Chapter 5 – REACTJS

Introducing React, Component React JS, Styling in React, Creating Complex Components, Transferring Properties, Dealing With State in React, Going from Data to UI in React

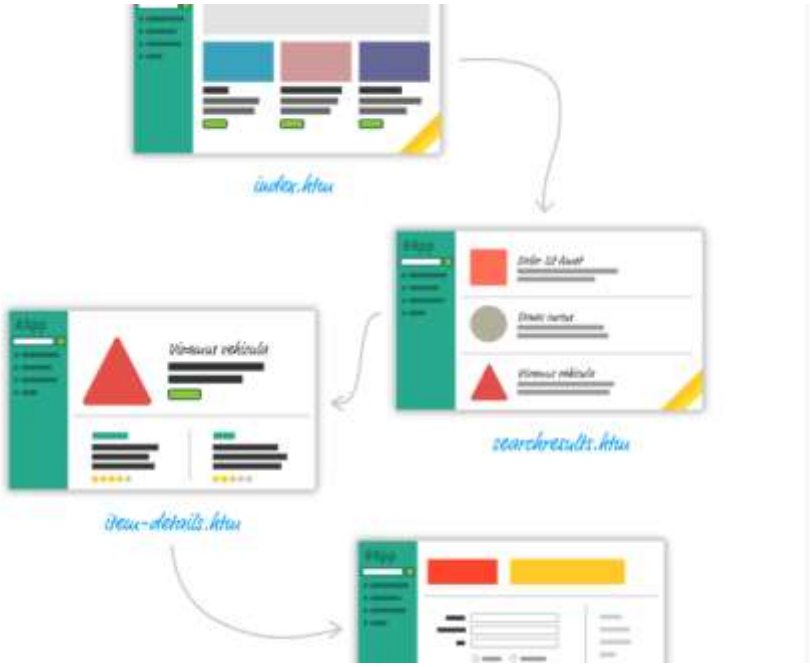
Introducing React

Ignoring for a moment that web apps today both ***look*** and ***feel*** nicer than what they did back in the day, there is something even more fundamental that has changed. The way we architect and build web apps is very different now.

This app is a simple catalog browser for something. Like any app of this sort, you have your usual set of pages revolving around a home page, a search results page, a details page, and so on.

Old School Multi-Page Design

If you had to build this app a few years ago, you would have taken an approach that involved multiple, individual pages. The flow would have looked something like this:



New School Single-Page Apps

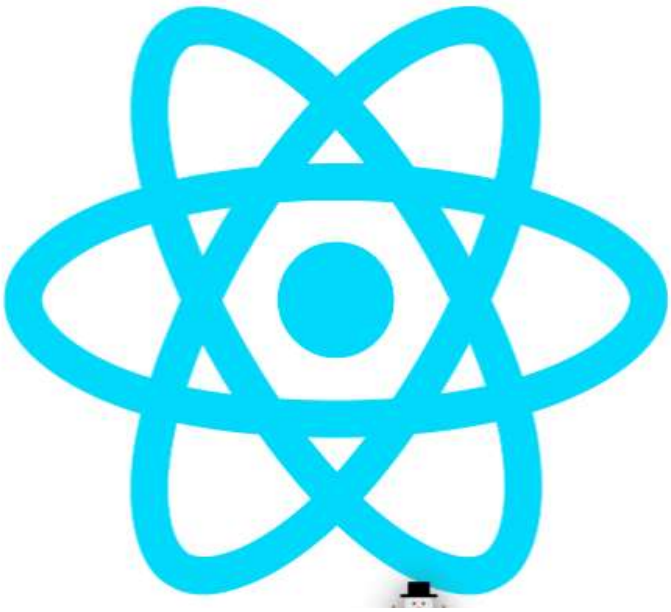
modern apps tend to adhere to what is known as a **Single-page App (SPA) model**. This is a world where you never navigate to different pages or ever even reload a page. Instead, the different views of your app are loaded and unloaded into the same page itself.

