React II

Highlights from Gifs app

Controlled components

The user **types** in an **input**,
The change triggers a **setState( )**
The **state** re-defines the **input value**

```
class SearchBar extends Component {
  constructor(props) {
    super(props)
    this.state = { term: '' };
  }

  render( ) {
    return (
      <input
        value={this.state.term}
        onChange={e => this.setState({term: e.target.value})} />
    );
  }
}
```

React state as the **single source of truth**

List patterns

Map over an array of **props**
Pass props to **children**
React needs a unique **key** by child
(think of a DB index)

```
const GifList = (props) => {
  return (
    <div className="gif-list">
      {props.gifs.map((gif) => {
        return <Gif key={gif.id} id={gif.id} />;
      })}
    </div>
  );
};
```

Handling `null`

Initial state might hold **`null` values**
Handle it with an **if statement**

```
const Gif = (props) => {
  // [...]
  if (!props.id) {
    return <p>Loading...</p>;
  }

  return <img src={url} className="gif" />;
};;
```

this binding (1)

```
class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.state = { term: '' };
  }

  handleChange(event) {
    this.setState({ term: event.target.value });
  ⚠️ // TypeError: Cannot read property 'setState' of undefined
  }

  render() {
    return (
      <input
        value={this.state.term}
        onChange={this.handleChange}
      />
    );
  }
}
```

this binding (2)

```
class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.state = { term: '' };
  }

  handleChange(event) {
    this.setState({ term: event.target.value });
  }

  render() {
    return (
      <input
        value={this.state.term}
        onChange={this.handleChange.bind(this)}
      />
    );
  }
}
```

this binding (3)

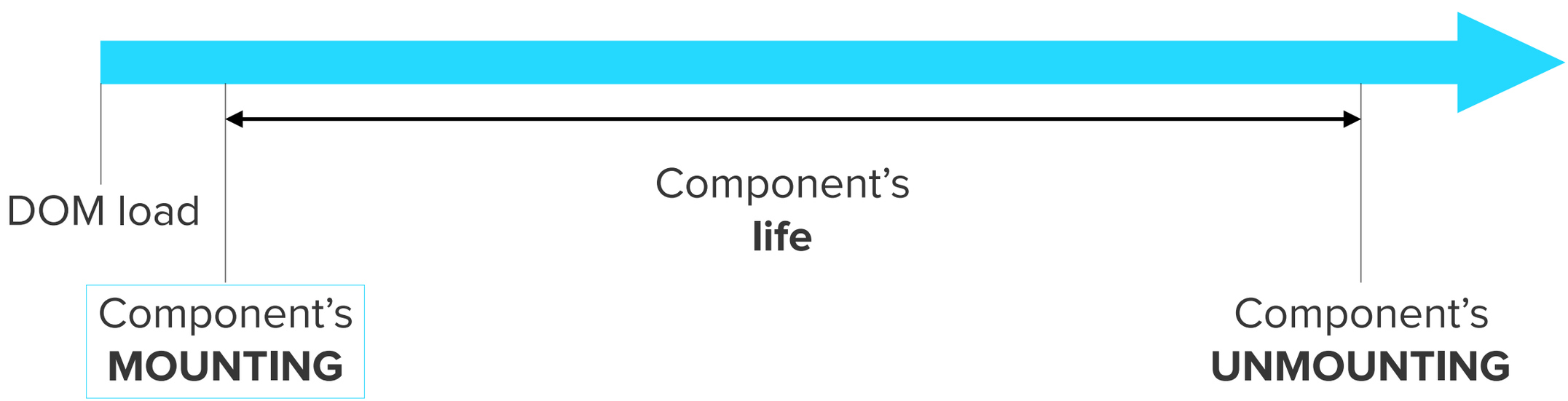
```
class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.state = { term: '' };
  }

  handleChange = (event) => {
    this.setState({ term: event.target.value });
  }

  render() {
    return (
      <input
        value={this.state.term}
        onChange={this.handleChange}
      />
    );
  }
}
```

