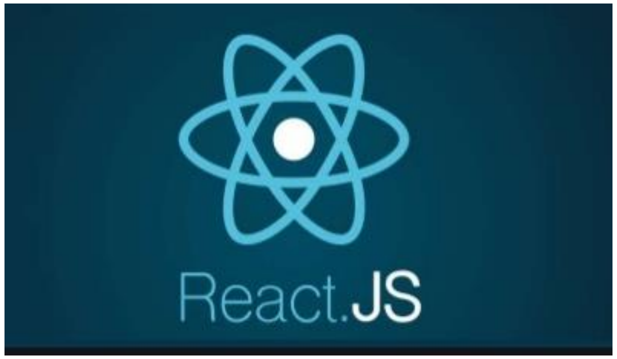

Presentation by Uplatz

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

Email: info@uplatz.com

Phone:+44 7836 212635

Table Of Contents:

Performance

Performance measurement with ReactJS

- You can't improve something you can't measure.
- To improve the performance of React components, you should be able to measure it.
- ReactJS provides with addon tools to measure performance.

Import the react-addons-perf

module to measure the performance

import Perf from 'react-addons-perf' // ES6

var Perf = require('react-addons-perf') // ES5 with npm

var Perf = React.addons.Perf; // ES5 with react-with-addons.js

- You can use below methods from the imported Perf module:

Perf.printInclusive()

Perf.printExclusive()

Perf.printWasted()

Perf.printOperations()

Perf.printDOM()

- The most important one which you will need most of the time is Perf.printWasted() which gives you the tabular representation of your individual component's wasted time

React's diff algorithm

- Generating the minimum number of operations to transform one tree into another have a complexity in the order of O(n^3) where n is the number of nodes in the tree.

React relies on two assumptions to solve this problem in a linear time - O(n)

1. Two components of the same class will generate similar trees and two components of different classes will generate different trees.

2. It is possible to provide a unique key for elements that is stable across different renders.

➢ In order to decide if two nodes are different, React differentiates 3 cases

1. Two nodes are different, if they have different types.

for example, <div>...</div> is different from <span>...</span>

2. Whenever two nodes have different keys

for example, <div key="1">...</div> is different from <div key="2">...</div>

- Moreover, what follows is crucial and extremely important to understand if you want to optimise performance
- If they [two nodes] are not of the same type, React is not going to even try at matching what they render.
- It is just going to remove the first one from the DOM and insert the second one.
- Here's why It is very unlikely that a element is going to generate a DOM that is going to look like what a would generate.
- Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

The Basics - HTML DOM vs Virtual DOM

HTML DOM is Expensive

- Each web page is represented internally as a tree of objects.
- This representation is called Document Object Model.
- Moreover, it is a language-neutral interface that allows programming languages (such as JavaScript) to access the

HTML elements.

In other words

- The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
- However, those DOM operations are extremely expensive.

Uplatz

Virtual DOM is a Solution

- So React's team came up with the idea to abstract the HTML DOM and create its own Virtual DOM in order to
- compute the minimum number of operations we need to apply on the HTML DOM to replicate current state of our application.

The Virtual DOM saves time from unnecessary DOM modifications.

How Exactly?

- At each point of time, React has the application state represented as a Virtual DOM.
- Whenever application state changes, these are the steps that React performs in order to optimise performance

- Generate a new Virtual DOM that represents the new state of our application
- 2. Compare the old Virtual DOM (which represents the current HTML DOM)
- to learn more about that - read React's Diff Algorithm
- 4. After those operations are found, they are mapped into their equivalent HTML DOM operations
- remember, the Virtual DOM is only an abstraction of the HTML DOM and there is a isomorphic relation

between them

- 5. Now the minimum number of operations that have been found and transferred to their equivalent HTML

Uplatz

- DOM operations are now applied directly onto the application's HTML DOM, which saves time from modifying the HTML DOM unnecessarily.
- **Note**: Operations applied on the Virtual DOM are cheap, because the Virtual DOM is a JavaScript Object.

Tips & Tricks

- When two nodes are not of the same type, React doesn't try to match them - it just removes the first node from the DOM and inserts the second one.
- This is why the first tip says

1. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class.

2. Use shouldComponentUpdate to prevent component from rerender, if you know it is not going to change, for example

```
shouldComponentUpdate: function(nextProps, nextState) {
 return nextProps.id !== this.props.id;
}
```

Testing the App

- The Timeline component dispays a list of statuses with a header with a dynamic title.
- We'll want to test any dynamic logic we have in our components.
- The simplest bit of logic we have to start out with our tests are around the dynamic title presented on the timeline.

