REACT DEVELOPMENT

What is React?

- A declarative, efficient, and flexible JavaScript library for building user interfaces.
- One-way data flow
  - Properties, a set of immutable values, are passed to a component's renderer as properties in its HTML tag. A component cannot directly modify any properties passed to it, but can be passed callback functions that do modify values.
- Virtual DOM
  - React creates an in-memory data structure cache, computes the resulting differences, and then updates the browser's displayed DOM efficiently. This allows to write code as if the entire page is rendered on each change.
- JSX
  - React components are typically written in JSX, a JavaScript extension syntax allowing quoting of HTML and using HTML tag syntax to render subcomponents.

Environment Prerequisites

- Node.Js >= 6.10.3
- NPM package manager
- Webpack module bundler

Environment – Installing Webpack

- Webpack is a module bundler which takes modules with dependencies and generates static assets by bundling them together based on some configuration.
- The support of loaders in Webpack makes it a perfect fit for using it along with React and we will discuss it later in this post with more details.
- Webpack installation is done using NPM.

Environment – Installing Webpack

- Let's start by initiating an npm environment:

```
npm init
```

- Install required webpack using npm:

```
npm install webpack babel-loader babel-preset-es2015 babel-preset-react --save
```

- Now we have the required modules to create a working webpack bundler for a React environment.
- But, how do the pieces glue together? Check out the next few slides for the answer!

Environment – Installing Webpack

- Webpack requires a config file to be present in the root folder of your project run:

```
var webpack = require('webpack');
var path = require('path');

var BUILD_DIR = path.resolve(__dirname, 'build/');
var APP_DIR = path.resolve(__dirname, 'src/');

var config = {
    entry: APP_DIR + '/index.jsx',
    output: {
        path: BUILD_DIR,
        filename: 'bundle.js',
        module: {
            loaders: [
                {
                    test: /\.jsx?/,
                    include: APP_DIR,
                    loader: 'babel'
                }
            ]
        }
    }
};

module.exports = config;
```

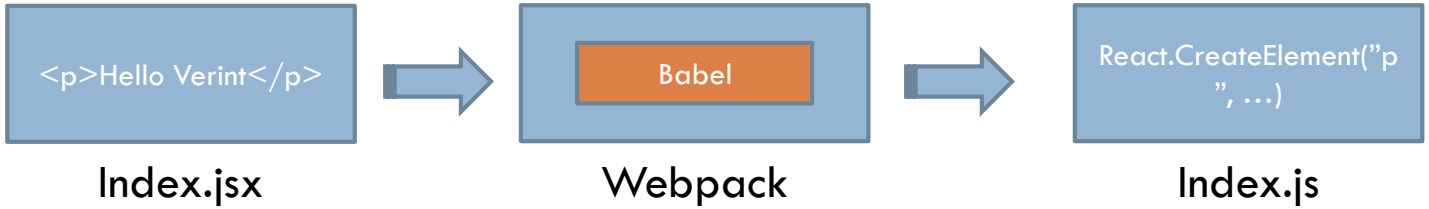
BUILD_DIR represents the directory path of the bundle file output.

The APP_DIR holds the directory path of the codebase

Tell's webpack which modules it should use in the build process

Environment – Setting up Babel

- JSX & ES6 syntax is not supported in most browsers by default. We need a tool which translates them to a format that is supported by the browsers. Babel is a tool that will do just that.



- Babel required a config file (.babelrc) to be present in the root folder of the project:

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

Environment – Gluing the Pieces

- A React project that is built using webpack should look somewhat familiar to the following structure:

```
my-react-project/
├── .babelrc
├── package.json
├── webpack.config.js
├── src/
├── build/
```

- .babelrc
  - Babel transformer configuration.
- package.json
  - NPM configuration.
- webpack.config.js
  - Webpack build tool configuration.

Environment – Gluing the Pieces

- Create An index.html file in the build folder:

```
<html>
<head>
   <title>React Hello World</title>
</head>
<body>
   <div id="app"/>
   <script src="bundle.js" type="text/javascript"></script>
</body>
</html>
```

- Notice the div with the id “app”? It is the element which our app will hook to.
- bundle.js.. Eh? It is the output of the webpack build process. It contains the transformed code of the app.

