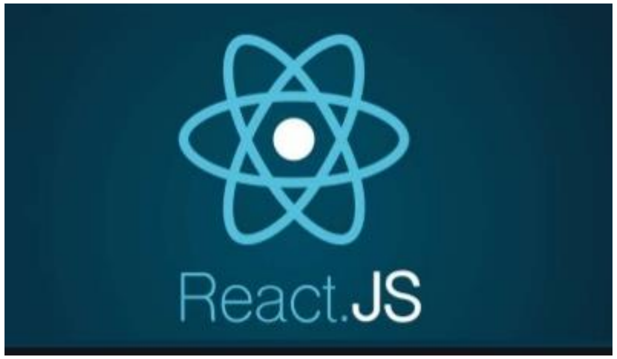

Presentation by Uplatz

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

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

## Table Of Contents:

**Better Understanding Of JSX:**

- ES5 (the ES stands for ECMAScript) is basically "regular JavaScript."
- The 5th update to JavaScript, ES5 was finalized in 2009. It has been supported by all major browsers for several years.
- Therefore, if you've written or seen any
- JavaScript in the recent past, chances are it was ES5.
- ES6 is a new version of JavaScript that adds some nice syntactical and functional additions.
- It was finalized in 2015.
- ES6 is almost fully supported (**http://kangax.github.io/compat-table/es6/**) by all major browsers.

- But it will be some time until older versions of web browsers are phased out of use.
- For instance, Internet Explorer 11 does not support ES6, but has about 12% of the browser market share.
- **In order to reap the benefits of ES6 today, we have to do a few things to get it to work in as many browsers as we can:**
- 1 We have to transpile our code so that a wider range of browsers understand our JavaScript. This means converting ES6 JavaScript into ES5 JavaScript.
- 2. We have to include a shim or polyfill that provides additional functionality added in ES6 that a browser may or may not have.

- Most of the code we'll write in this series will be easily translatable to ES5.
- In cases where we use ES6, we'll introduce the feature at first and then walk through it.
- As we'll see, all of our React components have a **render** function that specifies what the HTML output of our React component will be.
- JavaScript eXtension, or more commonly JSX, is a React extension that allows us to write JavaScript that looks like HTML.
- To see what this means, imagine we had a React component that renders an
- h1 HTML tag. JSX allows us to declare this element in a manner that closely resembles HTML:

```
class HelloWorld extends React.Component {
 render() {
 return (
 <h1 className='large'>Hello World</h1>
 );
 }
}
```

**Hello World**

- The render() function in the HelloWorld component looks like it's returning HTML, but this is actually JSX.
- The JSX is translated to regular JavaScript at
runtime.
- That component, after translation, looks like this:

```
class HelloWorld extends React.Component {
 render() {
 return (
 React.createElement(
 'h1',
 {className: 'large'},
 'Hello World'
 )
 );}}
```

- While JSX looks like HTML, it is actually just a terser way to write a React.createElement() declaration.
- When a component renders, it outputs a tree of React elements or a virtual representation of the HTML elements this component outputs.

- React will then determine what changes to make to the actual DOM based on this React element representation.
- In the case of the HelloWorld component, the HTML that React writes to the DOM will look like this:

**<h1 class='large'>Hello World</h1>**

- The class extends syntax we used in our first React

component is ES6 syntax.

- It allows us to write objects using a familiar Object-Oriented style.
- In ES5, the class syntax might be translated as:

**var HelloWorld = function() {}**

**Object.extends(HelloWorld, React.Component)**

**HelloWorld.prototype.render = function() {}**

- Because JSX is JavaScript, we can't use JavaScript reserved words.
- This includes words like class and for .
- React gives us the attribute className .
- We use it in HelloWorld to set the large class on our h1 tag.
- There are a few other attributes, such as the for

attribute on a label that React translates into htmlFor as for is also a reserved word.

- We'll look at these when we start using them.
- If we want to write pure JavaScript instead of rely on a JSX compiler, we can

just write the React.createElement() function and not worry about the layern of abstraction.

