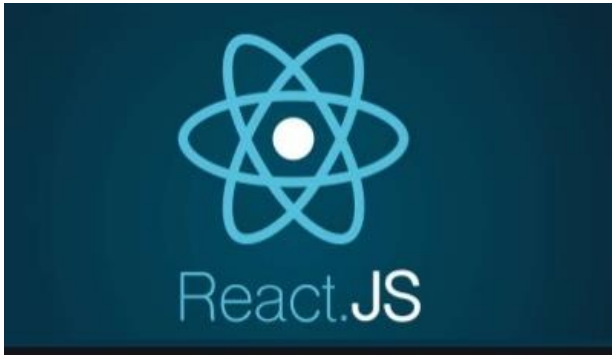

Presentation by Uplatz

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

Email: info@uplatz.com

Phone:+44 7836 212635

Uplatz

Table Of Contents:

What are props?

- Props is short for properties and they are used to pass data between React components.
- React's data flow between components is uni-directional (from parent to child only).

How do you pass data with props?

```
class ParentComponent extends Component {
   render() {
      return (
         <ChildComponent name="First Child" />
      );
   } }
const ChildComponent = (props) => {
```

return {props.name} ;

};

Code Explanation:

- Firstly, we need to define/get some data from the parent component and assign it to a child component's "prop" attribute.

<ChildComponent name="First Child" />

- "Name" is a defined prop here and contains text data. Then we can pass data with props like we're giving an argument to a function:

```
const ChildComponent = (props) => {
  // statements
};
```

- And finally, we use dot notation to access the prop data and render it:

return {props.name} ;

What is state?

- React has another special built-in object called state, which allows components to create and manage their own data.
- So unlike props, components cannot pass data with state, but they can create and manage it internally.

Here is an example showing how to use state:

```
class Test extends React.Component {
    constructor() {
        this.state = {
            id: 1,
            name: "test"
        };
```

```
}   render() {
      return (
        <div>
          <p>{this.state.id}</p>
          <p>{this.state.name}</p>
        </div>
      );
  }
}
```

How do you update a component's state?

- State should not be modified directly, but it can be modified with a special method called setState( ).

this.state.id = "2020"; // wrong

```
this.setState({        // correct
    id: "2020"
});
```

What happens when state changes?

- A change in the state happens based on user-input, triggering an event, and so on.
- Also, React components (with state) are rendered based on the data in the state.
- State holds the initial information.
- So when state changes, React gets informed and immediately re-renders the DOM – not the whole DOM, but only the component with the updated state. This is one of the reasons why React is fast.

And how does React get notified?

- with setState( ). The setState( ) method triggers the re-rendering process for the updated parts.
- React gets informed, knows which part(s) to change, and does it quickly without re-rendering the whole DOM.

there are 2 important points we need to pay attention to when using state:

- State shouldn't be modified directly – the setState( ) should be used
- State affects the performance of your app, and therefore it shouldn't be used unnecessarily

Can I use state in every component?

- Another important question you might ask about state is where exactly we can use it.

- In the early days, state could only be used in class components, not in functional components.
- That's why functional components were also known as stateless components.
- However, after the introduction of React Hooks, state can now be used both in class and functional components.
- If your project is not using React Hooks, then you can only use state in class components.

What are the differences between props and state?

- Components receive data from outside with props, whereas they can create and manage their own data with state
- Props are used to pass data, whereas state is for managing data

- Data from props is read-only, and cannot be modified by a component that is receiving it from outside
- State data can be modified by its own component, but is private (cannot be accessed from outside)
- Props can only be passed from parent component to child (unidirectional flow)
- Modifying state should happen with the setState ( ) method

Hello World

Without JSX

Here's a basic example that uses React's main API to create a React element and the React DOM API to render the

React element in the browser.