- We like to start out testing by listing our assumptions about a component and under what circumstances these assumptions are true.
- For instance, a list of assumptions we can make about our Timeline component might include the following:
- Under all circumstances, the Timeline will be contained within a <div /> with the class of .notificationsFrame
- Under all circumstances, we can assume there will be a title Under all circumstances, we assume the search button will start out as hidden
- There is a list of at least four status updates
- These assumptions will translate into our tests.

Testing

Let's open the file src/components/Timeline/__tests__/Timeline-test.js. We left off with some dummy tests in this file, so let's clear those off and start with a fresh describe block:

```
describe("Timeline", () => {
  // Tests go here
});
```

- For every test that we write against React, we'll want to import react into our test file.
- We'll also want to bring in the react test utilities:

```
import React from "react";
import TestUtils from "react-dom/test-utils";
```

```
describe("Timeline", () => {
  // Tests go here
});
```

- Since we're testing the Timeline component here, we'll also want to bring that into our workspace:

```
import React from "react";
import TestUtils from "react-dom/test-utils";
import Timeline from "../Timeline";
describe("Timeline", () => {
  // Tests go here
});
```

- Let's write our first test. Our first assumption is pretty simple to test.

- We're testing to make sure the element is wrapped in a .notificationsFrame class.
- With every test we'll write, we'll need to render our application into the working test document.
- The react-dom/test-utils library provides a function to do just this called renderIntoDocument():

```
import React from "react";
import TestUtils from "react-dom/test-utils";
import Timeline from "../Timeline";
describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    const wrapper = TestUtils.renderIntoDocument(<Timeline />);
  }); });
```

- If we run this test (even though we're not setting any expectations yet), we'll see that we have a problem with the testing code. React thinks we're trying to render an undefined component:
- Let's find the element we expect to be in the DOM using another TestUtils function called findRenderedDOMComponentWithClass().
- The findRenderedDOMComponentWithClass() function accepts two arguments.
- The first is the render tree (our wrapper object) and the second is the CSS class name we want it to look for:

```
import React from "react";
import TestUtils from "react-dom/test-utils";
import Timeline from "../Timeline";
```

```
describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame
class", () => {
    const wrapper =
TestUtils.renderIntoDocument(<Timeline />);
    const node =
TestUtils.findRenderedDOMComponentWithClass(
      wrapper,
      "notificationsFrame"
    );
  });
});
```

- With that, our tests will pass (believe it or not). The TestUtils sets up an expectation that it can find the component with the .notificationsFrame class.

- If it doesn't find one, it will throw an error and our tests will fail.
- As a reminder, we can run our tests using either the npm test command or the yarn test command.
- We'll use the yarn test command for now since we're testing one component:

Using Enzyme

```
import React from "react";
import TestUtils from "react-dom/test-utils";
import Timeline from "../Timeline";
describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    const wrapper = TestUtils.renderIntoDocument(<Timeline />);
TestUtils.findRenderedDOMComponentWithClass(wrapper, "notificationsFrame");
  });
});
```

- Although this works, it's not quite the easiest test in the world to read.
- Let's see what this test looks like when we rewrite it with Enzyme.
- Rather than testing the complete component tree with Enzyme, we can test just the output of the component.
- Any of the component's children will not be rendered. This is called shallow rendering.
- Enzyme makes shallow rendering super easy.
- use the shallow function exported by Enzyme t mount our component.

Let's first configure enzyme use the adapter that makes it compatible with React version

Create src/setupTests.js and add the following:

```
import { configure } from "enzyme";
import Adapter from "enzyme-adapter-react-16";
configure({ adapter: new Adapter() });
```

Let's update the src/components/Timeline/__tests__/Timeline-test.js file to include the shallow function from enzyme:

```
import React from "react";
import { shallow } from "enzyme";
describe("Timeline", () => {
  it("wraps content in a div with .notificationsFrame class", () => {
    // our tests });
});
```

- Shallow rendering is supported by react-dom/test-utils as well.
- In fact, Enzyme just wraps this functionality. While we didn't use shallow rendering yesterday, if we were to use it would look like this:

```
const renderer = ReactTestUtils.createRenderer();
renderer.render(<Timeline />);
const result = renderer.getRenderOutput();
```

- Now to render our component, we can use the shallow method and store the result in a variable.
- Then, we'll query the rendered component for different React elements (HTML or child components) that are rendered inside its virtual dom.