- But we like JSX. It's especially more readable with complex components.
- Consider the following JSX:

```
<div>
 <img src="profile.jpg" alt="Profile photo" />
 <h1>Welcome back Ari</h1>
</div>
```

- The JavaScript delivered to the browser will look like this:

```
React.createElement("div", null,
 React.createElement("img", {src: "profile.jpg", alt:
"Profile photo"}),
 React.createElement("h1", null, "Welcome back
Ari") );
```

- Again, while you can skip JSX and write the latter directly, the JSX syntax is
- well-suited for representing nested HTML elements.

**The state of things**

- React does not allow us to modify this.props on our components for good reason.
- Imagine if we passed in the title prop to the Header component and the Header component was able to modify it.

**How do we know what the title is of the Header component?**

- We set ourselves up for race-conditions,confusing data state, and it would be an all-around bad idea to modify a variable passed to a child component by a parent component.

- However, sometimes a component needs to be able to update its own state.
- For example, setting an active flag or updating a timer on a stopwatch, for instance.
- While it's preferable to use props as much as we can, sometimes we need to hold on to the state of a component.
- To handle this, React gives us the ability
- to hold state in our components.
- state in a component is intended to be completely internal to the Component and its children (i.e. accessed by the component and any children it used).
- Similar to how we access props in a component, the state can be accessed via this.state in a component.

- Whenever the state changes (via the this.setState() function), the component will rerender.
- For instance, let's say we have a simple clock component that shows the current time
- Even though this is a simple clock component, it does retain state in that it needs to know what the current time is to display.
- Without using state , we could set a timer and rerender the entire React component, but other
- components on the page may not need rerendering... this would become a headache and slow when we integrate it into a more complex application.
- Instead, we can set a timer to call rerender inside the component and change just the internal state of this component.

- Let's take a stab at building this component. First, we'll create the component we'll call Clock .
- Before we get into the state, let's build the component and create the render() function.
- We'll need to take into account the number and prepend a zero ( 0 ) to the number if the numbers are smaller than 10 and set the am/pm appropriately.
- The end result of the render() function might look

something like this:

**4:00:52 pm**

**47**

class Clock extends React {

```
class Clock extends React{
render() {
 const currentTime = new Date(),
 hours = currentTime.getHours(),
 minutes = currentTime.getMinutes(),
 seconds = currentTime.getSeconds(),
 ampm = hours >= 12 ? 'pm' : 'am';
 return (
 <div className="clock">
 {
 hours == 0 ? 12 :
 (hours > 12) ?
 hours - 12 : hours
}:{
 minutes > 9 ? minutes : `0${minutes}`
 }:{
 seconds > 9 ? seconds : `0${seconds}`
 } {ampm}
 </div>
 )
 }
}
```

- Alternatively, we could use the short snippet to handle padding the clock time:

```
("00" + minutes).slice(-2)
```

- But we've opted to be more clear with the previous code.

- If we render our new Clock component, we will only get a time rendered everytime the component itself rerenders. It's not a very useful clock (yet).
- In order to convert our static time display Clock component into a clock that displays the time, we'll need to update the time every second.
- In order to do that, we'll need to track the current time in the state of the component.
- To do this, we'll need to set an initial state value.
- To do so, we'll first create a getTime() function that returns a javascript object **containing hours** , **minutes** , **seconds and ampm** values.
- We will call this function to set our state.

```
class Clock extends React.Component {
 //...
 getTime() {
 const currentTime = new Date();
 return {
 hours: currentTime.getHours(),
 minutes: currentTime.getMinutes(),
 seconds: currentTime.getSeconds(),
 ampm: currentTime.getHours() >= 12 ? 'pm' : 'am'
 }
 }
 // ...
}
```

- In the ES6 class style, we can set the initial state of the component in the constructor() by setting this.state to a value (the return value of our

getTime() function).

```
constructor(props) {
 super(props);
 this.state = this.getTime();
 }
```