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="UTF-8" />
 <title>Hello React!</title>
 <!-- Include the React and ReactDOM libraries -->
 <script src="https://fb.me/react-15.2.1.js"></script>
 <script src="https://fb.me/react-dom-
15.2.1.js"></script>
 </head>
 <body>
 <div id="example"></div>
 <script type="text/javascript">
```

```
// create a React element rElement
 var rElement = React.createElement('h1', null, 'Hello, world!');
 // dElement is a DOM container
 var dElement = document.getElementById('example');
 // render the React element in the DOM container
 ReactDOM.render(rElement, dElement);
 </script>
 </body>
</html>
```

With JSX

- Instead of creating a React element from strings one can use JSX (a Javascript extension created by Facebook for adding XML syntax to JavaScript), which allows to write

var rElement = React.createElement('h1', null, 'Hello, world!');

as the equivalent (and easier to read for someone familiar with HTML)

var rElement = <h1>Hello, world!</h1>;

The code containing JSX needs to be enclosed in a <script type="text/babel"> tag. Everything within this tag will

be transformed to plain Javascript using the Babel library (that needs to be included in addition to the React

libraries).

So finally the above example becomes:

<!DOCTYPE html>

<html>

```
<head>
 <meta charset="UTF-8" />
 <title>Hello React!</title>
 <!-- Include the React and ReactDOM libraries -->
 <script src="https://fb.me/react-15.2.1.js"></script>
 <script src="https://fb.me/react-dom-
15.2.1.js"></script>
 <!-- Include the Babel library -->
 <script src="https://npmcdn.com/babel-
core@5.8.38/browser.min.js"></script>
 </head>
 <body>
 <div id="example"></div>
 <script type="text/babel">
```

```
// create a React element rElement using JSX
var rElement = <h1>Hello, world!</h1>;
// dElement is a DOM container
var dElement = document.getElementById('example');
// render the React element in the DOM container
ReactDOM.render(rElement, dElement);
</script>
</body>
</html>
```

Hello World Component

- A React component can be defined as an ES6 class that extends the base React.Component class.
- In its minimal form, a component must define a render method that specifies how the component renders to the DOM.

- The render method returns React nodes, which can be defined using JSX syntax as HTML-like tags.
- The following example shows how to define a minimal Component:

```
import React from 'react'
class HelloWorld extends React.Component {
 render() {
return <h1>Hello, World!</h1>
 }
}
export default HelloWorld
```

- A Component can also receive props.
- These are properties passed by its parent in order to specify some values the component cannot know by itself;

- a property can also contain a function that can be called by the component after certain events occur - for example, a button could receive a function for its onClick property and call it whenever it is clicked.
- When writing a component, its props can be accessed through the props object on the Component itself:

```
import React from 'react'
class Hello extends React.Component {
 render() {
 return <h1>Hello, {this.props.name}!</h1>
 }
}
export default Hello
```

- The example above shows how the component can render an arbitrary string passed into the name prop by its parent.
- Note that a component cannot modify the props it receives.
- A component can be rendered within any other component, or directly into the DOM if it's the topmost component, using ReactDOM.render and providing it with both the component and the DOM Node where you want the React

tree to be rendered:

import React from 'react'

import ReactDOM from 'react-dom'

import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))

- By now you know how to make a basic component and accept props. Lets take this a step further and introduce state.
- For demo sake, let's make our Hello World app, display only the first name if a full name is given.

```
import React from 'react'
class Hello extends React.Component {
 constructor(props){
 //Since we are extending the default constructor,
 //handle default activities first.
 super(props);
 //Extract the first-name from the prop
 let firstName = this.props.name.split(" ")[0];
```

Uplatz

```
 //In the constructor, feel free to modify the
 //state property on the current context.
 this.state = {
 name: firstName
 }
```

Components

Creating Components

This is an extension of Basic Example:

Basic Structure

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class FirstComponent extends Component {
 render() {
```

```
 return (
 <div>
 Hello, {this.props.name}! I am a FirstComponent.
 </div>
 );
 }
}
render(
 <FirstComponent name={ 'User' } />,
 document.getElementById('content')
);
```

- The above example is called a stateless component as it does not contain state (in the React sense of the word).

Stateless Functional Components

- In many applications there are smart components that hold state but render dumb components that simply receiveprops and return HTML as JSX.
- Stateless functional components are much more reusable and have a positive performance impact on your application.

