Code.Hub

The first Hub for Developers

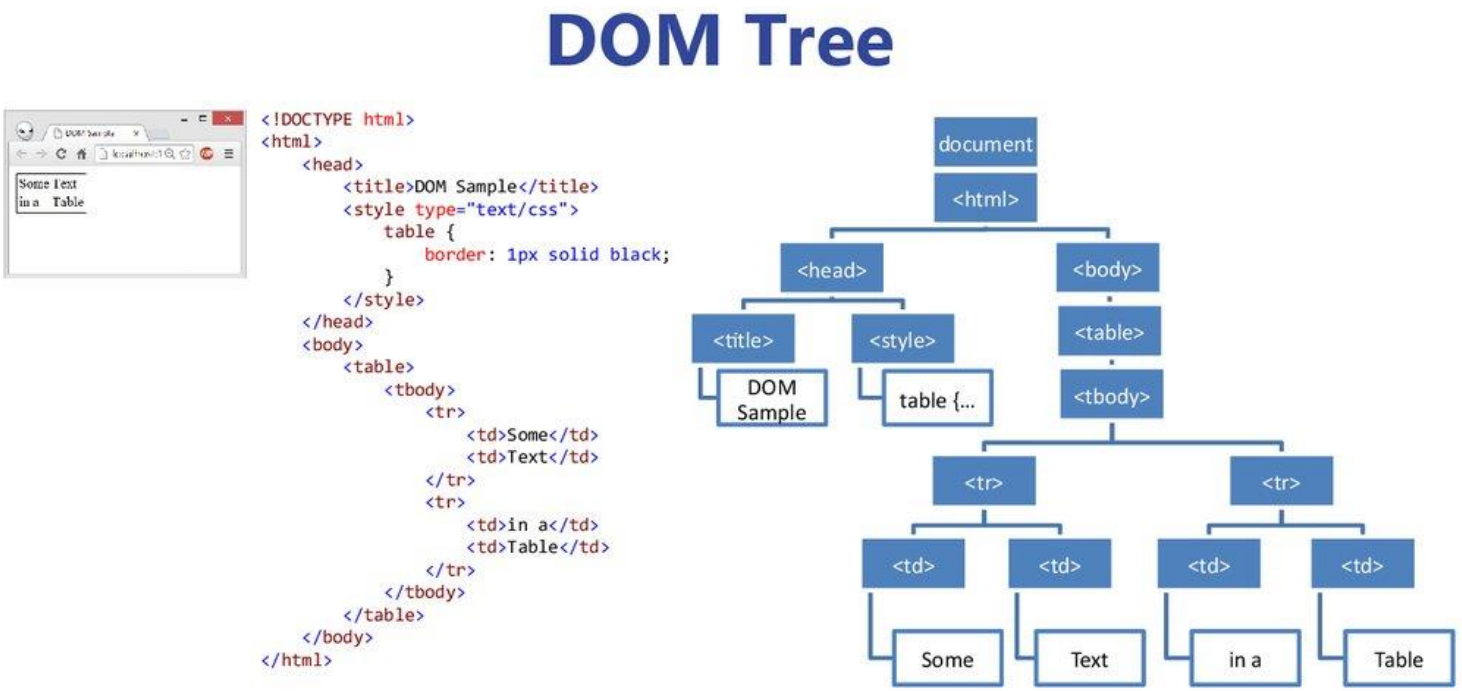
Ztoupis Konstantinos

Reconciliation

Code.Learn Program:
React

The DOM

The browser builds the DOM by parsing the code you write, it does this before it renders the page