- this.state will now look like the following object

```
{
 hours: 11,
 minutes: 8,
 seconds: 11,
 ampm: "am"
}
```

- The first line of the constructor should always call super(props) .
- If you forget this, the component won't like you very much (i.e. there will be errors).
- Now that we have a t**his.state** defined in our **Clock** component, we can reference it in the **render()** function using the **this.state** .
- Let's update our render() function to grab the values from this.state :

```
class Clock extends React.Component {
 // ...
 render() {
 const {hours, minutes, seconds, ampm} = this.state;
 return (
<div className="clock">
 {
 hours === 0 ? 12 :
 (hours > 12) ?
 hours - 12 : hours
 }:{
 minutes > 9 ? minutes : `0${minutes}`
 }:{
 seconds > 9 ? seconds : `0${seconds}`
 } {ampm}
 </div>
 )
 }
}
```

- Instead of working directly with data values, we can now update the state of the component and separate the render() function from the data management.
- In order to update the state, we'll use a special function called: **setState**() ,
- which will trigger the component to rerender.
- We need to call **setState()** on the this value of the
- component as it's a part of the **React.Component** class we are subclassing.
- In our Clock component, let's use the native **setTimeout()** JavaScript function to create a timer to update the **this.state** object in 1000 milliseconds. We'll place this functionality in a function as we'll want to call this again

```
class Clock extends React.Component {
 // ...
 constructor(props) {
 super(props);
 this.state = this.getTime();
 }
 // ...
 componentDidMount() {
 this.setTimer();
 }
 // ...
 setTimer() {
 clearTimeout(this.timeout);
 this.timeout = setTimeout(this.updateClock.bind(this),
1000);
```

```
 }// ...
 updateClock() {
 this.setState(this.getTime, this.setTimer);
 }// ...
}
```

- To start updating the timer immediately after the our
- component has been rendered, we call this.setTimer() in a React component lifecycle method called

componentDidMount .

- In the **updateClock()** function we'll want to update the state with the new time.
- We can now update the state in the **updateClock()** function:

```
class Clock extends React.Component {
 // ...
 updateClock() {
 this.setState(this.getTime, this.setTimer);
 }
 // ...
}
```

- The component will be mounted on the page and will update the time every second (approximately every 1000 milliseconds)
- Now the component itself might rerender slower than the timeout function gets called again, which would cause a rerendering bottleneck and needlessly using up precious battery on mobile devices.

- Instead of calling the setTimer() function after we call this.setState() , we can pass a second

argument to the this.setState() function which will be guaranteed to be called after the state has been updated.

```
class Clock extends React.Component {
 // ...
 updateClock() {
 const currentTime = new Date();
 this.setState({
 currentTime: currentTime
 }, this.setTimer);
 }
 // ...
}
```

## Props in React

Introduction

- props are used to pass data and methods from a parent component to a child component.

**Interesting things about props**

- 1. They are immutable.
- 2. They allow us to create reusable components.

**Basic example**

```
class Parent extends React.Component{
 doSomething(){
 console.log("Parent component");
 }
 render() {
 return <div>
```

```
<Child
 text="This is the child number 1"
 title="Title 1"
 onClick={this.doSomething} />
 <Child
 text="This is the child number 2"
 title="Title 2"
 onClick={this.doSomething} />
 </div>
 }
}
class Child extends React.Component{
 render() {
 return <div>
```

```
<h1>{this.props.title}</h1>
 <h2>{this.props.text}</h2>
 </div>
 }
}
```

As you can see in the example, thanks to props we can create reusable components.

**Default props**

- defaultProps allows you to set default, or fallback, values for your component props.
- defaultProps are useful when you call components from different views with fixed props, but in some views you need to pass different value.