As users interact with our app, we replace the contents of the dotted red region with the data and HTML that matches what the user is trying to do. The end result is a much more fluid experience. You can even use a lot of visual techniques to have your new content transition in nicely just like you might see in cool apps on your mobile device or desktop. This sort of stuff is simply not possible when navigating to different pages. When building single-page apps, there are three major issues that you'll encounter at some point:

1. In a single page application, the bulk of your time will be spent keeping your data in-sync with your UI
2. Manipulating the DOM is really REALLY slow.
3. Working with HTML templates can be a pain.

Meet React

Facebook (and Instagram) decided that enough is enough. Given their boatload of experience with single-page apps, they released a library called **React** to not only address these shortcomings, but to also change how we think about building single-page apps:



1. Automatic UI State Management
2. Lightning-fast DOM Manipulation
3. APIs to Create Truly Composable UIs
4. Visuals Defined Entirely in JavaScript

Installation

Step 1: Download NodeJS

https://nodejs.org/en

Step2: Install NodeJS

Step 3: Open CMD prompt change Director

D:\csec>

Type node –v

D:\csec>node –v

To Display Version of NodeJS

Step 4: Install React App

npm install –g create-react-app

Step 5: Create new Project

Create-react-app demo-project

Step 6: change dirctory

Down-load Visual-Studio

https://code.visualstudio.com/download

Type code . To enter visual studio

Go to File Menu click Open Folder- tp specify location of folder.

Go to Terminal Menu Select New Terminal

Type npm start

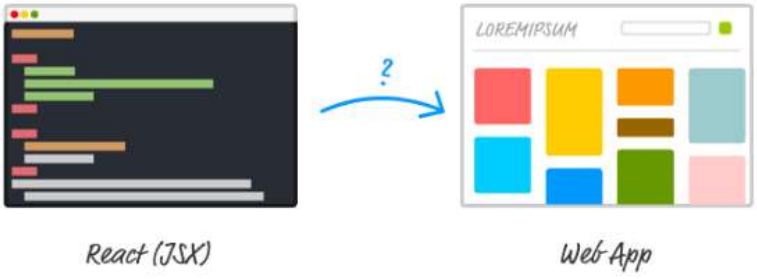
Building Your First React App

your web apps (and everything else your browser displays) are made up of HTML, CSS, and JavaScript:

if your web app was written using React or some other library like Angular, Knockout, or jQuery. **The end result** has to be some combination of HTML, CSS, and JavaScript. Otherwise, your browser really won't know what to do.

Now, here is where the specialness of React comes in. **Besides normal HTML, CSS, and JavaScript, the bulk of your React code will be written in something known as JSX**. JSX is a language that allows you to easily mix JavaScript and HTML-like tags to define user interface (UI) elements and their functionality.

To build a web app using React, we need a way to take our JSX and convert it into plain old JavaScript that your browser can understand:

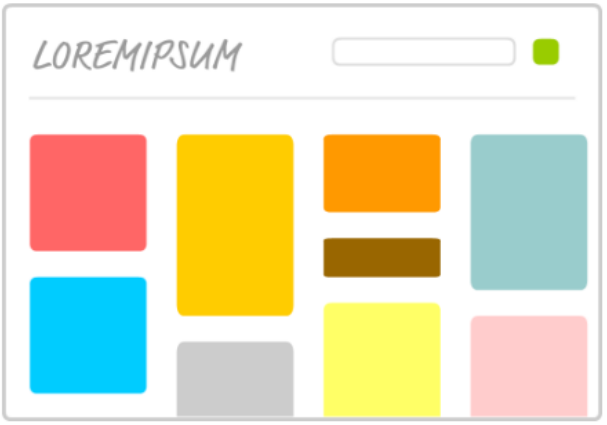


- Set up a development environment around Node and a handful of build-tools

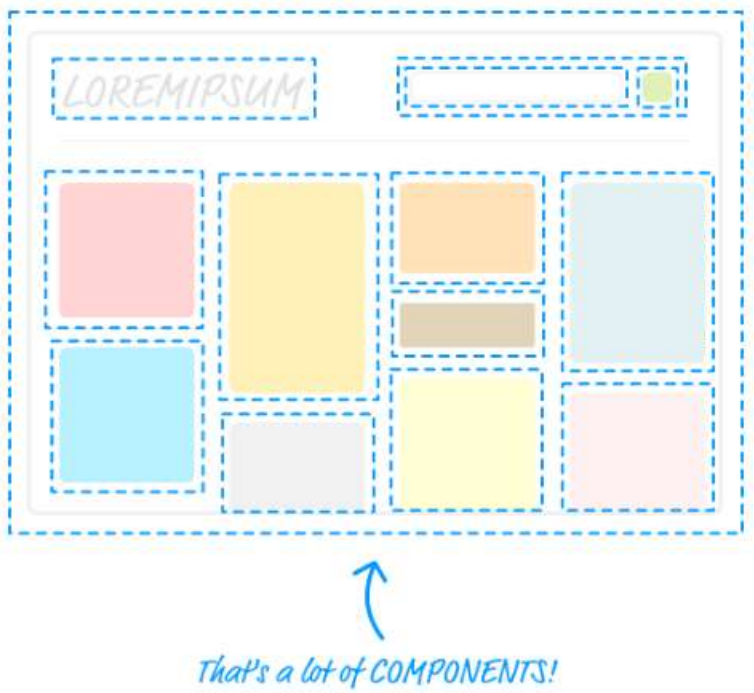
- Let your browser automatically convert JSX to JavaScript at runtime.

Components in React

Components are one of the things that make React...well, React! They are one of the primary ways you have for defining the visuals and interactions that make up what people see when they use your app. Let's say this is what your finished app looks like:

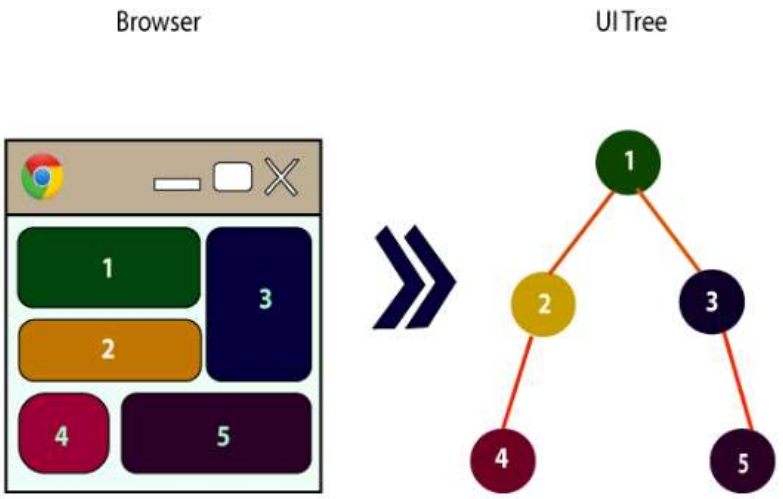


This is the finished sausage. During development, viewed from the lens of a React project, things might look a little less appealing. Almost every part of this app's visuals would be wrapped inside a self-contained module known as a **component**. To highlight what "almost every" means here, take a look at the following diagram:



each dotted line represents an individual component that is responsible for both what you see as well as any interactions that it may be responsible for. Don't let this scare you. While this looks really complicated, as you will see shortly, it will start to make a whole lot of sense once you've had a chance to play with components and some of the awesome things that they do...or at least try really hard to do

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.



In ReactJS, we have mainly two types of components. They are

1. Functional Components
2. Class Components

Functional Components

In React, function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input and returns what should be rendered.

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

```
import React from 'react'
function hello()
{
  return <div> Welcome to  ReactJS Program</div>
}

export default hello;
```

arrow function component

```
import React from 'react'
const hello = () =>
{
  return (
      <div>
          <h1>Welcome to React Js Programm</h1>
      </div>
  )
}
export default hello;
```

Class Components

Class components are more complex than functional components. It requires you to extend from React. Component and create a render function which returns a React element. You can pass data from one class to other class components

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>This is main component.</div>
    );
  }
}
```