They have 2 main characteristics:

1. When rendered they receive an object with all the props that were passed down
2. They must return the JSX to be rendered

// When using JSX inside a module you must import React

```
import React from 'react';
import PropTypes from 'prop-types';
const FirstComponent = props => (
 <div>
 Hello, {props.name}! I am a FirstComponent.
 </div>
);
//arrow components also may have props validation
FirstComponent.propTypes = {
 name: PropTypes.string.isRequired,
}
```

// To use FirstComponent in another file it must be exposed through an export call:

export default FirstComponent;

Stateful Components

- In contrast to the 'stateless' components shown above, 'stateful' components have a state object that can be
- updated with the setState method.
- The state must be initialized in the constructor before it can be set:

```
import React, { Component } from 'react';
class SecondComponent extends Component {
 constructor(props) {
 super(props);
 this.state = {
 toggle: true
 };
```

// This is to bind context when passing onClick as a callback

```
this.onClick = this.onClick.bind(this);
}
onClick() {
this.setState((prevState, props) => ({
toggle: !prevState.toggle
}));
}
render() {
return (
<div onClick={this.onClick}>
Hello, {this.props.name}! I am a SecondComponent.
<br />
```

```
Toggle is: {this.state.toggle}
 </div>
 );
 }
}
```

- Extending a component with PureComponent instead of Component will automatically implement the shouldComponentUpdate() lifecycle method with shallow prop and state comparison.
- This keeps your application more performant by reducing the amount of un-necessary renders that occur.
- This assumes your components are 'Pure' and always render the same output with the same state and props input.

Higher Order Components

- Higher order components (HOC) allow to share component functionality.

```
import React, { Component } from 'react';
const PrintHello = ComposedComponent => class extends Component {
 onClick() {
 console.log('hello');
 }
/* The higher order component takes another component as a parameter
 and then renders it with additional props */
 render() {
 return <ComposedComponent {...this.props }
```

```
onClick={this.onClick} />
 }
}
const FirstComponent = props => (
 <div onClick={ props.onClick }>
 Hello, {props.name}! I am a FirstComponent.
 </div>
);
const ExtendedComponent = PrintHello(FirstComponent);
```

- Higher order components are used when you want to share logic across several components regardless of howdifferent they render.

Basic Component

Given the following HTML file:

index.html

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8" />
 <title>React Tutorial</title>
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/
react.js"></script>
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/
react-dom.js"></script>
```

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
 </head>
 <body>
 <div id="content"></div>
 <script type="text/babel"
src="scripts/example.js"></script>
 </body>
</html>
```

You can create a basic component using the following code in a separate file:

Nesting Components

A lot of the power of ReactJS is its ability to allow nesting of components. Take the following two components:

```
var React = require('react');
var createReactClass = require('create-react-class');
var CommentList = reactCreateClass({
 render: function() {
 return (
 <div className="commentList">
 Hello, world! I am a CommentList.
 </div>
 );
 }
});
```

```
var CommentForm = reactCreateClass({
 render: function() {
 return (
 <div className="commentForm">
 Hello, world! I am a CommentForm.
 </div>
 );
 }
});
```

- You can nest and refer to those components in the definition of a different component:

```
var React = require('react');
var createReactClass = require('create-react-class');
```

```
var CommentBox = reactCreateClass({
 render: function() {
 return (
 <div className="commentBox">
 <h1>Comments</h1>
 <CommentList /> // Which was defined above and can be reused
 <CommentForm /> // Same here
 </div>
 );
 }
});
```

- Further nesting can be done in three ways, which all have their own places to be used

Nesting without using children

(continued from above)

```
var CommentList = reactCreateClass({
 render: function() {
 return (
 <div className="commentList">
 <ListTitle/>
 Hello, world! I am a CommentList.
 </div>
 );
 }
});
```

- This is the style where A composes B and B composes C.

Pros

- Easy and fast to separate UI elements
- Easy to inject props down to children based on the parent component's state

Cons

- Less visibility into the composition architecture
- Less reusability

Good if

- B and C are just presentational components
- B should be responsible for C's lifecycle

Nesting using children

(continued from above)

```
var CommentBox = reactCreateClass({
 render: function() {
```

Uplatz

```
return (
 <div className="commentBox">
 <h1>Comments</h1>
 <CommentList>
 <ListTitle/> // child
 </CommentList>
 <CommentForm />
 </div>
 );
 }
});
```

- This is the style where A composes B and A tells B to compose C. More power to parent components.

