REACT

CORE CONCEPTS

PART2

NOVEMBER 15, 2016

AGENDA

1. React Terminology
2. Virtual DOM
3. Reconciliation
4. Optimizing Performance

React Terminology

React Element

React Element is an immutable description object

JSX

```
<div className = "parent" id = "id_001" >
      <div className = "child">Some text</div>
</div>
```

React Element

```
{
      type : "div", //string
      props : {
            className : "parent“,
            id : "id_001",
            children : {
                  type : "div", //string
                  props : {
                        className : "child"
                        children : "Some text"
                  }
            }
      }
}
```

Component

Components Encapsulate React Elements

```
class Timer extends React.Component {
        constructor(props){
                super(props)
                this.state = {date: new Date()}
        }
        handleClick(){
                this.setState({date: new Date()})
        }
        render(){
                return (
                        <div>
                                <h1>{this.props.title}</h1>
                                <h2>It is {this.state.date.toLocaleTimeString()}</h2>
                                <button onClick = {this.handleClick.bind(this)}>Refresh Time</button>
                        </div>
                );
        }
}
ReactDOM.render(<Timer title="My Timer"/>, document.getElementById("container"));
```

Component Rendering

React Element from a component

An element describing a *component* **is also an element**

An element describing a *component* **is not a component instance**

JSX

```
<div className = "parent" id = "id_001" >
      <Timer title = "My Timer" />
</div>
```

React Element

```
{
      type : "div", //string
      props : {
            className : "parent“,
            id : "id_001",
            children : {
                  // type is a class!
                  type : function Timer(props){//…},
                  props : {
                        title: “My Timer"
                  }
            }
      }
}
```

Component Instance

An *instance* is what you refer to as `this` in the component class you write.

It is useful for storing local state and reacting to the lifecycle events.

Only components declared as classes have instances, and **you never create them directly**: React does that for you.

Virtual DOM

Virtual DOM

Virtual DOM is a JavaScript representation of the DOM

Virtual DOM

Virtual DOM is a JavaScript representation of the DOM

React Element is the main building block of Virtual DOM

Virtual DOM

Virtual DOM is a JavaScript representation of the DOM

React Element is the main building block of Virtual DOM

Component Instance is also part of Virtual DOM

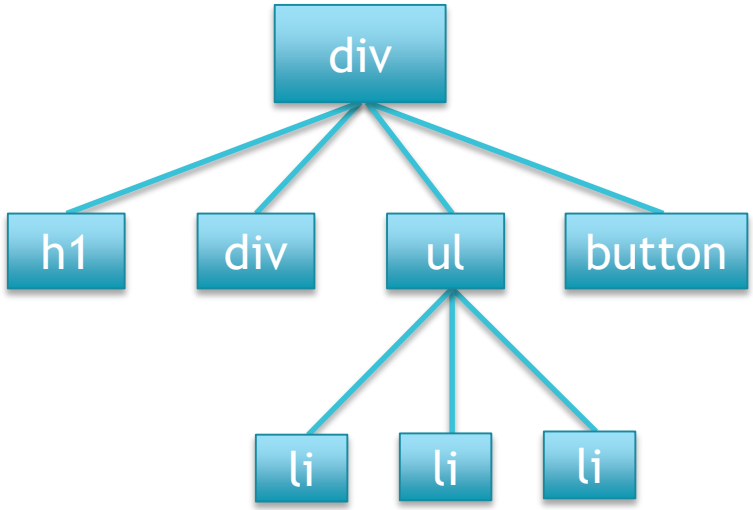
Comment List example

```
class CommentList extends React.Component {
        constructor(props){
                super(props);
                this.state = {data: ["comment1", "comment2", "comment3"]}
        }
        handleClick(){
                 let newData = [“NEW_comment1", "comment2", "comment3", "comment4"]
                 this.setState({data: newData })
        }
        render(){
                return (
                <div>
                        <h1>{this.props.title}</h1>
                        <div>Some complex DIV</div>
                        <ul>{this.state.data.map((elem)=><li>{elem}</li>)}</ul>
                        <button onClick={this.handleClick.bind(this)}>Resfresh</button>
                </div>
                );
        }
}
ReactDOM.render(<CommentList title="My List"/>, document.getElementById("container"));
```