Environment – Gluing the Pieces

- In-order to create a react app JSX entry point we need the react module to be present in our project:

```
npm install react react-dom --save
```

- Create an index.jsx file in the src folder:

```
import React from 'react';
import {render} from 'react-dom';

class HelloWorldApp extends React.Component {
    render() {
        return <p>Hello world!</p>
    }
}

render(HelloWorldApp, document.getElementById('app'));
```

- The following code represents a React component and a render function.

Environment – Gluing the Pieces

- After configuring the project, building is just a command away:

```
./node_modules/.bin/webpack -d --watch
```

- When the build process is done, the project's file structure should look somewhat similar to the following:

```
my-react-project/
├── .babelrc
├── package.json
├── webpack.config.js
├── src/
├── ├── index.jsx
├── build/
├── ├── index.html
├── ├── bundle.js
```

REACT COMPONENTS

Introduction

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.
- There are two main types of compoenets.
  - Stateful component – a component that hold's an internal state.
  - Pure component – a component that has no inner state but may receive data from it's props.

Creating a Component

- The simplest way to define a component is to write a JavaScript function:

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

- This function is a valid React component because it accepts a single "props" object argument with data and returns a React element.
- You can also use an ES6 class to define a component:

```
class HelloWorld extends React.Component {
  render() {
    return (<div>Hello world!</div>);
  }
}
```

- The above two components are equivalent from React's point of view.

Rendering a Component

- When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".
- For example, this code renders "Hello, Itay" on the page:

```
const HelloWorld = (props) => (
    <div>Hello, {props.name}</div>
);

const element = <HelloWorld name="Itay"/>;

ReactDOM.render(element, document.body);
```

- But, what is JSX Really? Find out on the next few slides.

Lifecycle hooks

- componentWillMount()
  - Invoked immediately before mounting occurs. It is called before render().
- componentDidMount()
  - Invoked immediately after a component is mounted. Initialization that requires DOM nodes should go here.
- shouldComponentUpdate()
  - Use shouldComponentUpdate() to let React know if a component's output is not affected by the current change in state or props.
- componentWillUpdate()
  - Invoked immediately before rendering when new props or state are being received.
- componentDidUpdate()
  - Invoked immediately after updating occurs.
- componentWillReceiveProps()
  - Invoked before a mounted component receives new props.
- componentWillUnmount()
  - invoked immediately before a component is unmounted and destroyed.

Introduction to JSX

- JSX is a syntax extension to JavaScript. It is recommend to be used with React to describe what the UI should look like.
- JSX may remind you of a template language, but it comes with the full power of JavaScript. JSX produces React "elements".
- You can embed any JS expression in JSX by wrapping it in curly braces:

```
function sayHello(name) {
    return 'Hello, ' + name;
}

const name = 'Itay';

const element = (
    <h1>{sayHello(name)}</h1>
);

ReactDOM.render(element, document.body);
```

Introduction to JSX

- After compilation, JSX expressions become regular JavaScript objects. This means that you can use JSX inside of an if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function GetGreeting(name) {
   if(name === 'Itay') {
      return <h1>Hello, Itay!</h1>;
   }
   return <h1>Hello, Stranger.</h1>;
}

const element = <GetGreeting name="Itay"/>;

ReactDOM.render(element, document.body);
```

Introduction to JSX

- JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello World!</h1>
    <p>How are you today?</p>
  </div>
);
```

- If a tag is empty, you may close it immediately with />:

```
const element = <img src={user.avatarUrl} />;
```

- Babel compiles JSX down to React.createElement() calls.
- The following examples are identical:

```
const element = (<h1 className="greeting">Hello, world!</h1>);

const element = React.createElement('h1', {className: 'greeting'}, 'Hello, world!');
```

JSX Pitfalls

- React Must Be in Scope:
  - JSX compiles into calls to React.createElement, the React library must also always be in scope.
- User-Defined Components Must Be Capitalized.
- Booleans, Null, and Undefined Are Ignored.
- There are several different ways to specify props in JSX:
  - You can pass any JavaScript expression as a prop, by surrounding it with {}.
  - You can pass a string literal as a prop.
  - If you pass no value for a prop, it defaults to true.
  - If you already have props as an object, and you want to pass it in JSX, you can use the spread operator to pass the whole props object.