The entire assertion comprises two lines:

```
import React from "react";
import { shallow, mount } from "enzyme";
import Timeline from "../Timeline";
describe("Timeline", () => {
  let wrapper;
it("wraps content in a div with .notificationsFrame
class", () => {
    wrapper = shallow(<Timeline />);
expect(wrapper.find(".notificationsFrame").length).toE
qual(1);
  });
});
```

- We can run our tests in the same manner as we did before using the yarn test command (or the npm test command):
- Our test passes and is more readable and maintainable.
- We'll structure the rest of our test suite first by writing out our describe and it blocks.
- We'll fill out the specs with assertions after:

```
import React from "react";
import { shallow } from "enzyme";
import Timeline from "../Timeline";
describe("Timeline", () => {
  let wrapper;
 it("wraps content in a div with .notificationsFrame
class", () => {
```

```
wrapper = shallow(<Timeline />);
expect(wrapper.find(".notificationsFrame").length).toEq
ual(1);
  });
it("has a title of Timeline");
describe("search button", () => {
    it("starts out hidden");
    it("becomes visible after being clicked on");
  });
 describe("status updates", () => {
    it("has 4 status updates at minimum");
  });
});
```

- ➢If we were following Test Driven Development (or TDD for short), we would write these assumptions first and then build the component to pass these tests.
- ➢Let's fill in these tests so that they pass against our existing Timeline component.
- ➢Our title test is relatively simple.
- ➢We expect the title to be available under a class of .title.
- ➢ So, to use the .title class in a spec, we can just grab the component using the find function exposed by Enzyme.
- ➢Since our Header component is a child component of our Timeline component, we can't use the shallow() method.
- ➢ Instead we have to use the mount() method provided by Enzyme.

Performance

Performance measurement with ReactJS

- You can't improve something you can't measure. To improve the performance of React components, you should
- be able to measure it. ReactJS provides with addon tools to measure performance. Import the react-addons-perf
- module to measure the performance

import Perf from 'react-addons-perf' // ES6

var Perf = require('react-addons-perf') // ES5 with npm

var Perf = React.addons.Perf; // ES5 with react-with-addons.js

- You can use below methods from the imported Perf module:

- **Perf.printInclusive()**
- **Perf.printExclusive()**
- **Perf.printWasted()**
- **Perf.printOperations()**
- **Perf.printDOM()**
- The most important one which you will need most of the time is Perf.printWasted() which gives you the tabular
- representation of your individual component's wasted time

React's diff algorithm

- Generating the minimum number of operations to transform one tree into another have a complexity in the order of O(n^3) where n is the number of nodes in the tree.

- React relies on two assumptions to solve this problem in a linear time - O(n)

1. Two components of the same class will generate similar trees and two components of different

classes will generate different trees.

2. It is possible to provide a unique key for elements that is stable across different renders.

In order to decide if two nodes are different, React differentiates 3 cases

1. Two nodes are different, if they have different types.

for example, <div>...</div> is different from <span>...</span>

- Whenever two nodes have different keys
- **for example, <div key="1">...</div> is different from <div key="2">...</div>**
- Moreover, what follows is crucial and extremely important to understand if you want to optimise performance
- If they [two nodes] are not of the same type, React is not going to even try at matching what they render.
- It is just going to remove the first one from the DOM and insert the second one.
- It is very unlikely that a element is going to generate a DOM that is going to look like what a would

generate.

- Instead of spending time trying to match those two structures, React just re-builds the tree from scr

Uplatz

The Basics - HTML DOM vs Virtual DOM

HTML DOM is Expensive

- Each web page is represented internally as a tree of objects. This representation is called Document Object Model.
- Moreover, it is a language-neutral interface that allows programming languages (such as JavaScript) to access the

HTML elements.

In other words

- The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
- However, those DOM operations are extremely expensive.

Virtual DOM is a Solution

- So React's team came up with the idea to abstract the HTML DOM and create its own Virtual DOM in order to compute the minimum number of operations we need to apply on the HTML DOM to replicate current state of our application.
- The Virtual DOM saves time from unnecessary DOM modifications.

How Exactly?