**Syntax**

ES5

```
var MyClass = React.createClass({
 getDefaultProps: function() {
 return {
 randomObject: {},
};
 }
}
```

ES6

```
class MyClass extends React.Component {...}
MyClass.defaultProps = {
 randomObject: {},
 ...
}
```

ES7

```
class MyClass extends React.Component {
 static defaultProps = {
 randomObject: {},
 ...
 };
}
```

- The result of getDefaultProps() or defaultProps will be cached and used to ensure that this.props.randomObject will have a value if it was not specified by the parent component.

**PropTypes**

- propTypes allows you to specify what props your component needs and the type they should be.
-

- Your component will work without setting propTypes, but it is good practice to define these as it will make your component more eadable, act as documentation to other developers who are reading your component, and during development,
- React will warn you if you you try to set a prop which is a different type to the definition you have set for it.

Some primitive propTypes and commonly useable propTypes are -

**optionalArray: React.PropTypes.array,**

**optionalBool: React.PropTypes.bool,**

**optionalFunc: React.PropTypes.func,**

**optionalNumber: React.PropTypes.number,**

**optionalObject: React.PropTypes.object,**

**optionalString: React.PropTypes.string,**

**optionalSymbol: React.PropTypes.symbol**

- If you attach isRequired to any propType then that prop must be supplied while creating the instance of that component.
- If you don't provide the required propTypes then component instance can not be created.

**Syntax**

ES5

```
var MyClass = React.createClass({
 propTypes: {
 randomObject: React.PropTypes.object,
 callback: React.PropTypes.func.isRequired,
 ...
 }}
```

ES6

```
class MyClass extends React.Component {...}
MyClass.propTypes = {
 randomObject: React.PropTypes.object,
 callback: React.PropTypes.func.isRequired,
 ...
};
```

ES7

```
class MyClass extends React.Component {
 static propTypes = {
 randomObject: React.PropTypes.object,
 callback: React.PropTypes.func.isRequired,
 ...
 };
}
```

More complex props validation

➢ In the same way, PropTypes allows you to specify more complex validation

**Validating an object**

```
...
 randomObject: React.PropTypes.shape({
 id: React.PropTypes.number.isRequired,
 text: React.PropTypes.string,
 }).isRequired,
...
```

**Validating on array of objects**

```
...
 arrayOfObjects:
React.PropTypes.arrayOf(React.PropTypes.shape({
 id: React.PropTypes.number.isRequired,
```

**text: React.PropTypes.string,**
**})).isRequired,**

...

Passing down props using spread operator

Instead of

**var component = <Component foo={this.props.x} bar={this.props.y} />;**

- Where each property needs to be passed as a single prop value you could use the spread operator ... Supported for arrays in ES6 to pass down all your values.
- The component will now look like this.

**var component = <Component {...props} />;**

- Remember that the properties of the object that you pass in are copied onto the component's props.

The order is important

Later attributes override previous ones.

**var props = { foo: 'default' };**

**var component = <Component {...props} foo={'override'} />;**

**console.log(component.props.foo); // 'override'**

- Another case is that you also can use spread operator to pass only parts of props to children components, then you can use destructuring syntax from props again.
- It's very useful when children components need lots of props but not want pass them one by one.

**const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };**

**var component = <Component {...{foo, bar}} />;**

```
const { foo, bar } = component.props
 console.log(foo, bar); // 'foo bar'
```

**Props.children and component composition**

- The "child" components of a component are available on a special prop, props.children.
- This prop is very useful for "Compositing" components together, and can make JSX markup more intuitive or reflective of the intended final structure of the DOM:

```
var SomeComponent = function () {
 return (
 <article className="textBox">
 <header>{this.props.heading}</header>
 <div className="paragraphs">
 {this.props.children}
 </div>
</article>
 );
}
```

- Which allows us to include an arbitrary number of sub-elements when using the component later:

```
var ParentComponent = function () {
 return (
 <SomeComponent heading="Amazing Article Box" >
 <p className="first"> Lots of content </p>
 <p> Or not </p>
 </SomeComponent>
 );
}
```