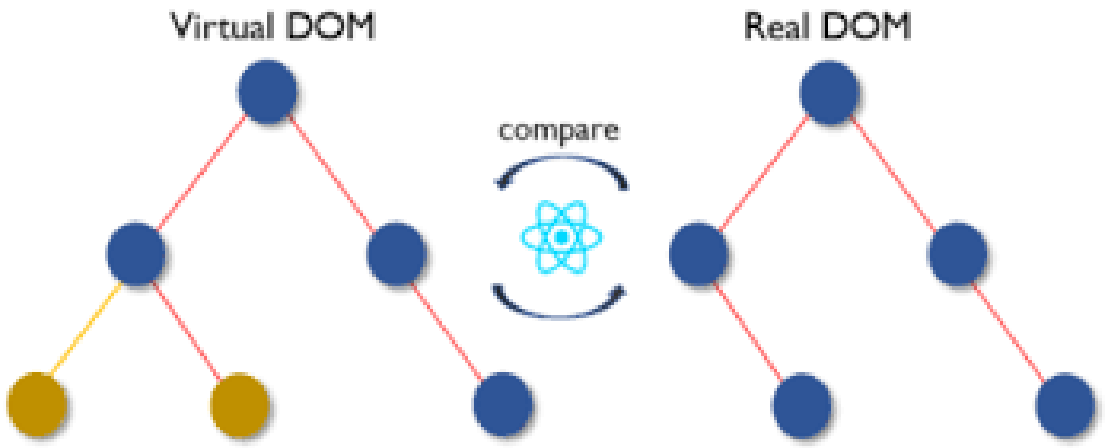
Code.Hub

The Issue

Most modern web pages have huge DOM structures and a simple change would cost too much, resulting in slower loading pages

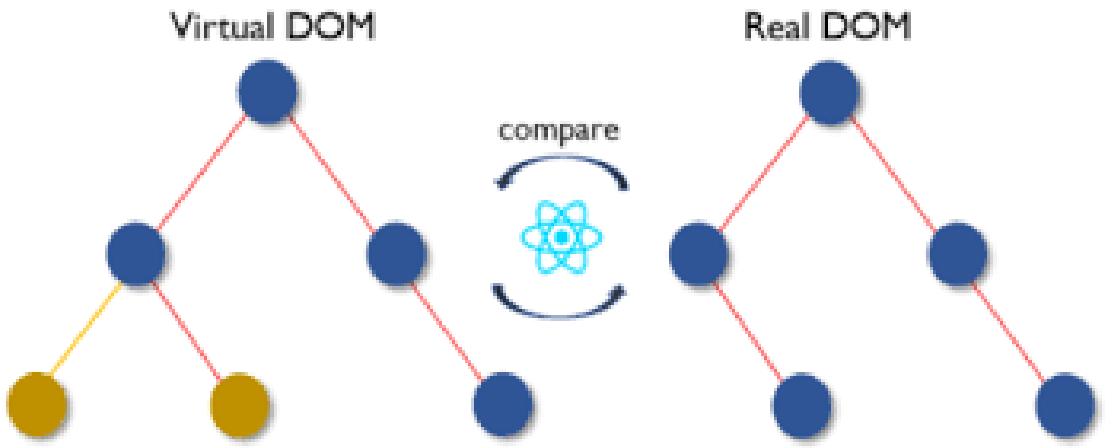
The Virtual DOM

The Virtual DOM is a copy of the HTML DOM. We can call it an abstraction of the HTML DOM.