Component – Props

- Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a, b) {
  return a + b;
}
```

- Such functions are called pure because they do not attempt to change their inputs, and always return the same result for the same inputs.
- In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

- React is pretty flexible but it has a single strict rule:
  - **All React components must act like pure functions with respect to their props.**

Component – Props Defaults

- You can setup default values for props which will be used on the component load. This code renders “Hello Itay”:

```
const HelloWorld = (props) => (
    <div>Hello, {props.name}</div>
);

HelloWorld.propTypes = {
    name: React.PropTypes.string
};

HelloWorld.defaultProps = {
    name: 'Itay'
};


const element = <HelloWorld />;

ReactDOM.render(element, document.body);
```

Component – State

- A component may or may not be stateful. The state is encapsulated inside a component and cannot be accessed directly from the outside but can be passed down to child components using props.

```
class Counter extends React.Component {
    constructor(props) {
        super(props);

        this.state = {
            counter: 0
        }
    }

    render() {
        return (
            <div>Counter: {this.state.counter}</div>
        );
    }
}

React.renderComponent(<Counter/>, document.body);
```

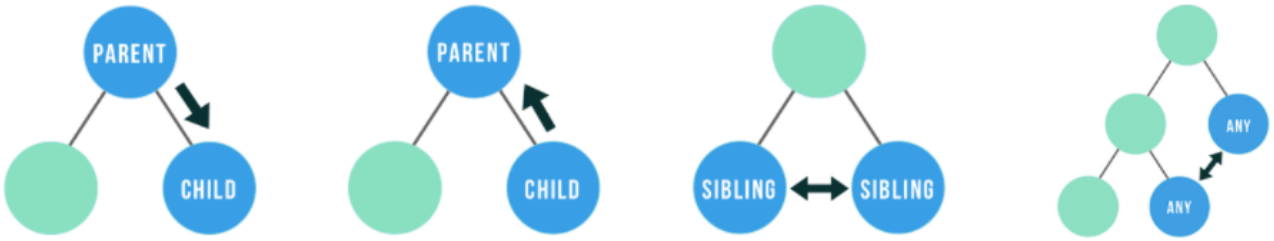
Component – Composing

- Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.
- If a component renders another component in it's render method, the renderer is the owner of the rendered component.
- The renderer owns the rendered component and has control over it. aka parenting.

```
const HelloWorldDisplay = (props) => (
    <div>Hello World!</div>
);

const App = () => (
    <div><HelloWorldDisplay/></div>
)

const element = <App />;

ReactDOM.render(element, document.body);
```

Interaction

- What if we had a button in a child component that needs to manipulates the state managed in a parent component?
- React component interaction comes in the form of data flow from **parent to child, child to parent and sibling to sibling.**
- Usually, component interacts with it's parent and siblings using callbacks (handlers) that are passed using props.



Parent to Child

1. Props
2. Instance Methods

Child to Parent

3. Callback Functions
4. Event Bubbling

Sibling to Sibling

5. Parent Component

Any to Any

6. Observer Pattern
7. Global Variables
8. Context

Interaction – Parent to Child

- Parent to child interaction is probably the simplest case as we just pass down props to the child:

```
const ChildComponent = (props) => (
    <div>
        <button>{props.buttonText}</button>
    </div>
);

class ParentComponent extends React.Component {
    render() {
        return (
            <div>
                <ChildComponent buttonText="Click Me!"/>
            </div>
        );
    }
}
```

Interaction – Child to Parent

- Child to parent interaction will again, in most cases happen over props. It is common to pass down callbacks through props:

```
const ChildComponent = (props) => (
    // Bind to the onClick event of button and invoke onClick callback when fired
    <div>
        <button onClick={props.onClick}>Click me</button>
    </div>
);

class ParentComponent extends React.Component {
    // Callback function that will be passed down to the child component
    handleChildClick(data) {
        console.log('Child component has returned with data:', data);
    }

    render() {
        // Pass down the callback function through the onClick prop
        return (
            <div>
                <ChildComponent onClick={this.handleChildClick}/>
            </div>
        );
    }
}
```

Interaction – Sibling to Sibling

- Sibling to sibling interaction is not very common. It requires us to keep a state in the parent component for each child component and pass down callbacks.
- The solution is complicated. I will present it hands-on.

Interaction – Any to Any

- It is an anti pattern.
- If you reach a state where you want all the component's in your app to communicate with each other, your are doing something very very wrong..

React Context (Experimental)

- Allows you to pass data down through the component hierarchy without the usage of props.
- It is experimental. React's documentation warn us about it:

> Context is an advanced and experimental feature. The API is likely to change in future releases.

- A context provider component is required. It should implement both childContextTypes and getChildContext:
  - *childContextTypes* - static property that allows you to declare the structure of the context object being passed to your component's descendants.
  - *getChildContext* - prototype method that returns the context object to pass down the component's hierarchy. Every time the component renders, this method will be called.

React Context (Experimental)

- The following code will show the interaction between a provider component and a consumer component:

```
class Provider extends React.Component {
   static childContextTypes = {
      versionId: React.PropTypes.string
   };

   getChildContext() {
      return {versionId: '1.0.0'};
   }

   render() {
      return (...);
   }
}

const Consumer = (props, context) => (
   <p>{context.versionId}</p>
);

ContextConsumer.contextTypes = {
   context: React.PropTypes.string
};
```

- Note that The consumer needed to declare what it want's from the context via *contextTypes*

Component Ref

- React supports a special attribute that you can attach to any component. The ref attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted.
- When the ref attribute is used on an HTML element, the ref callback receives the underlying DOM element as its argument.
- Use at your own risk. Try and accomplish most of your data flow needs using props.

```
class CustomTextInput extends React.Component {
  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <div>
        <input type="text" ref={(input) => { this.myRef = input; }} />
      </div>
    );
  }
}
```

REACT VIRTUAL DOM

DOM

- DOM stands for *Document Object Model* and is an abstraction of a structured text. For web developers, this text is an HTML code. *Elements* of HTML become *nodes* in the DOM. So, while HTML is a text, the DOM is an in-memory representation of this text.
- The HTML DOM provides an interface (API) to traverse and modify the nodes:

```
const item = document.getElementById("li");
item.parentNode.removeChild(item);
```

DOM - Issues

- The HTML DOM is always tree-structured. Traversing trees is fairly easily. Unfortunately, easily doesn't mean quickly here.
- The DOM trees are huge nowadays as we push more and more towards dynamic web apps (SPA etc..) which may cause performance and maintenance issues.
- Consider the following: a DOM tree is made of thousands of elements with lots of methods that handle events - clicks, submits etc.. A typical jQuery-like event handler looks like this:
  - Find every DOM node that is interested in an event.
  - Update node if necessary.