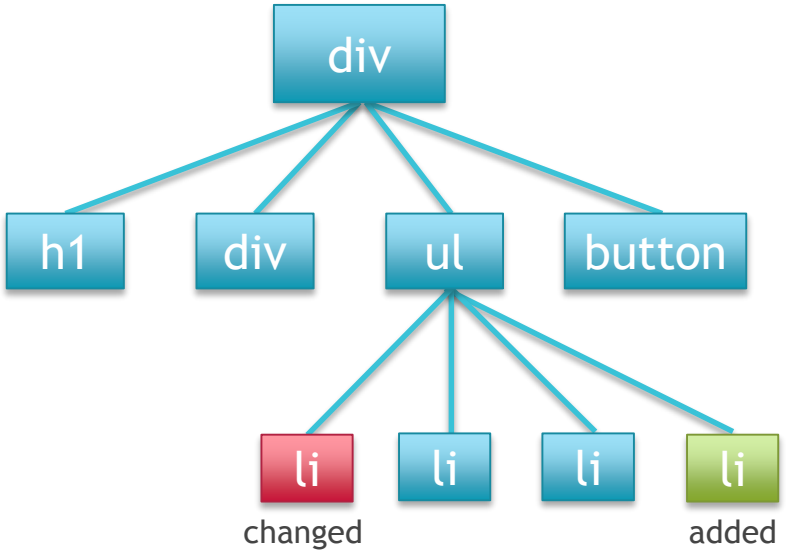
Component returns core of Virtual DOM

```
class CommentList extends React.Component {
        constructor(props){
                super(props);
                this.state = {data: ["comment1", "comment2", "comment3"]}
        }
        handleClick(){
                 let newData = [“NEW_comment1", "comment2", "comment3", "comment4"]
                 this.setState({data: newData })
        }
        render(){
                return (
                <div>
                        <h1>{this.props.title}</h1>
                        <div>Some complex DIV</div>
                        <ul>{this.state.data.map((elem)=><li>{elem}</li>)}</ul>
                        <button onClick={this.handleClick.bind(this)}>Resfresh</button>
                </div>
                );
        }
}
ReactDOM.render(<CommentList title="My List"/>, document.getElementById("container"));
```