```
import React, {Component} from 'react'
export default class App extends Component
{
  render()
  {

  return (
      <div>
          <h1>Welcome to React Js Programm for Class
Component</h1>
      </div>
  )
}
}
```

Styling in React

For generations, mankind (and probably really smart dolphins) have styled their HTML content using CSS. Things were good. With CSS, you had a good separation between the content and the presentation. The selector syntax gave you a lot of flexibility in choosing which elements to style and which ones to skip.

You couldn't even find too many issues to hate the *whole cascading thing* that CSS is all about.

While React doesn't actively hate CSS, it has a different view when it comes to styling content. As we've seen so far, one of React's core ideas is to have our app's visual pieces be self-contained and reusable. That is why the HTML elements and the JavaScript that impacts them are in the same bucket we call a **component**

CSS in React is used to style the React App or Component. The **style** attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time. It accepts a JavaScript object in **camelCased** properties rather than a CSS string. There are many ways available to add styling to your React App or Component with CSS. Here, we are going to discuss mainly **four** ways to style React Components, which are given below:

1. Inline Styling
2. CSS Stylesheet
3. CSS Module
4. Styled Components

1. Inline Styling

The inline styles are specified with a JavaScript object in camelCase version of the style name. Its value is the style?s value, which we usually take in a string.

App.js

import React from 'react';

import ReactDOM from 'react-dom';

```
class App extends React.Component {
  render() {
    return (
      <div>
      <h1 style={{color: "Green"}}>Hello React JS
Programming!</h1>
      <p>Here, you can used INLINE-STYLE.</p>
      </div>
    );
  }
}
export default App;
```

2. CSS Stylesheet

You can write styling in a separate file for your React application, and save the file with a .css extension. Now, you can **import** this file in your application.

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './App.css';

class App extends React.Component {
  render() {
    return (
      <div>
      <h1> Hello React JS Programming </h1>
      <p> Here, you can used CSS-STYLE SHEET.</p>
      </div>
```

```
    );
  }
}
export default App;
```

App.css

```
div {
  background-color: #008080;
  color: yellow;
  padding: 40px;
  font-family: Arial;
  text-align: center;
}
```

3. CSS Module

CSS Module is another way of adding styles to your application. It is a **CSS file** where all class names and **animation** names are scoped locally by default. It is available only for the component which imports it, means any styling you add can never be applied to other components without your permission, and you never need to worry about name conflicts. You can create CSS Module with the **.module.css** extension like a **myStyles.module.css** name.

4. Styled Components

Styled-components is a **library** for React. It uses enhance CSS for styling React component systems in your application, which is written with a mixture of JavaScript and CSS.

The styled-components provides:

- Automatic critical CSS
- No class name bugs
- Easier deletion of CSS
- Simple dynamic styling

- Painless maintenance

Installation

The styled-components library takes a single command to install in your React application. which is:

$ npm install styled-components --save

Creating Complex Components

We learned that components are the primary ways through which React allows our visual elements to behave like little reusable bricks that contain all of the HTML, JavaScript and styling needed to run themselves. Beyond reusability, there is another major advantage components bring to the table. They allow for ***composability***. You can combine components to create more complex components.

we will look at two things:

i. The boring technical stuff that you need to know.
ii. The boring stuff you need to know about how to identify components when you look at a bunch of visual elements

From Visuals to Components

The various examples we've looked at so far have been pretty basic. They were great for highlighting technical concepts, but they weren't great for preparing you for the real world:

In the real world, what you'll be asked to implement in React will never be so simple as a list of names or...colorful blocks of vowels. Instead, you'll be given a visual of some complex user interface. That visual can take many forms - such as a scribble, diagram, screenshot, video, redline, comp, etc. It is up to you to bring all of those static pixels to life, and we are going to get some hands-on practice in doing just that.

What we are going to do is build a simple color palette card:

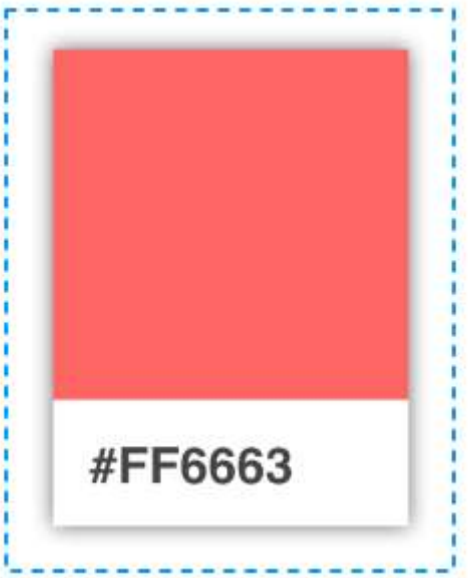
our mission is to re-create one of these cards using React.