**Props.children can also be manipulated by the component.**

- Because props.children may or may not be an array,React provides utility functions for them as React.Children.
- Consider in the previous example if we had wanted to wrap each paragraph in its own <section> element:

```
var SomeComponent = function () {
 return (
 <article className="textBox">
 <header>{this.props.heading}</header>
 <div className="paragraphs">
{React.Children.map(this.props.children, function
(child) {
 return (
 <section className={child.props.className}>
.
React.cloneElement(child)
 </section>
 );
 }))}
 </div>
 </article>
 );
}
```

- Note the use of React.cloneElement to remove the props from the child <p> tag - because props are immutable,these values cannot be changed directly.
- Instead, a clone without these props must be used.

Additionally, when adding elements in loops, be aware of how React reconciles children during a rerender, andstrongly consider including a globally unique key prop on child elements added in a loop

**Detecting the type of Children components**

- Sometimes it's really useful to know the type of child component when iterating through them.
- In order to iterate through the children components you can use React Children.map util function:

```
React.Children.map(this.props.children, (child) => {
 if (child.type === MyComponentType) {
 ...
 }
});
```

- The child object exposes the type property which you can compare to a specific component

**PropTypes**

- You may have noticed we use props quite a bit in our components.
- For the most part, we'll expect these to be a particular type or set of types (aka an

object or a string ).

- React provides a method for defining and validating

these types that allow us to easily expose a component API.

- Not only is this a good practice for documentation purposes, it's great forbuilding reusable reactcomponents
- The prop-types object exports a bunch of different types which we can use to define what type a component's prop should be.

Uplatz

- We can define these using the propTypes method in the ES6 class-style React prop

```
class Clock extends React.Component {
 // ...
}
Clock.propTypes = {
 // key is the name of the prop and
 // value is the PropType
}
```

- From within this prop , we can define an object which has the key of a prop as the name of the prop we are defining and a value defines the type (ortypes) it should be defined as.
- For instance, the Header component we built a few days ago accepts a a prop called title and we expect it to be a string.

- **We can define it's type to be a string as such**
- First, we'll need to import the PropTypes object from the prop-types package using the import keyword again:

**import PropTypes from 'prop-types'**

- You can also use the PropTypes object directly in your browser by adding the following script tag in your page

**<script src="https://unpkg.com/prop-types@15.6/proptypes.min.js (https://unpkg.com/prop-types@15.6/proptypes.min.js)"></script>**

```
import PropTypes from 'prop-types'
class Header extends React.Component {
 // ...
}
```

```
Header.propTypes = {
 title: PropTypes.string
}
```

- React has a lot of types to choose from, exported on the PropTypes object and even allows for us to define a custom object type.
- Let's look at an overall list of available types:

**React exposes a few basic types we can use out of the box.**

| **type** | **example** | **class** |
|---|---|---|
| String | 'hello' | PropTypes.string |
| Number | 10, 0.1 | PropTypes.number |
| Boolean | true / false | PropTypes.bool |
| Function | const say => (msg) => console.log("Hello world") | |

| | | |
|---|---|---|
| | | PropTypes.func |
| Symbol | Symbol("msg") | PropTypes.symbol |
| Object | {name: 'Ari'} | PropTypes.object |
| Anything | 'whatever', 10, {} | |

- It's possible to tell React we want it to pass through anything that can be rendered by using **PropTypes.node :**

type exampleclass

**A rendererable10, 'hello' PropTypes.node**

```
Clock.propTypes = {
 title: PropTypes.string,
 count: PropTypes.number,
 isOn: PropTypes.bool,
 onDisplay: PropTypes.func,
 symbol: PropTypes.symbol,
user: PropTypes.object,
 name: PropTypes.node
}
```

- We've already looked at how to communicate from a parent component to a child component using props .
- We can communicate from a child component

to a parent component using a function.

- We'll use this pattern quite often when we want to manipulate a parent component from a child.