Rebuild the whole Virtual DOM

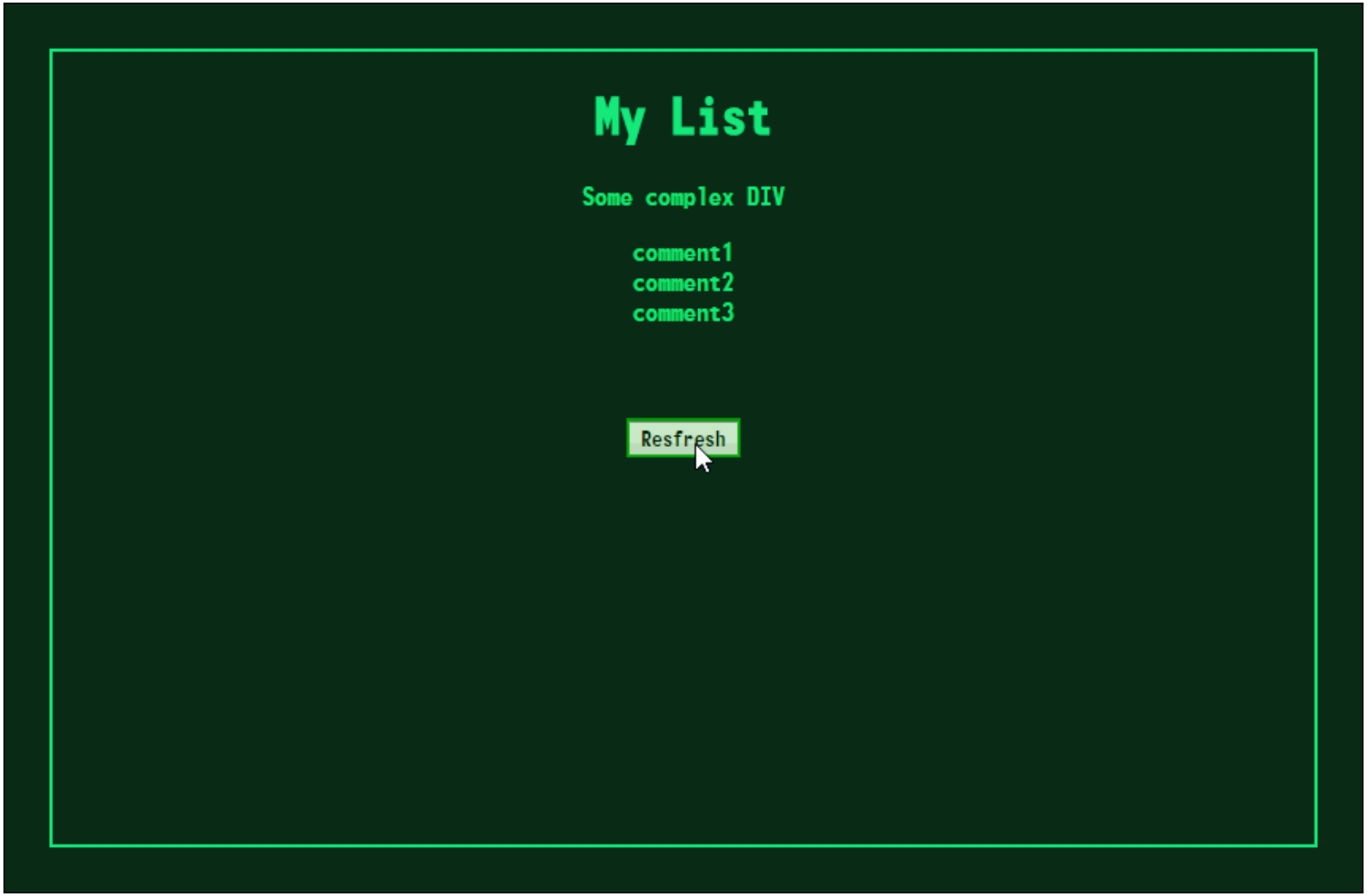


Virtual DOM t=1



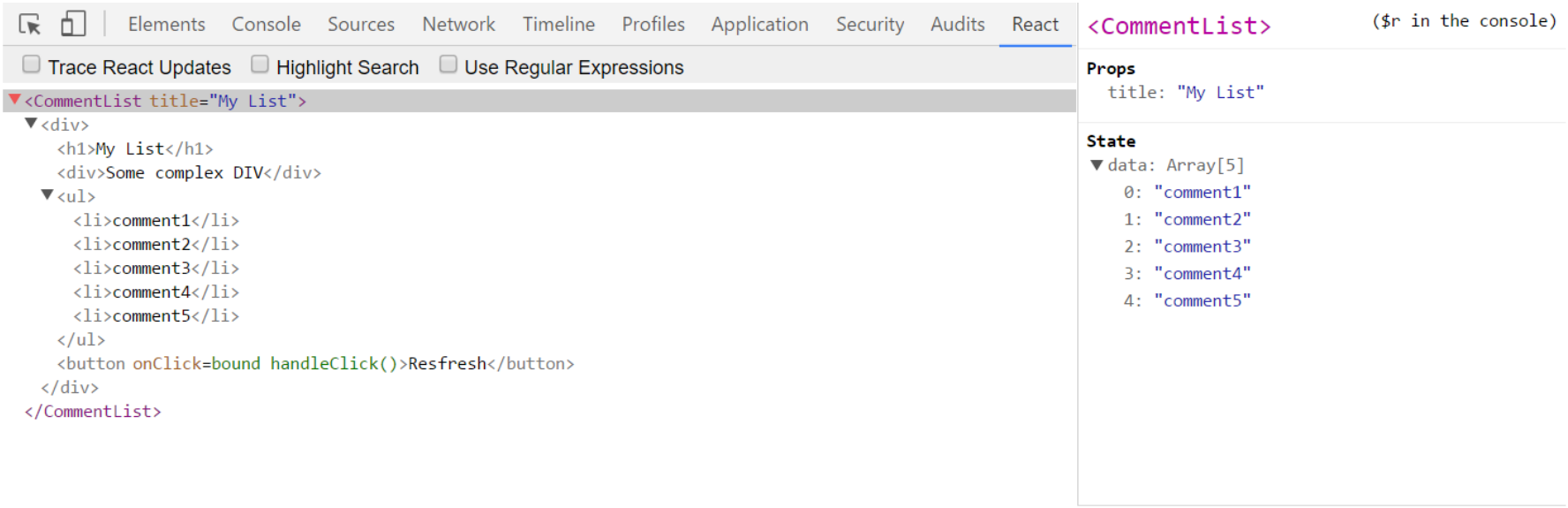
Virtual DOM t=2

Mutations in real DOM

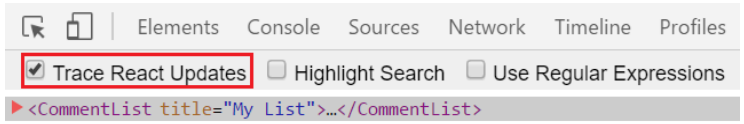


React Developers Tools

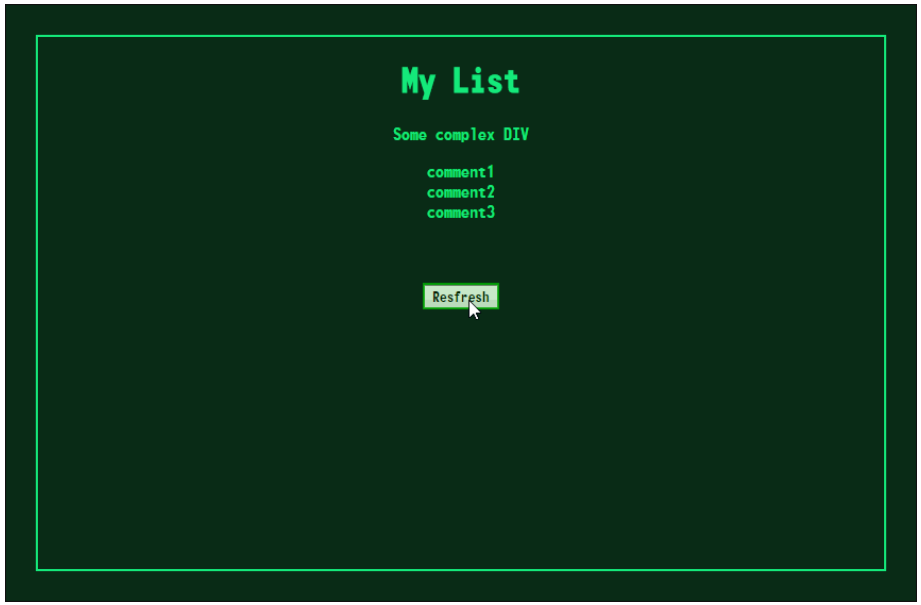
React Developers Tools is a system that allows you to inspect a React Virtual DOM including the Component hierarchy



React Developers Tools



Mutations in real DOM



Mutations in virtual DOM

Reconciliation

Reconciliation

The process of updating your UI to match your application state

Accurate Diff algorithm

Accurate Diff algorithm generates the **minimum number of operations** to transform one tree into another.