you simplify and make sense of even the most complex user interfaces. This approach involves two steps:

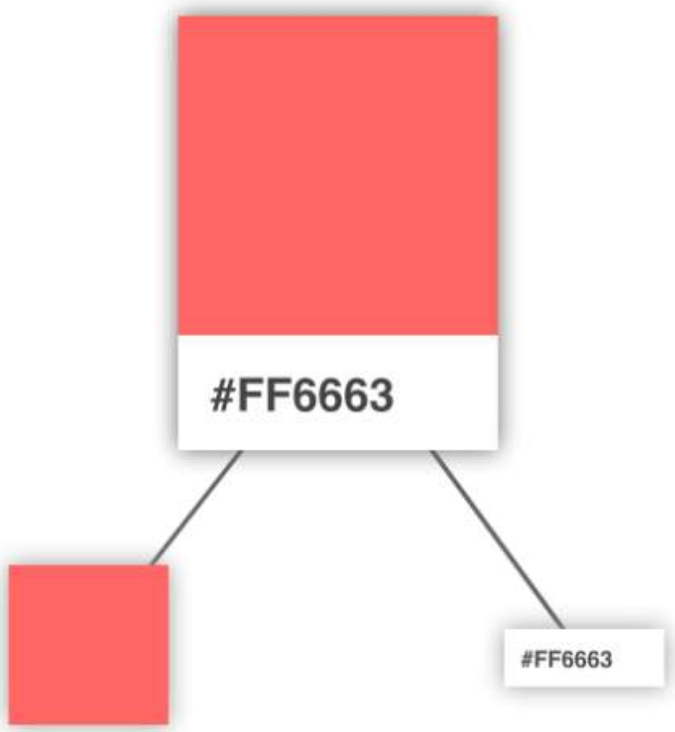
i. Identify the major visual elements
ii. Figure out what the components will be

Identifying the Major Visual Elements

The first step is to identify all of the visual elements we are dealing with. No visual element is too minor to omit - at least, not initially. The easiest way to start identifying the relevant pieces is to start with the obvious visual elements and then diving into the less obvious ones.

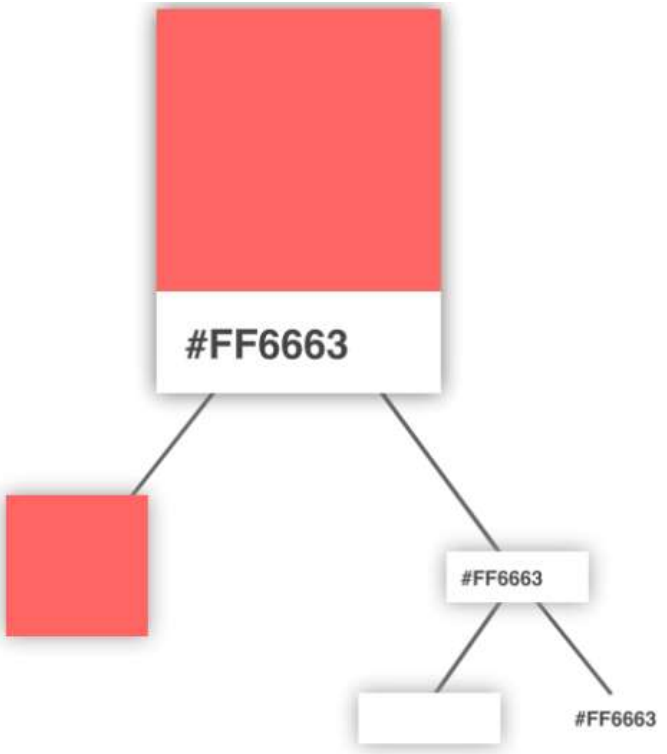


Within the card, you'll see that there are two distinct regions. The top region is a square area that displays a particular color. The bottom region is a white area that displays a hex value.