**Collection types**

- We can pass through iterable collections in our props .
- We've already seen how we can do this when we passed through an array with our activities.

- To declare a component's proptype as an array, we can use the PropTypes.array annotation.
- We can also require that an array holds only objects of a certain type using

**PropTypes.arrayOf([]) .**

| **type** | **example** | **class** |
|---|---|---|
| Array | [] | PropTypes.array |
| Array of numbers | [1, 2, 3] | PropTypes.arrayOf([type]) |
| Enum | ['Red', 'Blue'] | PropTypes.oneOf([arr]) |

- It's possible to describe an object that can be one of a few different types as well using **PropTypes.oneOfType([types]) .**

```
Clock.propTypes = {
 counts: PropTypes.array,
 users: PropTypes.arrayOf(PropTypes.object),
 alarmColor: PropTypes.oneOf(['red', 'blue']),
 description: PropTypes.oneOfType([
 PropTypes.string,
 PropTypes.instanceOf(Title)
 ]),
}
```

## Object types

- It's possible to define types that need to be of a certain shape or instance of a certain class.

| type | example | class |
|---|---|---|
| Object | {name: 'Ari'} | PropTypes.object |

| | | |
|---|---|---|
| **Number** | **object {count: 42}** | **PropTypes.objectOf()** |
| **Instance** | **new Message()** | **PropTypes.objectOf()** |
| **Object shape** | **{name: 'Ari'}** | **PropTypes.shape()** |

```
Clock.propTypes = {
 basicObject: PropTypes.object,
 numbers: PropTypes
 .objectOf(PropTypes.numbers),
 messages: PropTypes
 .instanceOf(Message),
 contactList: PropTypes.shape({
 name: PropTypes.string,
 phone: PropTypes.string,
 })
}
```

## React types

- We can also pass through React elements from a parent to a child.
- This is incredibly useful for building templates and providing customization with the templates.

| **type** | **example** | **class** |
|---|---|---|
| Element | <Title /> | PropTypes.element |

```
Clock.propTypes = {
 displayEle: PropTypes.element
}
```

- When we use element, React expects that we'll be able to accept a single child component.
- That is, we won't be able to pass multiple elements.

```
// Invalid for elements
<Clock displayElement={
 <div>Name</div>
 <div>Age</div>
}></Clock>
// Valid
<Clock displayElement={
 <div>
 <div>Name</div>
 <div>Age</div>
 </div>
}></Clock>
```

- It's possible to require a prop to be passed to a component by appending any of the proptype descriptions with .isRequired :

```
Clock.propTypes = {
 title: PropTypes.name.isRequired,
}
```

**Requiring types**

- Setting a prop as required is very useful for times when the component is dependent upon a prop to be passed in by it's parent component and won't
- work without it.

**Custom types**

- Finally, it's also possible to pass a function to define custom types.
- We can do this for a single prop or to validate arrays. The one requirement for the custom function is that if the validation does not pass, it expects we'll return an Error object:

**type example class**

Custom 'something_crazy' function(props, propName, componentName)

{}

CustomArray ['something',

'crazy‘]

PropTypes.arrayOf(function(props, propName,

componentName) {})

**UserLink.propTypes = {**

 **userWithName: (props, propName, componentName) => {**

 **if (!props[propName] || !props[propName].name) {**

 **return new Error(**

```
 "Invalid " + propName + ": No name property defined for
component " + componentName
 )
 } }}
```

**Default props**

- Sometimes we want to be able to set a default value for a prop.
- For instance, our <Header /> component, we built yesterday might not require a title to be
- passed.
- If it's not, we'll still want a title to be rendered, so we can define a common title instead by setting it's default prop value.

- To set a default prop value, we can use the defaultProps object key on the component

```
Header.defaultProps = {
 title: 'Github activity'
}
```

- to build our resuable components using the propTypes and defaultProps attributes of components.

## Summary:

- Better Understanding Of JSX
- state of things
- Props in React
- PropTypes
- Collection Types

## Thank You………

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