to contact the provider first to access the objects.
- (a) To set or manipulate the message by Child1, it has to access Provider and set the states of the Provider.
- (b) To view/access the data by Child2, it has to access Provider to get the states.
- **Step 3:** Use MyProvider component as a Parent to the two children — Child1, Child2.

```
class App extends React.Component {  
  render() {  
    return (  
      <div>
```

<MyProvider>

<div className="App">

<Child1/>

<Child2/>

</div>

</MyProvider>

</div>

);} }

- **Step 4:** Implement the desired result in the same manner, but this time, using `ContextObject.Consumer` as explained below:
- Both the children — `Child1` and `Child2` are the consumers of the Provider.
- Henceforth, they access the Provider within Consumer Tags.

```
import MContext
class Child1 extends React.Component {
  render() {
    return (
      <div>
        <Mcontext.Consumer>
          {(context) => (
            <button onClick={()=>{context.setMessage("New
Arrival")}}>Send</button>
          )}
        </Mcontext.Consumer>
      </div>
    )}
  }
}
```

- How does Child2 receives the data now?
- Simply, accessing the Provider withing Consumer tags.

```
import MContext  
class Child2 extends React.Component {  
  render() {  
    return (  
      <div>  
        <Mcontext.Consumer>  
          {(context) => (  
            <p>{context.state.message}</p>))  
        </Mcontext.Consumer>  
      </div>  
    )}  
}
```

- I hope this gives clear implementation details for Data Passing between various components in React.
- How to pass data from child component to its parent in ReactJS?
- Now, update default App.js with below code snippet

src \App.js

```
import React, { Component } from 'react';  
import Area from './components/Area';  
import Parameter from './components/Parameter';  
class App extends Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      data: 0,
```

```
len: 40,  
  bre: 50  
}  
this.updateState = this.updateState.bind(this);  
} updateState(calculation) {  
  if(calculation == "Area")  
  {  
    this.setState({data: this.state.len * this.state.bre})  
  }  
  else  
  {  
    this.setState({data: 2 * (this.state.len +  
this.state.bre)})  
  }  
}
```

```
render() {  
  return (  
    <div>  
      Calculation: {this.state.data}  
      <Area update={this.updateState} />  
      <Parameter update={this.updateState} />  
    </div>  
  )  
}
```

export default App;

- Create new Area component and type out the below code into Area.js

src\components\Area.js


```
import React, { Component } from 'react';
class Area extends Component {
  render() {
    return (
      <div>
        Area:
        <button onClick={() => this.props.update('Area')}>
          Area
        </button>
      </div>
    );
  }
}
export default Area;
```

- Create new Parameter component and type out the below code into Parameter.js

src \ components \ Parameter.js

import React, { Component } from 'react';

class Parameter extends Component {

render() {

return (

<div>

Parameter:

<button onClick={() =>

this.props.update('Parameter')}}>

Parameter

</button>

</div>

);} }

export default Parameter;

- The Area and Parameter components are passing the value from child component to parent component.

Pure Components

- React offers several different methods for creating components.
- Today we'll talk about the final method of creating components, the function stateless pure component.
- We've looked at a few different ways to build react components.
- One method we left out up through this point is the stateless component/functional method of building React components.

- As we've seen up through this point, we've only worked through building components using the `React.Component` and `React.createClass()` methods.
- For more performance and simplicity, React also allows us to create pure, stateless components using a normal JavaScript function.
- A Pure component can replace a component that only has a render function.
- Instead of making a full-blown component just to render some content to the screen, we can create a pure one instead.
- Pure components are the simplest, fastest components we can write

- They are easy to write, simple to reason about, and the quickest component we can write.
- Before we dive into why these are better, let's write one, or heck a couple!

// The simplest one

```
const HelloWorld = () => (<div>Hello world</div>);
```

// A Notification component

```
const Notification = (props) => {  
  const {level, message} = props;  
  const classNames = ['alert', 'alert-' + level]  
  return (  
    <div className={classNames}>  
      {message}  
    </div>  
  ) };
```

// In ES5

```
var ListItem = function(props) {  
  var handleClick = function(event) {  
    props.onClick(event);  
  };  
  return (  
    <div className="list">  
      <a  
        href="#"  
        onClick={handleClick}>  
        {props.children}  
      </a>  
    </div>  
  )  
}
```

- The idea is that if React knows the props that are sent into a component, it can be deterministic in knowing if it has to rerender or not.
- The same props in equal the same output virtual DOM.
- In React, functional components are called with an argument of props (similar to the `React.Component` constructor class), which are the props it's called with as well as with the current context of the component tree.
- For instance, let's say we want to rewrite our original Timer component using functional components as we want to give our users a dynamic way to set their own clock styles (24 hour clock vs. 12, different separators, maybe they don't want to display the seconds, etc).