Arranging your visuals into this tree-like structure (aka a **visual hierarchy**) is a good way to get a better feel for how your visual elements are grouped. The goal of this exercise is to identify the important visual elements and break them into a parent/child arrangement until you can divide them no further.

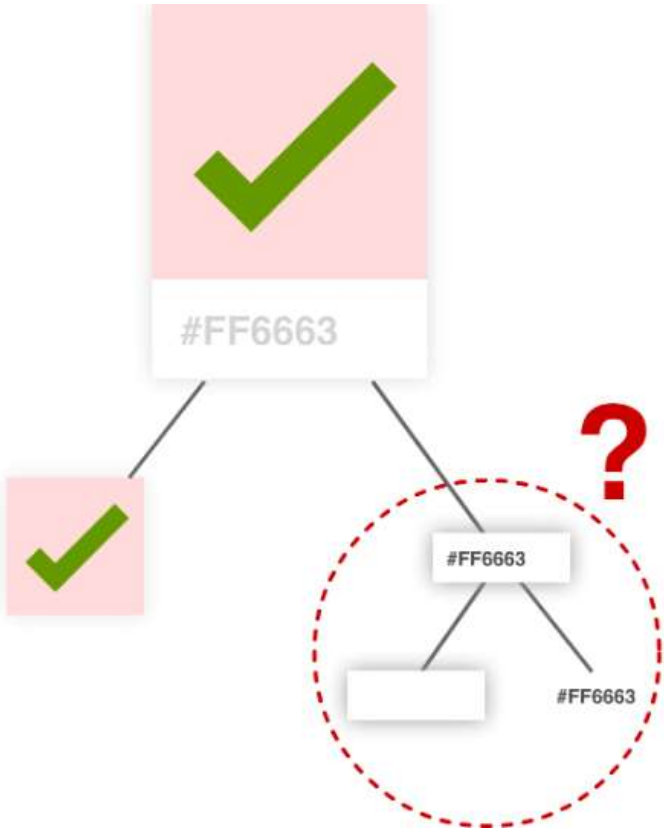
we can see that our colorful square isn't something that we can divide further. That doesn't mean we are done, though. We can further divide the label from the white region that surrounds it. Right now, our visual hierarchy looks as follows with our label and white region occupying a separate spot in our tree:



Identifying the Components

This is where things get a little interesting. We need to figure out which of the visual elements we've identified will be turned into a component and which ones will not. Not every visual element will need to be turned into a component, and we certainly don't want to create only a few extremely complex components either.

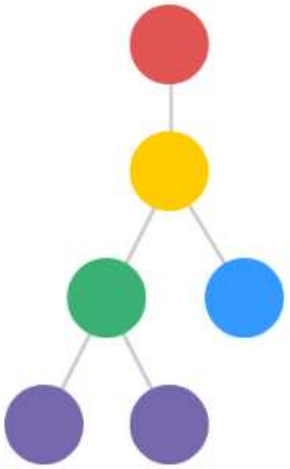
There is an art to figuring out what visual elements become part of a component and which ones don't. **The general rule is that our components should do just one thing.** If you find that your potential component will end up doing too many things, you probably want to break your component into multiple components. On the flipside, if your potential component does too little



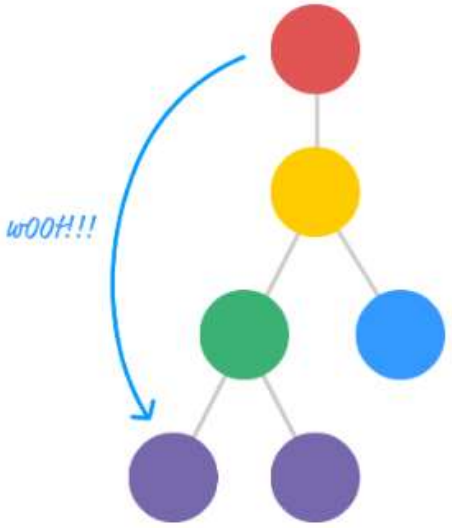
Transferring Properties

Problem Overview

Let's say that you have a deeply nested component, and its hierarchy (modeled as awesomely colored circles) looks as follows:



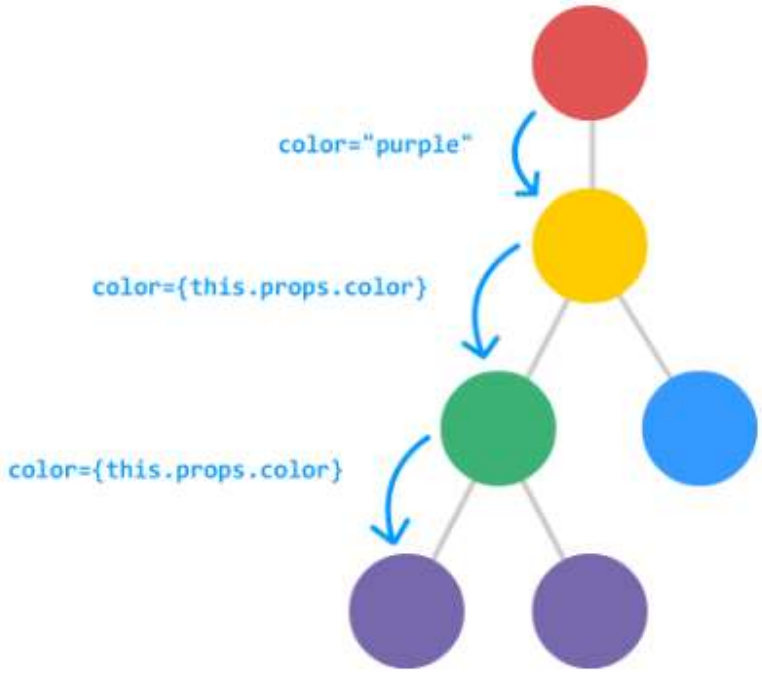
What you want to do is pass a property from your red circle all the way down to our purple circles where it will be used. What we can't do is this very obvious and straightforward thing:



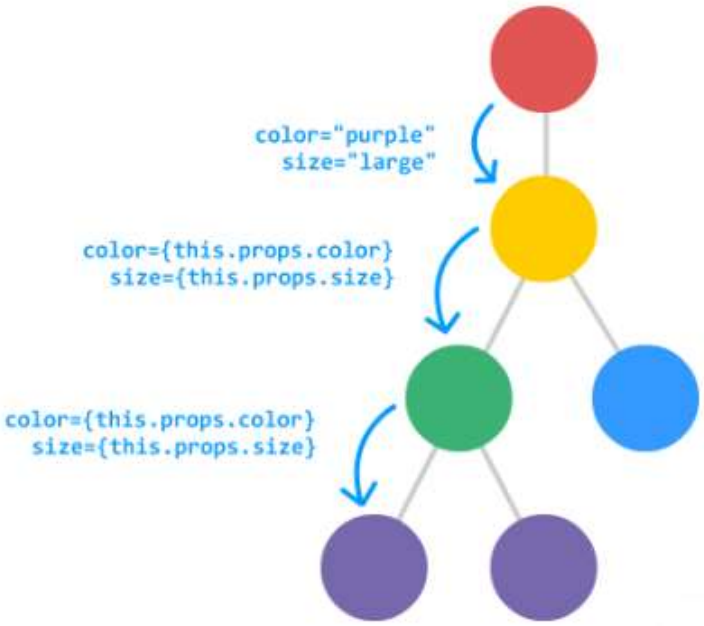
You can't pass a property directly to the component or components that you wish to target. The reason has to do with how React works. **React enforces a chain of command**

where properties have to flow down from a parent component to an immediate child component.

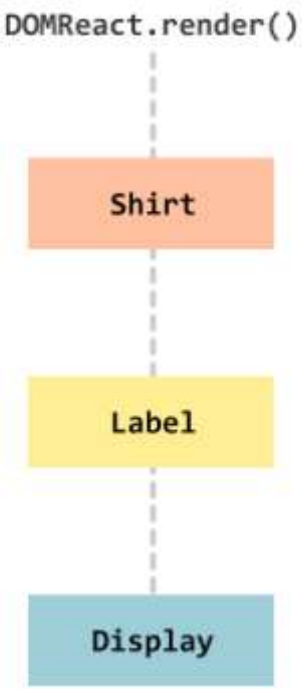
If we had to send a property called color from the component representing our red circle to the component representing our purple circle, its path to the destination would look something like this:



Now, imagine we have two properties that we need to send:



What we have is a Shirt component that relies on the output of the Label component which relies on the output of the Display component. (Try saying that sentence five time fast!) Anyway, the component hierarchy looks as follows



When you run this code, what gets output is nothing special. It is just three lines of text:

Dealing With State in React

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and how it will render. They are also responsible for making a component dynamic and interactive.