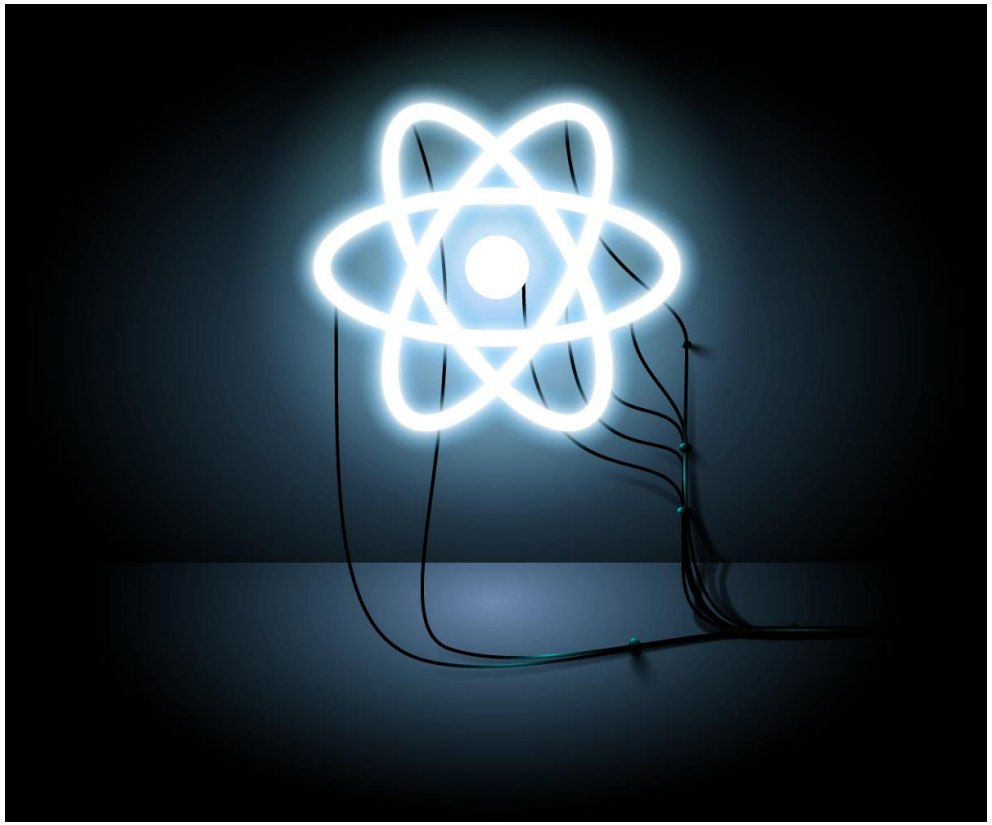
The Virtual DOM

Learning React's Virtual DOM and use this knowledge you will speed up your applications



Master of React

But to truly master React, you need to *think in React*



Things go wrong

- input fields get laggy
- checkboxes take a second to be checked
- modals have a hard time showing up

Solution: Understand a React component takes from being defined, to being rendered and then updated on a page

Behind JSX

- mix of HTML and JavaScript known as JSX
- browsers have no clue about JSX and its syntax
- browsers only understand plain JavaScript, so JSX will have to be transformed into it

Behind JSX

```
<div className='title'>
  Text
</div>
```

```
React.createElement(
  'div',
  { className: 'title' },
  'Text'
);
```

Behind JSX

- first argument: type of element. For HTML tags it will be a string with a tag's name.
- second argument: object with all of the element's attributes. It can also be an empty object if there are none.
- following arguments: element's children.

```
React.createElement(
  'div',
  { className: 'title' },
  'Text'
);
```

Behind JSX

```
<div className= 'title'>
  Text 1
  <br />
  Text 2
</div>
```

Behind JSX

```
React.createElement(
  'div',
  { className: 'title' },
  'Text 1',                // 1st child
  React.createElement('br'), // 2nd
child
  'Text 2'                 // 3rd child
)
```

Behind JSX

Values can also serve as arguments:

- Primitives *false, null, undefined* and *true*
- Arrays
- React *components*

```
React.createElement(
  'div',
  { className: 'title' },
  ['Text 1', React.createElement('br'), 'Text 2']
)
```

Power of React

```
function Table({ rows }) {
  return (<table>
      {rows.map(row => (
        <tr key={row.id}>
          <td>{row.title}</td>
        </tr>
      ))}
    </table>);
}
```

Reusable Components

Power of React

Users' perspective

```
<Table rows={rows} />
```



```
React.createElement(Table, { rows: rows });
```