Accurate Diff algorithm

Accurate Diff algorithm has a complexity of

O(n^3)

Accurate Diff algorithm

Accurate Diff algorithm have a complexity of

O(n^3)

10000 nodes in a tree

10000^3 = 1000 * 10^9

≈ 1000 sec! at 1GHz

Accurate Diff algorithm

Accurate Diff algorithm have a complexity of

O(n^3)

10000 nodes in a tree

10000^3 = 1000 * 10^9

≈ 1000 sec! at 1GHz

≈ **17 minutes!!!**

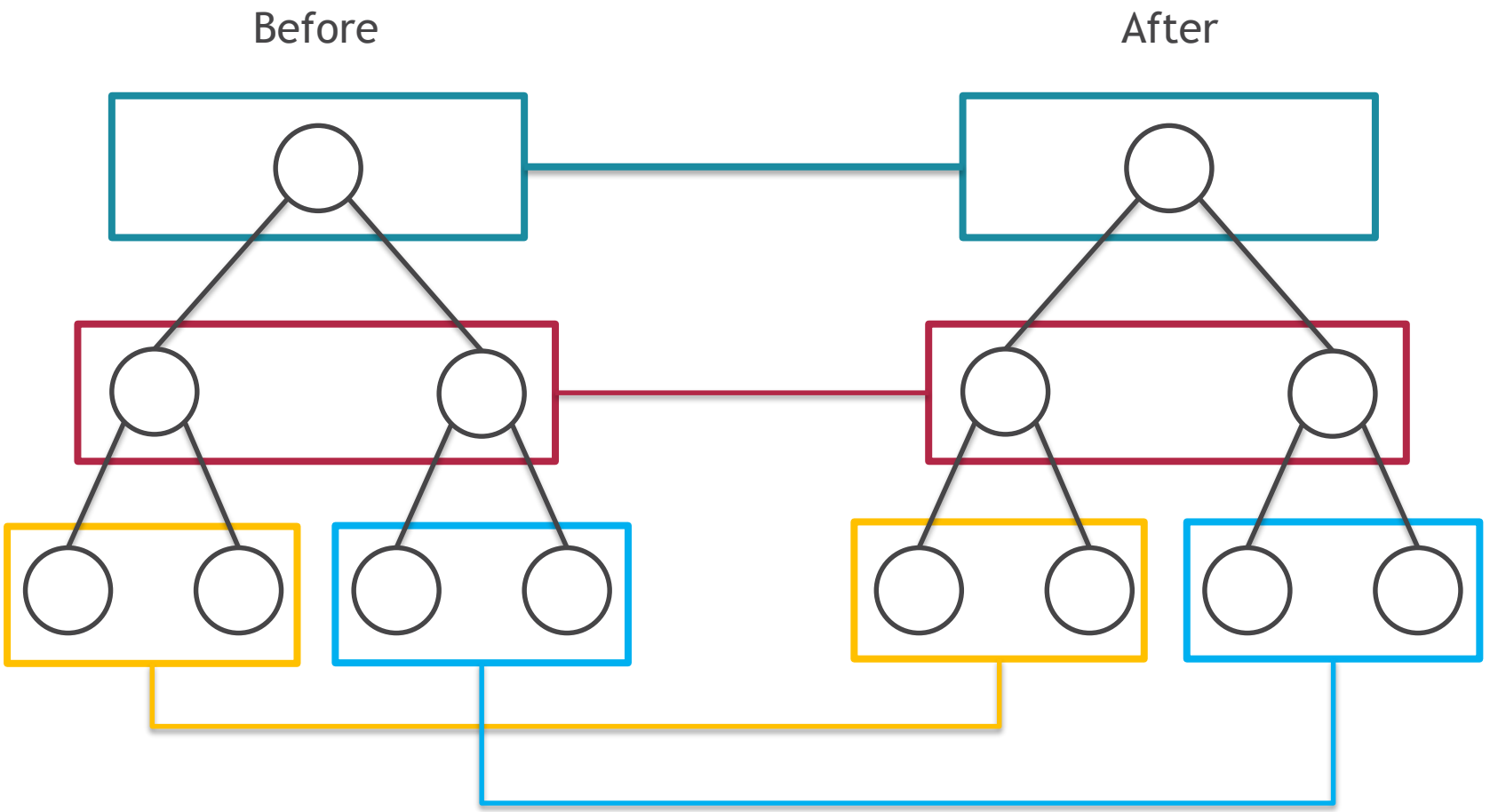
React Diff algorithm

Instead of optimal algorithm, React implements a heuristic algorithm which has a complexity of