- The following has two main issues, It's hard to manage and It's inefficient.

Virtual Dom

The paradigm was not invented by React but React uses it and provides it for free.

- PROS
  - Is an abstraction of the DOM.
  - Fast and efficient "diffing" algorithm.
  - Multiple frontends (JSX, hyperscript).
  - Detached from browser-specific implementations.
  - Can be used without React (i.e. as an independent engine).
- CONS
  - Full in-memory copy of the DOM (higher memory use).

Virtual Dom

- The virtual DOM is pretty similar to the “regular” DOM.
- In most cases, when you have an HTML code, it’s pretty straight forward to make it into a static React component:
- There are more, rather minor, differences between the DOMs:
  - key, ref and dangerouslySetInnerHTML do not exist in “real” DOM.
  - You can read about the differences on the React website.

ReactElement

- ReactElement is the primary type in React.
- ReactElement is light, stateless, immutable, virtual representation of a DOM Element.
- ReactElements live in the virtual DOM.
- Almost every HTML tag can be a ReactElement (div, p, etc..).
- Once defined, ReactElements can be rendered into the “real” DOM. The moment when React ceases to control the elements. They become slow, boring DOM nodes:

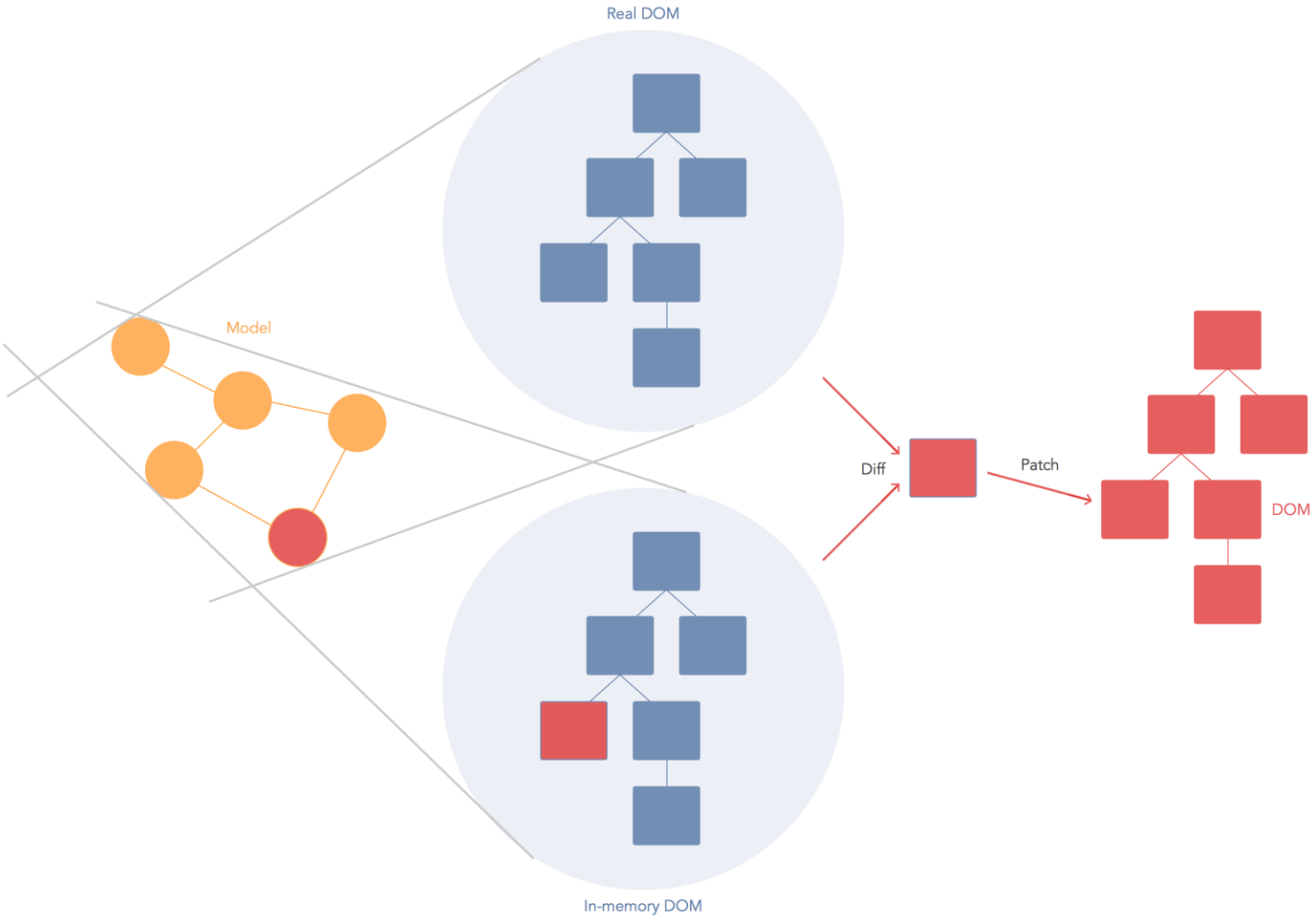
```
var root = React.createElement('div');
ReactDOM.render(root, document.getElementById('app'));
// JSX Equivalent
var root = <div />;
ReactDOM.render(root, document.getElementById('app'));
```

ReactComponent

- *ReactComponents are stateful.*
- ReactComponents don't have the access to the virtual DOM, but they can be easily converted to ReactElements:

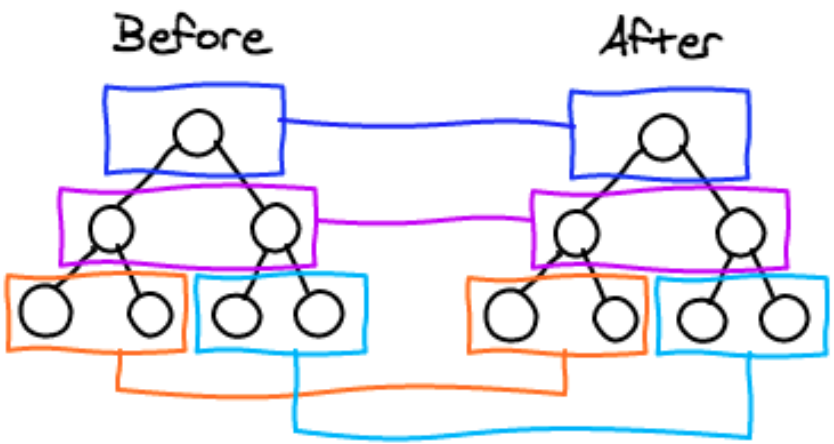
```
var element = React.createElement(MyComponent);
// or equivalently, with JSX
var element = <MyComponent />;
```

Reconciliation Process



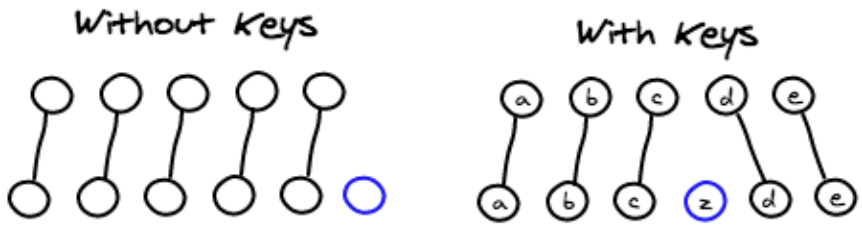
The Diff Algorithm – Level by level

- Finding the minimal number of modifications between two arbitrary trees is a O(n^3) problem. React uses simple and yet powerful heuristics to find a very good approximation in O(n).