Browsers' perspective

Adding components on a page

```
// defining a component
function Table({ rows }) { /* ... */ }

// rendering a component
ReactDOM.render(
// creating a component
  React.createElement(Table, { rows: rows }),
// inserting it on a page
  document.getElementById('#root')
);
```

Virtual DOM Object

ReactDOM.render is called, React.createElement is called too and it returns the following object:

```
{
  type: Table,
  props: {
    rows: rows
  },
  // ...
}
```

Virtual DOM

```
React.createElement(
  'div',
  { className: 'title' },
  'Text 1',
  'Text 2',
);
```

```
{
  type: 'div',
  props: {
    className: 'title',
    children: [
      'Text 1',
      'Text 2'
    ]
  }
}
```

Props directly in the JSX code

```
<div className= 'title'>Text 1Text 2</div>
```

=

```
<div
  className='title'
  children={['Text 1', 'Text 2',
]}
```

DOM Nodes

ReactDOM.render transforms virtual DOM objects into DOM nodes according to those rules:

- If a **type** attribute holds a *string* with a tag name: create a tag with all attributes listed under **props**.
- If we have a function or a class under **type:** call it and repeat the process recursively on a result.
- If there are any **children** under **props**: repeat the process for each child one by one and place results inside the parent's DOM node.

React Rendering Flow

```
<div className= 'title'>
  Text 1Text 2
</div>
```

Component

1

```
React.createElement(
  'div',
  { className: 'title' },
  ['Text 1', ' Text 2']
)
```

'formal'

2

```
{
  type: 'div',
  props: {
    className: 'title',
    children: ['Text 1','Text 2']
  }
}
```