O(n)

React Diff algorithm

React only tries to reconcile trees **level by level**



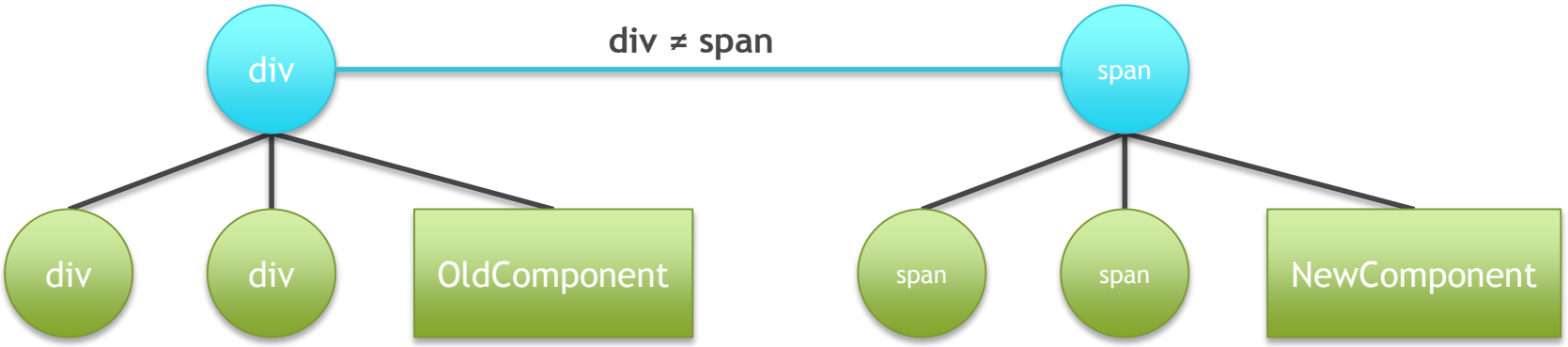
React Diff algorithm

Assumption:

Two elements of different types will produce different trees.

```
<div>
    <div>Some complex DIV</div>
    <div>One more complex DIV</div>
    <OldComponent/>
</div>
```

```
<span>
    <span>Some span</div>
    <span>One more span</div>
    <NewComponent/>
</span>
```



React Diff algorithm

Reconciliation can drastically change the behavior and performance of a React application

Case 1

```
render() {
  if (this.state.showWarning) {
        return (
              <div>
                    <Warning />
                    <StatefulComponent />
              </div>
        );
  }

  return (
    <div>
      <StatefulComponent />
    </div>
  );
}
```

Case 2

```
render() {
        return (
        <div>
              {this.state.showWarning ? <Warning/> : null}
              <StatefulComponent />
        </div>
        );
}
```

React Diff algorithm

Pass	1	2
this.state.showWarning	false	true
Case 1 return	[<StatefulComponent>]	[<Warning>, <StatefulComponent>]
Case 2 return	[null, <StatefulComponent>]	[<Warning>, <StatefulComponent>]

React Diff algorithm

Assumption:
The developer can hint at which child element may be stable across different renders with a key prop.

- React iterates through the new set of children.
- For each child, React checks whether there is an old child that has the same key as the new child. If an explicit key is not provided, React uses its position.
  - If there is **NO new child with the same key as an old child**, the old child is unmounted.
  - If there is **NO old child with the same key as a new child**, the new child is mounted.
  - If there is an **old and new child with the same key**, we use shouldUpdateReactComponent to decide whether we should update the instance vs doing a clean unmount/mount.

React Diff algorithm

If an explicit key is not provided, React uses its position.

```
<ul>
      <li>first</li>
      <li>second</li>
</ul>
```

```
<ul>
      <li> first </li>
      <li> second </li>
      <li> third </li>
</ul>
```

React Diff algorithm

Inserting an element at the beginning has worse performance.

```
<ul>
      <li>Paris</li>
      <li>New York</li>
</ul>
```

```
<ul>
      <li>London</li>
      <li>Paris</li>
      <li>New York</li>
</ul>
```

React will mutate every child instead of realizing it!