- React reconcile trees level by level. This drastically reduces the complexity and isn't a big loss as it is very rare in web applications to have a component being moved to a different level in the tree. They usually only move laterally among children.



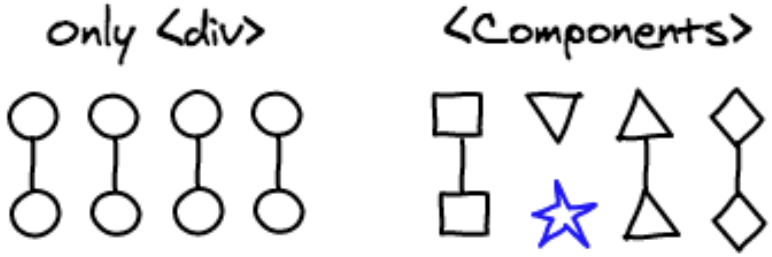
The Diff Algorithm – List

- Let say that we have a component that on one iteration renders 5 components and the next inserts a new component in the middle of the list. This would be really hard with just this information to know how to do the mapping between the two lists of components. By default, React associates the first component of the previous list with the first component of the next list, etc. You can provide a key attribute in order to help React figure out the mapping. In practice, this is usually easy to find out a unique key among the children.



The Diff Algorithm – Components

- A React app is usually composed of many user defined components that eventually turns into a tree composed mainly of divs. This additional information is being taken into account by the diff algorithm as React will match only components with the same class. For example, if a is replaced by an <fooBlock>, React will remove the div and create a foo block. We don't need to spend precious time trying to match two components that are unlikely to have any resemblance.

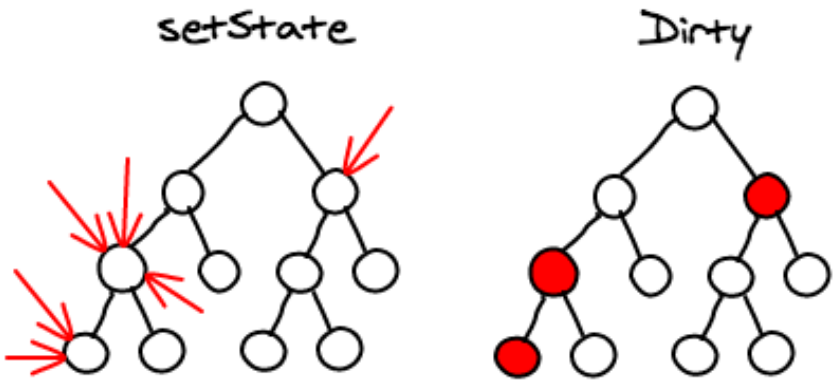


Reconcile - Event Delegation

- Attaching event listeners to DOM nodes is painfully slow and memory-consuming. Instead, React implements a popular technique called “event delegation”. React goes even further and re-implements a W3C compliant event system.
- it’s implemented as follows: A single event listener is attached to the root of the document. When an event is fired, the browser gives us the target DOM node. In order to propagate the event through the DOM hierarchy, React doesn’t iterate on the virtual DOM hierarchy. Instead, we use the fact that every React component has a unique id that encodes the hierarchy. We can use simple string manipulation to get the id of all the parents. By storing the event listeners in a hash map, we found that it performed better than attaching them to the virtual DOM.