Virtual Dom

3

Browser

Text 1Text

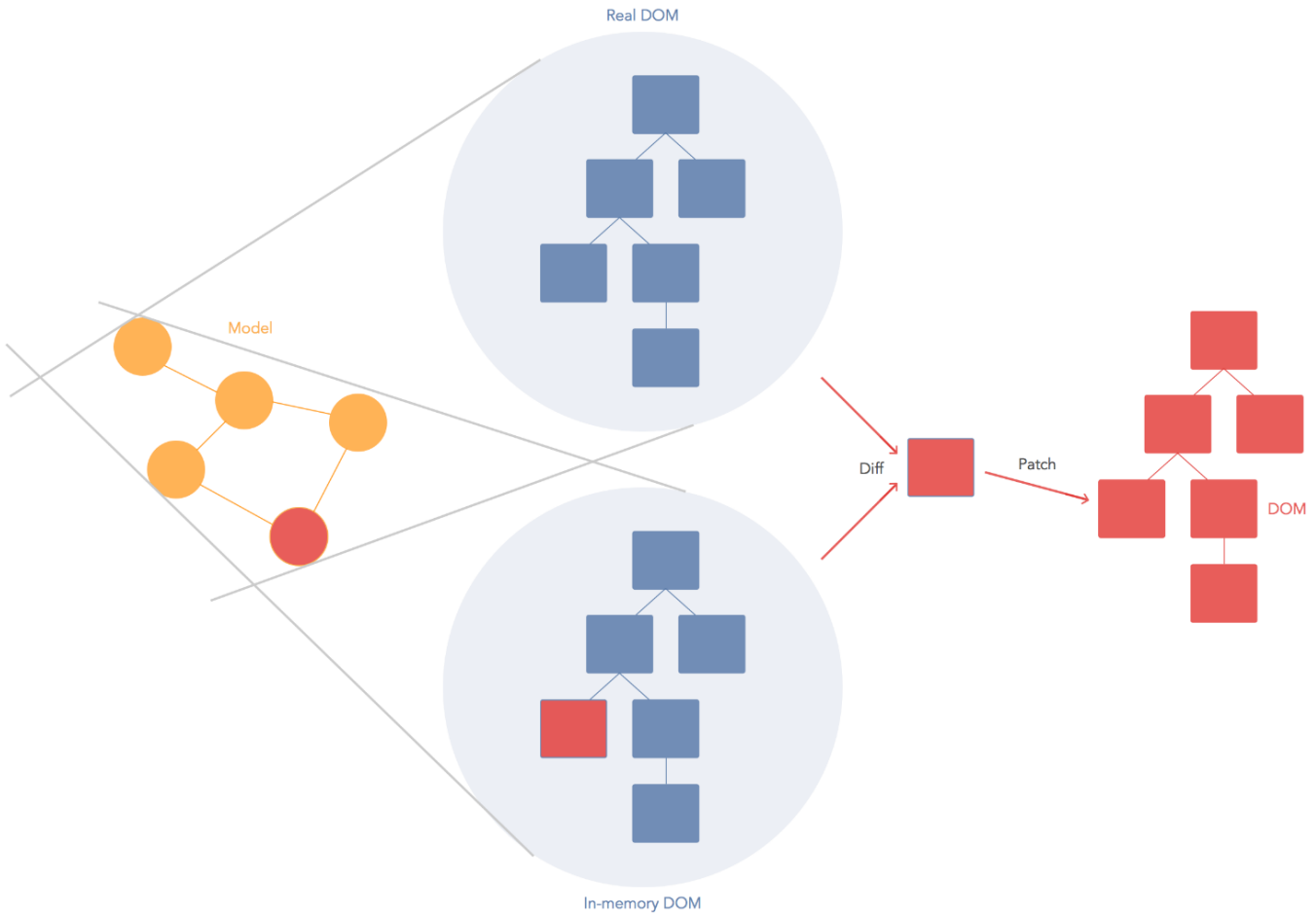
Code.Hub

Rebuilding the DOM

- Creating DOM nodes from scratch and adding them on the page
- Compare Virtual DOM objects
- 4 scenarios

Reconciliation

The process through which React updates the DOM



Code.Hub

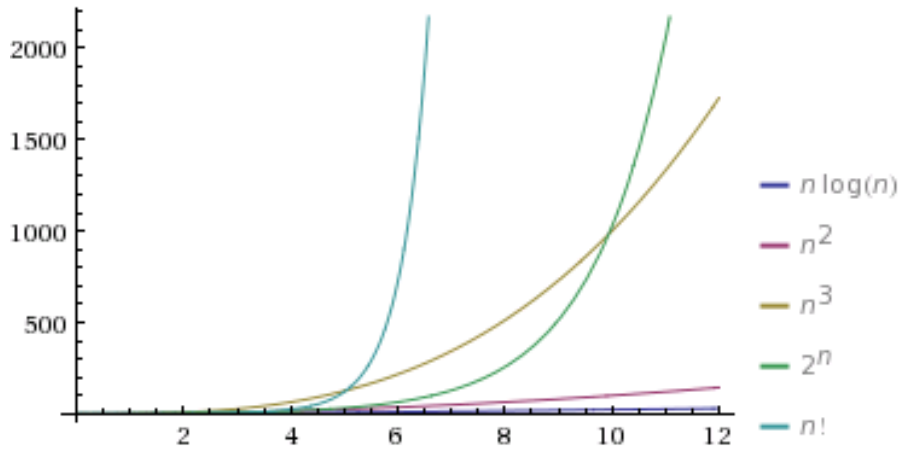
Motivation

Minimum number of operations to transform one tree into another:
a complexity in the order of $O(n^3)$