- At each point of time, React has the application state represented as a Virtual DOM.
- Whenever application state changes, these are the steps that React performs in order to optimise performance
- 1. Generate a new Virtual DOM that represents the new state of our application

- . Compare the old Virtual DOM (which represents the current HTML DOM) vs the new Virtual DOM
- 3. Based on 2. find the minimum number of operations to transform the old Virtual DOM (which represents the
- current HTML DOM) into the new Virtual DOM

to learn more about that - read React's Diff Algorithm

- 4. After those operations are found, they are mapped into their equivalent HTML DOM operations

remember, the Virtual DOM is only an abstraction of the HTML DOM and there is a isomorphic relation between them

5. Now the minimum number of operations that have been found and transferred to their equivalent HTML

- DOM operations are now applied directly onto the application's HTML DOM, which saves time from modifying the HTML DOM unnecessarily.
- Note: Operations applied on the Virtual DOM are cheap, because the Virtual DOM is a JavaScript Object.

Tips & Tricks

- When two nodes are not of the same type, React doesn't try to match them - it just removes the first node from theDOM and inserts the second one.
- This is why the first tip says
- 1. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class.

➢. Use shouldComponentUpdate to prevent component from rerender, if you know it is not going to change,

for example

shouldComponentUpdate: function(nextProps, nextState) {

return nextProps.id !== this.props.id;

}

Introduction to Server-Side Rendering:

Rendering components:

➢There are two options to render components on server: renderToString and renderToStaticMarkup.

renderToString

➢This will render React components to HTML on server. This function will also add data-react- properties t
HTML

➢elements so React on client won't have to render elements again.

import { renderToString } from "react-dom/server";

renderToString(<App />);

renderToStaticMarkup

➢This will render React components to HTML, but without data-react- properties, it is not recommended to use

➢components that will be rendered on client, because components will rerender.

import { renderToStaticMarkup } from "react-dom/server";

renderToStaticMarkup(<App />);

Setting Up React Environment

Simple React Component

➢We want to be able to compile below component and render it in our webpage

Filename: src/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
class ToDo extends React.Component {
 render() {
 return (<div>I am working</div>);
 }
}
ReactDOM.render(<ToDo />,
document.getElementById('App'));
```

Uplatz

Install all dependencies

```
# install react and react-dom
$ npm i react react-dom --save
# install webpack for bundling
$ npm i webpack -g
# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save
# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Configure webpack

➢Create a file webpack.config.js in the root of your working directory

Filename: webpack.config.js

```
module.exports = {
 entry: __dirname + "/src/index.jsx",
 devtool: "source-map",
 output: {
 path: __dirname + "/build",
 filename: "bundle.js"
 },
 module: {
 loaders: [
 {test: /\.jsx?$/, exclude: /node_modules/, loader: 
"babel-loader"}
 ]
 }
}
```

Configure babel

Create a file .babelrc in the root of our working directory

Filename: .babelrc

```
{
 "presets": ["es2015","react"]
}
```

HTML file to use react component

➢Setup a simple html file in the root of the project directory

Filename: index.html

```
<!DOCTYPE html>
<html>
 <head>
```

```
<meta charset="utf-8">
 <title></title>
 </head>
 <body>
 <div id="App"></div>
 <script src="build/bundle.js" charset="utf-8"></script>
 </body>
</html>
```

- Transpile and bundle your component
- Using webpack, you can bundle your component:

```
$ webpack
```

- This will create our output file in build directory.
- Open the HTML page in a browser to see component in action

➢Using React with Flow

➢How to use the Flow type checker to check types in React components.

➢ Using Flow to check prop types of stateless **functional components**

```
type Props = {
 posts: Array<Article>,
 dispatch: Function,
 children: ReactElement
}
const AppContainer =
 ({ posts, dispatch, children }: Props) => (
 <div className="main-app">
 <Header {...{ posts, dispatch }} />
```

{children}
</div>
)

Using Flow to check prop types

```
import React, { Component } from 'react';
type Props = {
 posts: Array<Article>,
 dispatch: Function,
 children: ReactElement
}
class Posts extends Component {
 props: Props;
 render () {
 // rest of the code goes here
 } }
```

JSX

Props in JSX

- There are several different ways to specify props in JSX.

JavaScript Expressions

- You can pass any JavaScript expression as a prop, by surrounding it with {}. For example, in this JSX:

<MyComponent count={1 + 2 + 3 + 4} />

- Inside the MyComponent, the value of props.count will be 10, because the expression 1 + 2 + 3 + 4 gets evaluated.
- If statements and for loops are not expressions in JavaScript, so they can't be used in JSX directly.