Pros

- Better components lifecycle management
- Better visibility into the composition architecture
- Better reusuability

Cons

- Injecting props can become a little expensive
- Less flexibility and power in child components

Good if

- B should accept to compose something different than C in the future or somewhere else
- A should control the lifecycle of C

B would render C using this.props.children, and there isn't a structured way for B to know what those children are for.

- So, B may enrich the child components by giving additional props down, but if B needs to know exactly
- what they are, #3 might be a better option.

Nesting using props

(continued from above)

```
var CommentBox = reactCreateClass({
 render: function() {
 return (
 <div className="commentBox">
 <h1>Comments</h1>
 <CommentList title={ListTitle}/> //prop
 <CommentForm />
 </div>
 );
 }
```

});

This is the style where A composes B and B provides an option for A to pass something to compose for a specificpurpose.

More structured composition.

Pros

Composition as a feature

Easy validation

Better composaiblility

Cons

Injecting props can become a little expensive

Less flexibility and power in child components

Good if

B has specific features defined to compose something

- B should only know how to render not what to render
- #3 is usually a must for making a public library of components but also a good practice in general to make
- composable components and clearly define the composition features. #1 is the easiest and fastest to make something that works, but #2 and #3 should provide certain benefits in various use cases.

Props

- Props are a way to pass information into a React component, they can have any type including functions -
- sometimes referred to as callbacks.

In JSX props are passed with the attribute syntax

<MyComponent userID={123} />

Inside the definition for MyComponent userID will now be accessible from the props object

// The render function inside MyComponent

```
render() {
 return (
 <span>The user's ID is {this.props.userID}</span>
 )
}
```

It's important to define all props, their types, and where applicable, their default value:

// defined at the bottom of MyComponent

```
MyComponent.propTypes = {
 someObject: React.PropTypes.object,
```

```
userID: React.PropTypes.number.isRequired,
 title: React.PropTypes.string
};
MyComponent.defaultProps = {
 someObject: {},
 title: 'My Default Title'
}
```

- In this example the prop someObject is optional, but the prop userID is required.
- If you fail to provide userID to MyComponent, at runtime the React engine will show a console warning you that the required prop was not provided.
- Beware though, this warning is only shown in the development version of the React library

, theproduction version will not log any warnings.

Using defaultProps allows you to simplify

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

to

```
console.log(this.props.title);
```

- It's also a safeguard for use of object array and functions.
- If you do not provide a default prop for an object, thefollowing will throw an error if the prop is not passed:

```
if (this.props.someObject.someKey)
```

In example above, this.props.someObject is undefined and therefore the check of someKey will throw an error andthe code will break.

- With the use of defaultProps you can safely use the above check.

Component states - Dynamic user-interface

- Suppose we want to have the following behaviour - We have a heading (say h3 element) and on clicking it, we want it to become an input box so that we can modify heading name.
- React makes this highly simple and intuitive using
- component states and if else statements. (Code explanation below)

// I have used ReactBootstrap elements. But the code works with regular html elements also

var Button = ReactBootstrap.Button;

var Form = ReactBootstrap.Form;

```
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;
var Comment = reactCreateClass({
 getInitialState: function() {
 return {show: false, newTitle: ''};
 },
 handleTitleSubmit: function() {
 //code to handle input box submit - for example,
issue an ajax request to change name in
database
 },
 handleTitleChange: function(e) {
 //code to change the name in form input box.
```

newTitle is initialized as empty string.

- We need to update it with the string currently entered by user in the form

```
this.setState({newTitle: e.target.value});
},
changeComponent: function() {
// this toggles the show variable which is used for dynamic UI
this.setState({show: !this.state.show)};
},
render: function() {
var clickableTitle;
if(this.state.show) {
clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
```

```
<FormGroup controlId="formInlineTitle">
 <FormControl type="text"
onChange={this.handleTitleChange}>
 </FormGroup>
 </Form>;
 } else {
 clickabletitle = <div>
 <Button bsStyle="link"
onClick={this.changeComponent}>
 <h3> Default Text </h3>
 </Button>
 </div>;
 }
 return (
```

```
<div className="comment">
 {clickableTitle}
 </div>
 );
 }
});
ReactDOM.render(
 <Comment />, document.getElementById('content')
);
```

- The main part of the code is the clickableTitle variable. Based on the state variable show, it can be either be a
- Form element or a Button element.
- React allows nesting of components.