Motivation

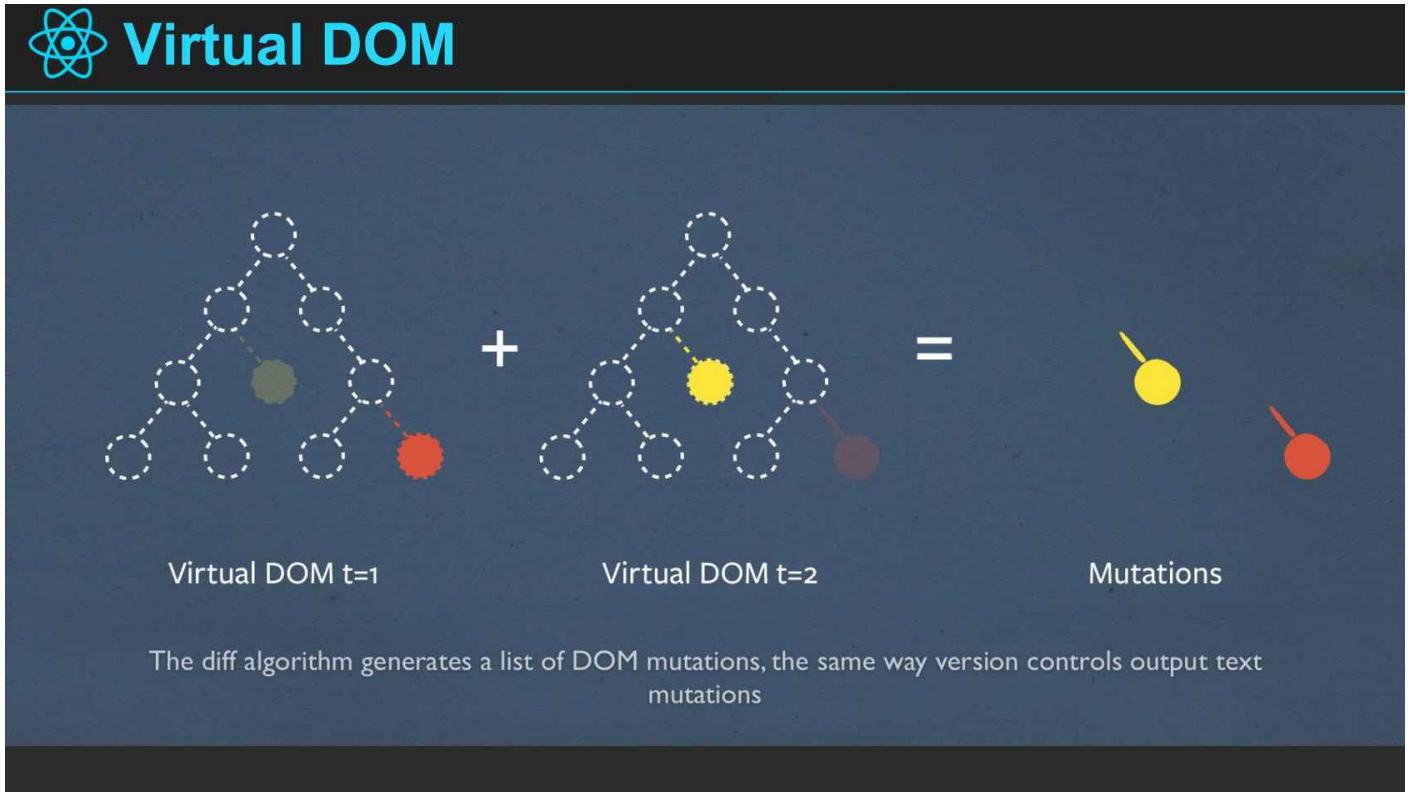
React implements a heuristic O(n) algorithm based on two assumptions:

- Two elements of different types will produce different trees
- The developer can hint at which child elements may be stable across different renders with a key prop



The Diffing Algorithm

React uses === (triple equals) to compare type values, so they have to be the same instances of the same class
or the same function



Code.Hub

Scenario 1

type is a string, *type* stayed the same across calls, *props* did not change either

```
// before update
{ type: 'div', props: { className: 'title' } }

// after update
{ type: 'div', props: { className: 'title' } }
```

Scenario 2

type is still the same string, *props* are different

```
// before update
{ type: 'div', props: { className: 'title' } }

// after update
{ type: 'div', props: { className: 'magic' } }
```