React Diff algorithm

When children have keys, React uses the key to match children in the original tree with children in the subsequent tree.

```
<ul>
      <li key = "2015">Paris</li>
      <li key = "2016">New York</li>
</ul>
```

```
<ul>
      <li key = "2014">London</li>
      <li key = "2015">Paris</li>
      <li key = "2016">New York</li>
</ul>
```

Now React knows that the element with key “2014” is the new one, and the elements with the keys “2015” and “2016” have just moved

Optimizing Performance

Avoid Reconciliation

If you are sure that Component doesn’t need to update, you can tell React to do nothing with it.

shouldComponentUpdate

shouldComponentUpdate is a lifecycle function of React.Component

```
shouldComponentUpdate(nextProps, nextState) {
      return true; //returns true by default
}
```

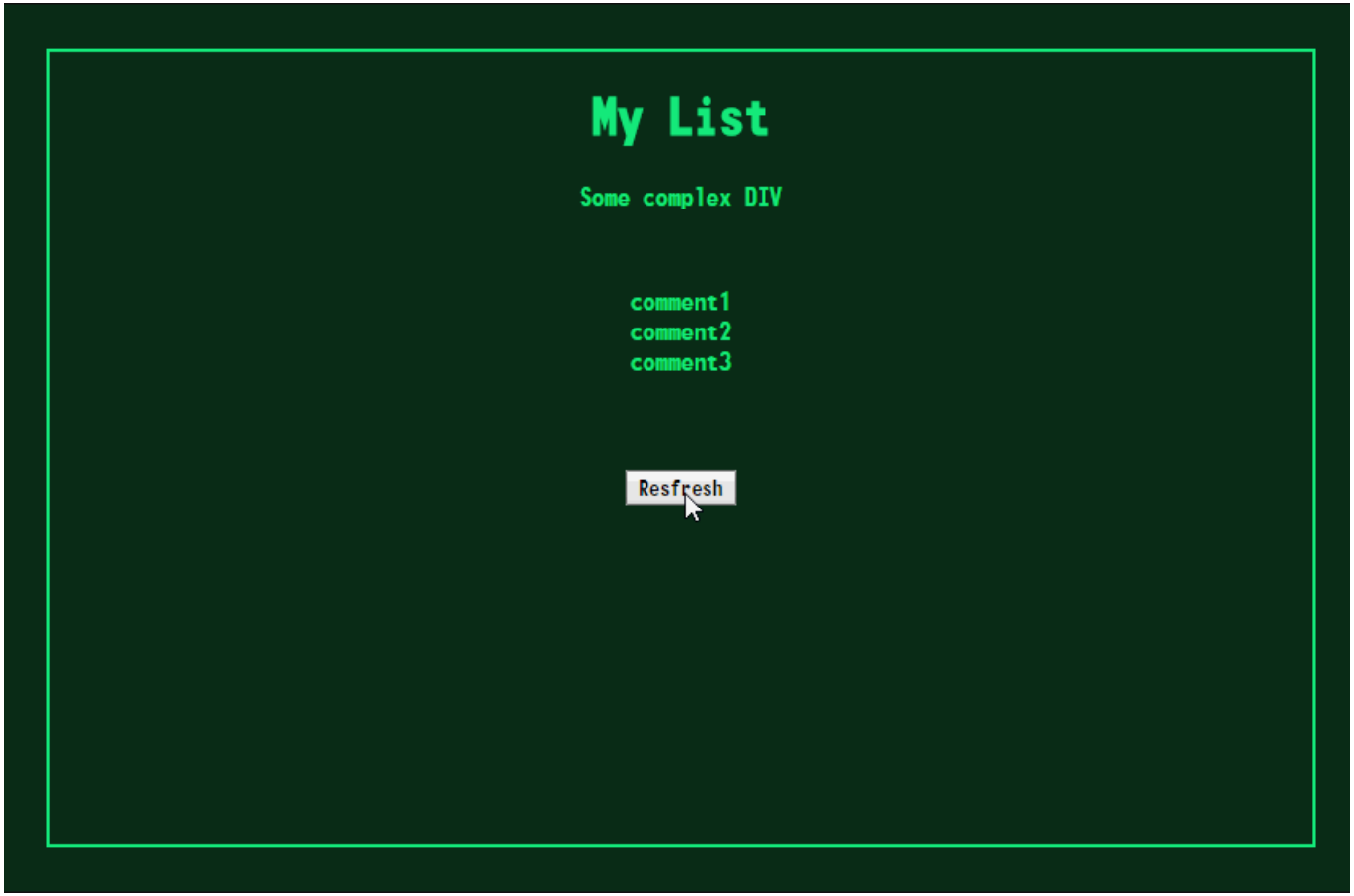
You can implement this function in your Component and make it to return ‘false’ to skip rendering of Component.