A state must be kept as simple as possible. It can be set by using the **setState()** method and calling setState() method triggers UI updates. A state represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

Defining State

To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using this.state. The '**this.state**' property can be rendered inside **render()** method.

```
class Greetings extends React.Component {
  state = {
    name: "PHP"
  };
    render() {
      return(
```

```
        <div>
            {this.state.name}
        </div>
    )
  }
}
```

Changing the State

We can change the component state by using the setState() method and passing a new state object as the argument.

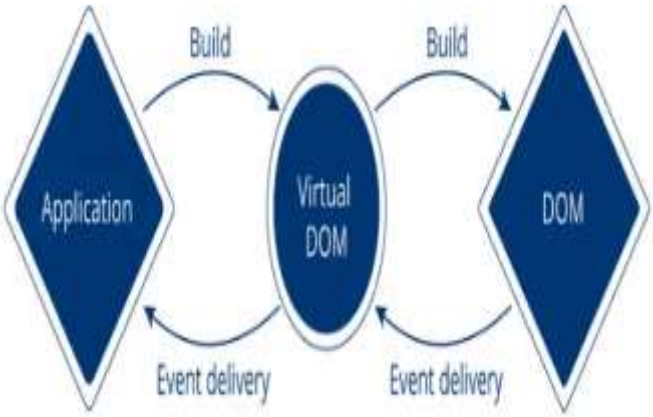
```
class Greetings extends React.Component {
  state = {
    name: "PHP"
  };
  updateName() {
    this.setState({ name: "React JS" });
  }
  render() {
     return(
        <div>
            {this.state.name}
        </div>
    )
  }
}
```

Going from Data to UI in React

- When you are building your apps, thinking in terms of props, state, components, JSX tags, render methods may be the last thing on your mind.
- Most of the time, you are dealing with data in the form of objects, arrays, and other data structures that have no knowledge (nor interest) in React or anything visual.
- Bridging the gulf between your data and what you eventually see.

Events in React

- Event handling essentially allows the user to interact with a webpage and do something specific when a certain event like a click or a hover happens.
- When the user interacts with the application, events are fired, for example, mouseover, key press, change event, and so on.
- The application must handle events and execute the code.
- In short, events are the actions to which javascript can respond.



•

some general events that you would see in and out when dealing with react based websites:

- Clicking an element
- Submitting a form
- Scrolling page
- Hovering an element
- Loading a webpage
- Input field change
- User stroking a key
- Image loading

Event attributes:

- onmouseover : The mouse is moved over an element
- onmouseup : The mouse button is released
- onmouseout : The mouse is moved off an element
- onmousemove: The mouse is moved
- Onmousedown: mouse button is pressed
- onload : A image is done loading
- onunload: Existing the page
- onblur : Losing Focus on element

- onchange : Content of a field changes
- onclick: Clicking an object
- ondblclick: double clicking an object
- onfocus element getting a focus
- Onkeydown: pushing a keyboard key
- Onkeyup: keyboard key is released
- Onkeypress: keyboard key is pressed
- Onselect: text is selected

the event handlers triggered in the bubbling phase:

- MouseEvents

  onClick

  onDrag

  onDoubleClick
- Keyboard Events

  onKeyDown

  onKeyPress

  onKeyUp
- Focus Events

  onFocus

  onBlur

```
import React from "react";

const App = () => {
  return (
    <button onClick={() => alert("Hello!")}>Say Hello</button>
  );
};
export default App;
```

```
const App = () => {
```

```
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
      <button onClick={() => setCount(count -
1)}>Decrement</button>
    </div>
  );};
export default App;
```