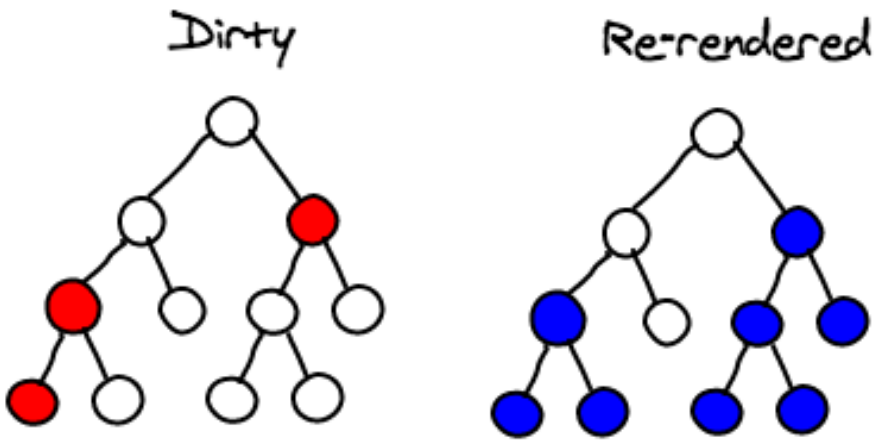
Reconcile - Rendering

- Whenever you set a state on a component, React will mark it as dirty. At the end of the event loop, React looks at all the dirty components and re-renders them. This batching means that during an event loop, there is exactly one time when the DOM is being updated. This property is key to building a performant app and yet is extremely difficult to obtain using commonly written JavaScript. In a React application, you get it by default.



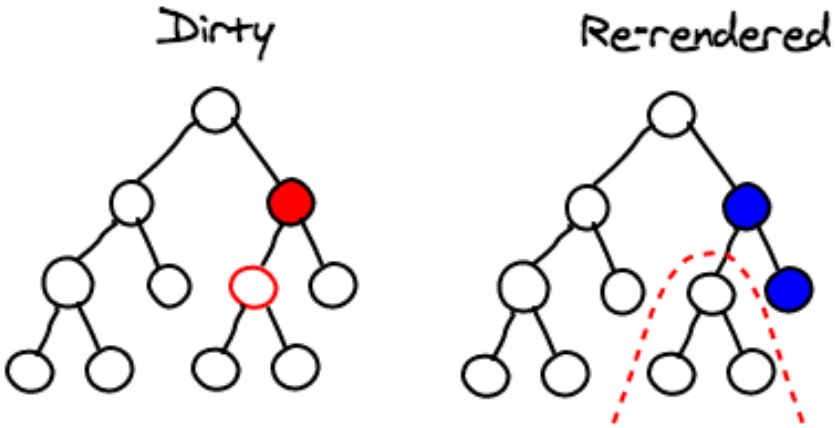
Reconcile - Sub-tree Rendering

- When setState function is called, the component rebuilds the virtual DOM for its children. If you call setState on the root element, then the entire React app is re-rendered. All the components, even if they didn't change, will have their render method called. This may sound scary and inefficient but in practice, this works fine because we're not touching the actual DOM.



Reconcile - Selective Sub-tree Rendering

- It is possible to prevent some sub-trees to re-render. If you implement the shouldComponentUpdate(nextProps, nextState) method on a component.
- Keep in mind that this function is going to be called all the time, so you want to make sure that it takes less time to compute than heuristic than the time it would have taken to render the component, even if re-rendering was not strictly needed.



Conclusion

- The techniques that make React fast are not new.
- We want to make as little changes to the “real” DOM.
- In practice, DOM optimizations are very hard to implement in regular JavaScript code. React gives you the optimizations ”On the house”.
- Simple performance cost model: every setState re-renders the whole sub-tree. If you want to squeeze out performance, call setState as low as possible and use shouldComponentUpdate to prevent re-rendering an large sub-tree.