Scenario 3

type has changed to a different *String*, or from *String* to a component

```
// before update
{ type: 'div', props: { className: 'title' } }

// after update
{ type: 'span', props: { className: 'title' } }
```

Scenario 4

type is a component (function, class)

```
// before update:
{ type: Table, props: { rows: rows } }

// after update:
{ type: Table, props: { rows: rows } }
```

Scenario 4

- start tree reconciliation process
- look inside the component to make sure that the values returned on render did not change
- rinse and repeat for each component down the tree

Recursing On Children

```
// before
props: {
  children: [
      { type: 'span' },
      { type: 'div' },
      { type: 'br' }
  ]
},
// ...
```

```
// after
props: {
  children: [
    { type: 'div' },
    { type: 'span' },
    { type: 'br' }
  ]
},
// ..
```

Recursing On Children

- checking any array inside props.children
- starts comparing elements in it with the ones in the array: index 0 will be compared to index 0, index 1 to index 1, etc …
- for each pair, React will apply the set of rules described above

Keys

- React supports a key attribute. Children have keys -> React uses the key to match children in the original tree with children in the subsequent tree.

- Finding a key is usually not hard. The element you are going to display may already have a unique ID

```
// Now React will look on key, not index
props: {
  children: [
    { type: 'span', key: 'key0' },
    { type: 'div', key: 'key1' },
    { type: 'br', key: 'key2' }
  ]
},
// ...
```

Code.Hub

Keys

Keys help React identify which items have changed, are added or are removed

```
const items = [3, 2, 1];
const list = items.map((item) =>
  <li key={item.toString()}>
    {item}
  </li>
);
```

Best way: use a string that uniquely identifies a list item among its siblings as a key

Lists

```
<ul>
        <li>title 1</li>
        <li>title 2</li>
</ul>
->
<ul>
        <li>title 1</li>
        <li>title 2</li>
        <li>title 3</li>
</ul>
```

Inserting an element at the beginning has worse performance

```
<ul>
        <li>title 1</li>
        <li>title 2</li>
</ul>
->
<ul>
        <li>title 3</li>
        <li>title 1</li>
        <li>title 2</li>
</ul>
```

Indexes as keys

Do not use indexes for keys if the order of items may change

This can negatively impact performance and may cause issues with component state

Keys Must Only Be Unique Among Siblings

- Keys used within arrays should be unique among their siblings
- No need to be globally unique. We can use the same keys when we produce two different arrays

Extracting Components with Keys

```react
function ListItem(props) {
  const value = props.value;
  return (
    // Wrong! There is no need to specify the key here:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Wrong! The key should have been specified here:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

```react
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()}
              value={number} />

  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Pass keys to a component

Keys serve as a hint to React but they don't get passed to components. Pass the key in a component, pass it explicitly as a prop with a different name

```
const list = items.map((item) =>
  <List
    key={item.id}
    id={item.id}
    content={item.content} />
);
```

Use index as A Key

You can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

key=index

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
5	first	22:31:22 GMT+0300 (GTB Daylight Time)
3	second	22:31:14 GMT+0300 (GTB Daylight Time)
1	third	22:31:04 GMT+0300 (GTB Daylight Time)
2	fourth	22:31:11 GMT+0300 (GTB Daylight Time)
4	fifth	22:31:18 GMT+0300 (GTB Daylight Time)

key=index

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
1	first	22:31:04 GMT+0300 (GTB Daylight Time)
2	second	22:31:11 GMT+0300 (GTB Daylight Time)
3	third	22:31:14 GMT+0300 (GTB Daylight Time)
4	fourth	22:31:18 GMT+0300 (GTB Daylight Time)
5	fifth	22:31:22 GMT+0300 (GTB Daylight Time)