String Literals

- Of course, you can just pass any string literal as a prop too. These two JSX expressions are equivalent:

<MyComponent message="hello world" />

<MyComponent message={'hello world'} />

- When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />

- This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default Value

- If you pass no value for a prop, it defaults to true. These two JSX expressions are equivalent:

<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />

- However, the React team says in their docs using this approach is not recommended, because it can be confused
- with the ES6 object shorthand {foo} which is short for {foo: foo} rather than {foo: true}. They say this behavior
- is just there so that it matches the behavior of HTML.

Spread Attributes

- If you already have props as an object, and you want to pass it in JSX, you can use ... as a spread operator to pass
- the whole props object. These two components are equivalent:

```
function Case1() {
 return <Greeting firstName="Kaloyab"
    lastName="Kosev" />;
function Case2() {
 const person = {firstName: 'Kaloyan', lastName:
'Kosev'};
 return <Greeting {...person} />;
}
```

Children in JSX

➢In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as aspecial prop: props.children.

➢ There are several different ways to pass children:

➢String Literals

➢You can put a string between the opening and closing tags and props.children will just be that string.

➢This is useful for many of the built-in HTML elements. For example:

<MyComponent>

 <h1>Hello world!</h1>

</MyComponent>

➢This is valid JSX, and props.children in MyComponent will simply be <h1>Hello world!</h1>.

➢Note that the HTML is unescaped, so you can generally write JSX just like you would write HTML.

removes whitespace at the beginning and ending of a line; removes blank lines;

- new lines adjacent to tags are removed;
- new lines that occur in the middle of string literals are condensed into a single space.

JSX Children

- You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
 <MyFirstComponent />
 <MySecondComponent />
</MyContainer>
```

- You can mix together different types of children, so you can use string literals together with JSX children.
- This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
 <h2>Here is a list</h2>
 <ul>
 <li>Item 1</li>
 <li>Item 2</li>
 </ul>
</div>
```

- Note that a React component can't return multiple React elements, but a single JSX expression can have
- multiple children.
- So if you want a component to render multiple things you can wrap them in a div like the
- example above.

JavaScript Expressions

➢You can pass any JavaScript expression as children, by enclosing it within {}.

➢ For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

➢This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
const Item = ({ message }) => (
 <li>{ message }</li>
);
const TodoList = () => {
 const todos = ['finish doc', 'submit review', 'wait
stackoverflow review'];
return (
 <ul>
 { todos.map(message => (<Item key={message}
message={message} />)) }
 </ul>
 );
};
```

➢Note that JavaScript expressions can be mixed with other types of children.

Functions as Children

➢Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things.

➢However, props.children works just like any other prop in that it can pass any sort of data, not just the sc
that

➢React knows how to render. For example, if you have a custom component, you could have it take a callback as props.children:

```
const ListOfTenThings = () => (
 <Repeat numTimes={10}>
 {(index) => <div key={index}>This is item {index} in the list</div>}
 </Repeat>
);
// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
 let items = [];
 for (let i = 0; i < numTimes; i++) {
 items.push(children(i));
}
 return <div>{items}</div>;
};
```

➢Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering.

➢ This usage is not common, but it works if you want to stretch what JSX is capable of.

Ignored Values

➢Note that false, null, undefined, and true are valid children. But they simply don't render. These JSX expressions

will all render to the same thing:

```
<MyComponent />
<MyComponent></MyComponent>
<MyComponent>{false}</MyComponent>
<MyComponent>{null}</MyComponent>
<MyComponent>{true}</MyComponent>
```

➢This is extremely useful to conditionally render React elements. This JSX only renders a if showHeader is true:

```
<div>
 {showHeader && <Header />}
 <Content />
</div>
```

➢One important caveat is that some "falsy" values, such as the 0 number, are still rendered by React. For example,

- this code will not behave as you might expect because 0 will be printed when props.messages is an empty array:

```
<div>
 {props.messages.length &&
 <MessageList messages={props.messages} />
 }
</div>
```

- One approach to fix this is to make sure that the expression before the && is always boolean:

```
<div>
 {props.messages.length > 0 &&
 <MessageList messages={props.messages} />
 }
</div>
```

Summary:

- Performance
- Testing the App
- Using Enzyme
- Setting Up React Environment
- JavaScript Expressions

Thank You………

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