- We can break up our clock into multiple components where we can use each block of time as an individual component.
- We might break them up like so:

```
const Hour = (props) => {  
  let {hours} = props;  
  if (hours === 0) { hours = 12; }  
  if (props.twelveHours) { hours -= 12; }  
  return (<span>{hours}</span>)  
}  
  
const Minute = ({minutes}) => (<span>{minutes<10 &&  
'0'}{minutes}</span>)  
  
const Second = ({seconds}) => (<span>{seconds<10  
&& '0'}{seconds}</span>)
```



```
const Separator = ({separator}) => (<span>{separator  
| | ':'}</span>)
```

```
const Ampm = ({hours}) => (<span>{hours >= 12 ? 'pm' :  
'am'}</span>)
```

- With these, we can place individual components as through they are full-blown React components (they are):

```
<div>Minute: <Minute minutes={12} /></div>
```

```
<div>Second: <Second seconds={51} /></div>
```

- We can refactor our clock component to accept a format string and break up this string selecting only the components our user is interested in showing.
- There are multiple ways we can handle this, like forcing the logic into the Clock component or we can create another stateless component that accepts a format string.

Let's do that (easier to test):

```
const Formatter = (props) => {  
  let children = props.format.split('').map((e, idx) => {  
    if (e === 'h') {  
      return <Hour key={idx} {...props} />  
    } else if (e === 'm') {  
      return <Minute key={idx} {...props} />  
    } else if (e === 's') {  
      return <Second key={idx} {...props} />  
    } else if (e === 'p') {  
      return <Ampm key={idx} {...props} />  
    } else if (e === ' ') {  
      return <span key={idx}> </span>;  
    } else {
```

```
return <Separator key={idx} {...props} />
  }
});
return <span>{children}</span>;
}
```

- This is a little ugly with the key and {...props} thingie in there.
- React gives us some helpers for mapping over children and taking care of handling the unique key for each child through the React.Children object.
- The render() function of our Clock component can be greatly simplified thanks to the Formatter component into this:

```
class Clock extends React.Component {  
  state = { currentTime: new Date() }  
  componentDidMount() {  
    this.setState({  
      currentTime: new Date()  
    }, this.updateTime);  
  }  
  componentWillUnmount() {  
    if (this.timerId) {  
      clearTimeout(this.timerId)  
    }  
  }  
  updateTime = e => {  
    this.timerId = setTimeout(() => {
```

```
this.setState({
  currentTime: new Date()
}, this.updateTime);
})
}

render() {
  const { currentTime } = this.state
  const hour = currentTime.getHours();
  const minute = currentTime.getMinutes();
  const second = currentTime.getSeconds();
  return (
    <div className='clock'>
      <Formatter
        {...this.props}
```

```
state={this.state}
  hours={hour}
  minutes={minute}
  seconds={second}
/>
</div>
)
}
}
```

We can now render the clock in a custom format:

➤ `ReactDOM.render(<Clock format="h:m:s p" />, document.querySelector("#app"));`

- Not only is our Clock component much simpler, but it's so much easier to test. It also will help us transition to using a data state tree, like Flux/Redux frameworks, but more on those later.

Output:

20:17:02 pm

Advantages to using functional components in React are:

- We can do away with the heavy lifting of components, no constructor, state, life-cycle madness, etc.
- There is no this keyword (i.e. no need to bind)
- Presentational components (also called dumb components) emphasize UI over business logic (i.e. no state manipulation in the component)

- Encourages building smaller, self-contained components
- Highlights badly written code (for better refactoring)
- They are easy to reuse
- You might say why not use a functional component? Well, some of the disadvantage of using a functional component are some of the advantages:
- No life-cycle callback hooks
- Limited functionality
- There is no `this` keyword

Overall, it's a really good idea to try to prefer using functional components over their heavier `React.Component` cousins

User interaction

- The browser is an event-driven application.
- Everything that a user does in the browser fires an event, from clicking buttons to even just moving the mouse.
- In plain JavaScript, we can listen for these events and attach a JavaScript function to interact with them.
- For instance, we can attach a function to the mousemove browser event with the JS:

```
const ele = document.getElementById('mousemove');  
ele.innerHTML = 'Move your mouse over this text';  
ele.addEventListener('mousemove', function(evt) {  
  const { screenX, screenY } = evt;
```

```
ele.innerHTML = '<div>Mouse is at: X: ' +  
    screenX + ', Y: ' + screenY +  
    '</div>';  
})
```

This results in the following functionality:

Move your mouse over this text

- In React, however we don't have to interact with the browser's event loop in raw JavaScript as React provides a way for us to handle events using props.
- For instance, to listen for the mousemove event from the (rather unimpressive) demo above in React, we'll set the prop onMouseMove (notice the camelcasing of the event name).

```
class MouseMover extends React.Component {  
  state = {  
    x: 0,  
    y: 0  
  };  
  handleMouseMove = e => {  
    this.setState({  
      x: e.clientX,  
      y: e.clientY  
    });  
  };  
  render() {  
    return (  

```

```
<div onMouseMove={this.handleMouseMove}>
  {this.state.x || this.state.y
    ? "The mouse is at x: " + this.state.x + ", y: " +
this.state.y
    : "Move the mouse over this box"}
</div>
);
}
}
```

Summary:



React and ES6 Workflow with Webpack



Passing Data Between React Components — Parent, Children, Siblings



Pure Components



User interaction(MouseOver)

Thank You.....

If you have any queries please write to info@uplatz.com".