key=index

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
5	first	22:31:22 GMT+0300 (GTB Daylight Time)
4	second	22:31:18 GMT+0300 (GTB Daylight Time)
3	third	22:31:14 GMT+0300 (GTB Daylight Time)
2	fourth	22:31:11 GMT+0300 (GTB Daylight Time)
1	fifth	22:31:04 GMT+0300 (GTB Daylight Time)

key=id

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
5	fifth	22:33:44 GMT+0300 (GTB Daylight Time)
3	third	22:33:31 GMT+0300 (GTB Daylight Time)
1	first	22:33:17 GMT+0300 (GTB Daylight Time)
2	second	22:33:28 GMT+0300 (GTB Daylight Time)
4	fourth	22:33:35 GMT+0300 (GTB Daylight Time)

key=id

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
1	first	22:33:17 GMT+0300 (GTB Daylight Time)
2	second	22:33:28 GMT+0300 (GTB Daylight Time)
3	third	22:33:31 GMT+0300 (GTB Daylight Time)
4	fourth	22:33:35 GMT+0300 (GTB Daylight Time)
5	fifth	22:33:44 GMT+0300 (GTB Daylight Time)

key=id

Add New to Start | Add New to End | Sort by Earliest | Sort by Latest

ID		created at
5	fifth	22:33:44 GMT+0300 (GTB Daylight Time)
4	fourth	22:33:35 GMT+0300 (GTB Daylight Time)
3	third	22:33:31 GMT+0300 (GTB Daylight Time)
2	second	22:33:28 GMT+0300 (GTB Daylight Time)
1	first	22:33:17 GMT+0300 (GTB Daylight Time)

When state changes

- causes a re-render too
- not of the whole page, but *only of a component itself and its children*
- parents and siblings are spared

Mounting/Unmounting

JSX

```
<div>
  <Header />
  <Content />
  <Footer />
</div>
```

Virtual DOM

```
// ...
props: {
  children: [
    { type: Header },
    { type: Content },
    { type: Footer }
  ]
}
// ...
```

Content includes 100 components

Mounting/Unmounting

Virtual DOM

Removing Header

```
// ...
props: {
  children: [
    { type: Content },
    { type: Footer }
  ]
}
// ...
```

React unmounts the whole Content and mounts it again, rendering all its children: 100+ components

Mounting/Unmounting

JSX

```
<div>
  {isShown && <Header />}
  <Content />
  <Footer />
</div>
```

Virtual DOM

```
// ...
props: {
  children: [
    false,
    { type: Table },
    { type: Footer }
  ]
}
// ...
```

Use short circuit boolean evaluation

Mounting/Unmounting with HOCs

```
function withTitle(Component) {
return function(props) {
    return <Component {...props} title={title} />;
  }
}
```

```
render() {
  // Creates a new instance on each render
  const CompoWithTitle = withTitle(Component);
  return <CompoWithTitle />
}
```

Mounting/Unmounting with HOCs

```
{  // on first render
  type: CompoWithTitle,
  props: {}
}
{  // on second render
  type: CompoWithTitle, // Same name, but different instance
  props: {}
}
```

Mounting/Unmounting with HOCs

- diffing algorithm on CompoWithTitle
- same title references a different instance
- triple equals comparison fails
- a full re-mount has to happen

As creating a HOC inside of a parent's render method, when the tree is re-rendered, full re-mount has to happen.

Mounting/Unmounting with HOCs

Always create a HOC outside of render:

```
// Creates a new instance just once
const CompoWithTitle = withTitle(Component);

class App extends React.Component() {
  render() {
    return <CompoWithTitle />;
  }
}
```

Tradeoffs

The reconciliation algorithm is an implementation detail.

React relies on heuristics -> if the assumptions behind them are not met, performance will suffer.

- The algorithm will not try to match subtrees of different component types.
- Keys should be stable, predictable, and unique.

Wrapping Up

- It’s good to understand reconciliation process to make your app fast
- React only changes what it needs, not full rerender
- The diffing process is so fast that you will not notice it