- So we can add a {clickableTitle} element in the render function. It looks for the clickableTitle variable.
- Based on the value 'this.state.show', it displays the corresponding element.

Variations of Stateless Functional Components

```
const languages = [
 'JavaScript',
 'Python',
 'Java',
 'Elm',
 'TypeScript',
 'C#',
 'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>
Language.propTypes = {
 message: React.PropTypes.string.isRequired
}
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
 <ul>
 {languages.map(language => <Language
language={language} />)}
 </ul>
```

```
}
LanguagesList.PropTypes = {
 languages: React.PropTypes.array.isRequired
}
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
```

```
const LanguageSection = ({header, languages}) => {
 // do some work
 const formattedLanguages =
languages.map(language =>
language.toUpperCase())
 return (
 <fieldset>
 <legend>{header}</legend>
 <LanguagesList languages={formattedLanguages} />
 </fieldset>
 )
}
LanguageSection.PropTypes = {
 header: React.PropTypes.string.isRequired,
```

```
 languages: React.PropTypes.array.isRequired
}
ReactDOM.render(
 <LanguageSection
header="Languages"
 languages={languages} />,
 document.getElementById('app')
)
```

setState pitfalls

You should use caution when using setState in an asynchronous context. For example, you might try to call

setState in the callback of a get request:

```
class MyClass extends React.Component {
 constructor() {
 super();
 this.state = {
 user: {}
 };
 }
 componentDidMount() {
 this.fetchUser();
 }
 fetchUser() {
 $.get('/api/users/self')
 .then((user) => {
 this.setState({user: user});
```

```
});
 }
 render() {
 return <h1>{this.state.user}</h1>;
 }
}
```

- This could call problems - if the callback is called after the Component is dismounted, then this.setState won't be afunction.
- Whenever this is the case, you should be careful to ensure your usage of setState is cancellable.
- In this example, you might wish to cancel the XHR request when the component dismounts:

```
class MyClass extends React.Component {
```

```
constructor() {
 super();
 this.state = {
 user: {},
 xhr: null
 };
 }
 componentWillUnmount() {
 let xhr = this.state.xhr;
 // Cancel the xhr request, so the callback is never
called
 if (xhr && xhr.readyState != 4) {
 xhr.abort();
 }
 }
```

```
componentDidMount() {
 this.fetchUser();
 }
 fetchUser() {
 let xhr = $.get('/api/users/self')
 .then((user) => {
 this.setState({user: user});
 });
 this.setState({xhr: xhr});
 }
}
```

- The async method is saved as a state. In the componentWillUnmount you perform all your cleanup – including canceling the XHR request.

- .You could also do something more complex.
- In this example, I'm creating a 'stateSetter' function that accepts the this object as an argument and prevents this.setState when the function cancel has been called:

```
function stateSetter(context) {
 var cancelled = false;
 return {
 cancel: function () {
 cancelled = true;
 },
 setState(newState) {
 if (!cancelled) {
 context.setState(newState);
 }
```

```
}
 }
}
class Component extends React.Component {
 constructor(props) {
 super(props);
 this.setter = stateSetter(this);
 this.state = {
 user: 'loading'
 };
 }
 componentWillUnmount() {
 this.setter.cancel();
 }
 componentDidMount() {
```

```
componentDidMount() {
 this.fetchUser();
 }
 fetchUser() {
 $.get('/api/users/self')
 .then((user) => {
 this.setter.setState({user: user});
 });
 }
 render() {
 return <h1>{this.state.user}</h1>
 }
}
```

- This works because the cancelled variable is visible in the setState closure we created.

Summary:

- Props
- State
- Hello World Component
- Creating Components
- Nesting Components
- Component states - Dynamic user-interface a
- Variations of Stateless Functional Components
- setState pitfalls

Uplatz

Thank You.....

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

Uplatz