Example with shouldComponentUpdate

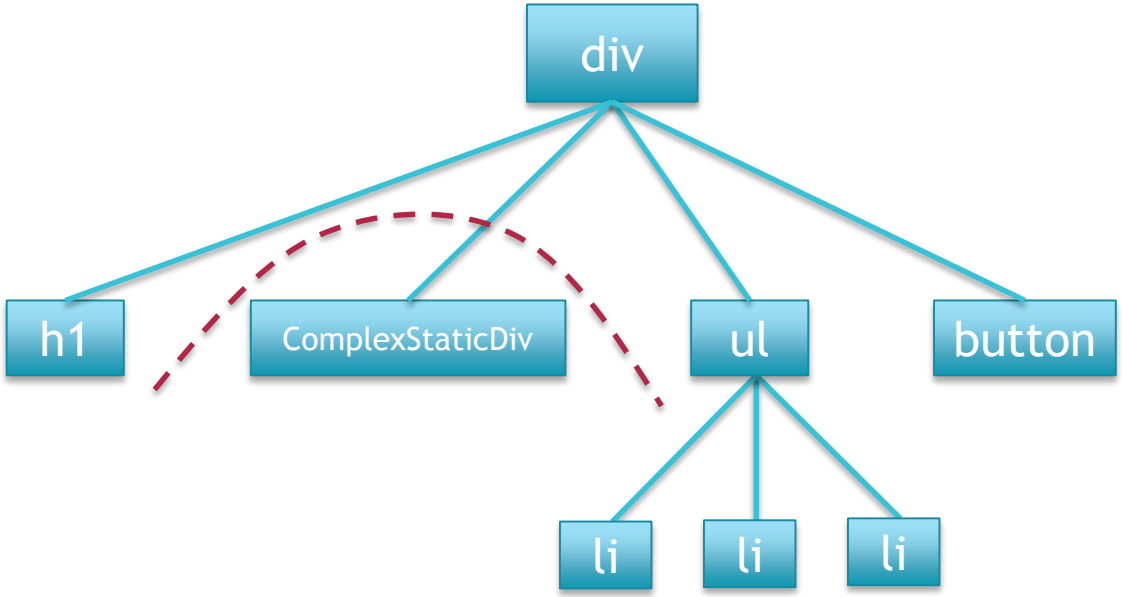
```
class ComplexStaticDiv extends React.Component{

        shouldComponentUpdate(nextProps, nextState){
                return false;
        }
        render(){
                return(
                        <div>Some complex DIV</div>
                );
        }
}

class CommentList extends React.Component {
//…
        render(){
                return (
                <div>
                        <h1>{this.props.title}</h1>
                        <ComplexStaticDiv/>
                        <ul>{this.state.data.map((elem)=><li>{elem}</li>)}</ul>
                        <button onClick={this.handleClick.bind(this)}>Resfresh</button>
                </div>
                );
        }
}
```

Example with shouldComponentUpdate

updating of **Virtual DOM**



Selective Sub-tree Rendering



THANK YOU!

QUESTIONS?