⚠️ requires **babel-plugin-transform-class-properties**

Components lifecycle

Life in the DOM

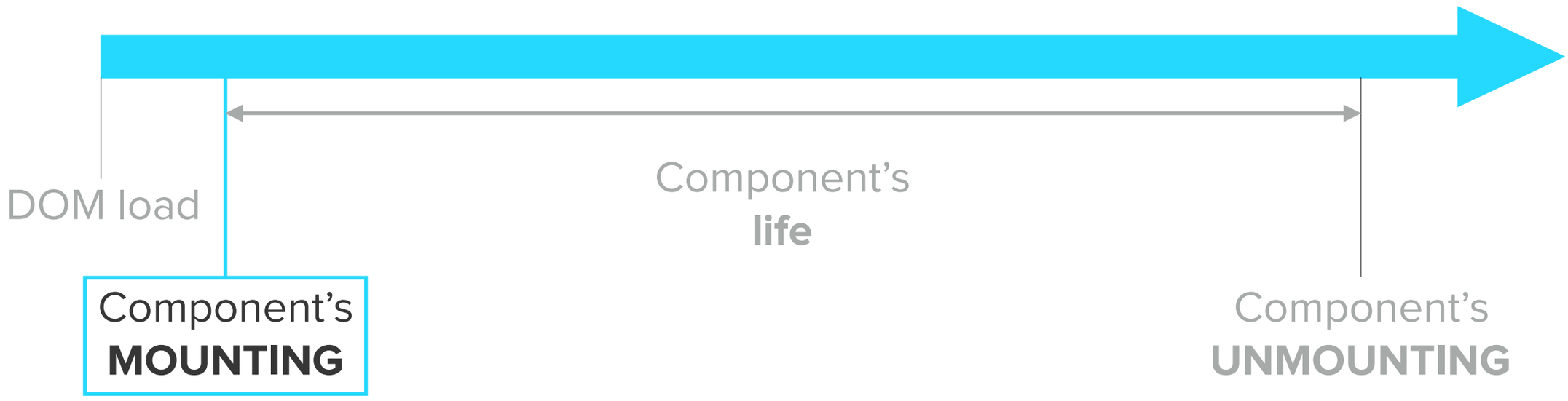
The life of a component
begins when it's **inserted** in the DOM
ends when the component **leaves** the DOM



Mounting

When a component is **inserted** in the DOM
The following methods are successively called:

```
constructor()
componentWillMount()
render()
componentDidMount()
```

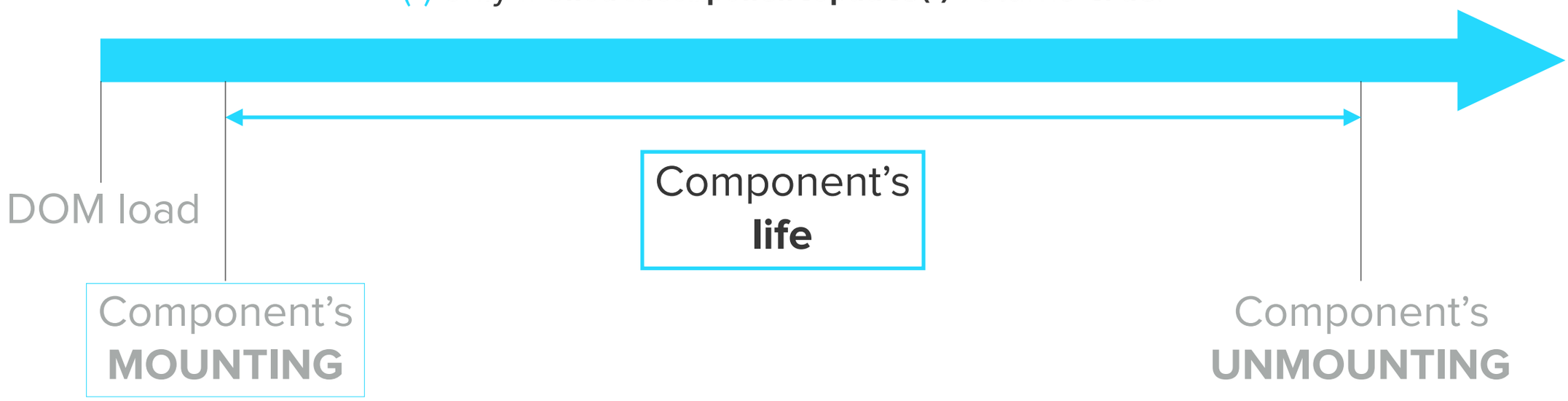


Updating props

During its life, when it receives **new props** from its parent
The following methods are successively called:

```
componentWillReceiveProps()
shouldComponentUpdate()
componentWillUpdate() (*)
render() (*)
componentDidUpdate() (*)
```

(*) only if **`shouldComponentUpdate()`** returns **`true`**!

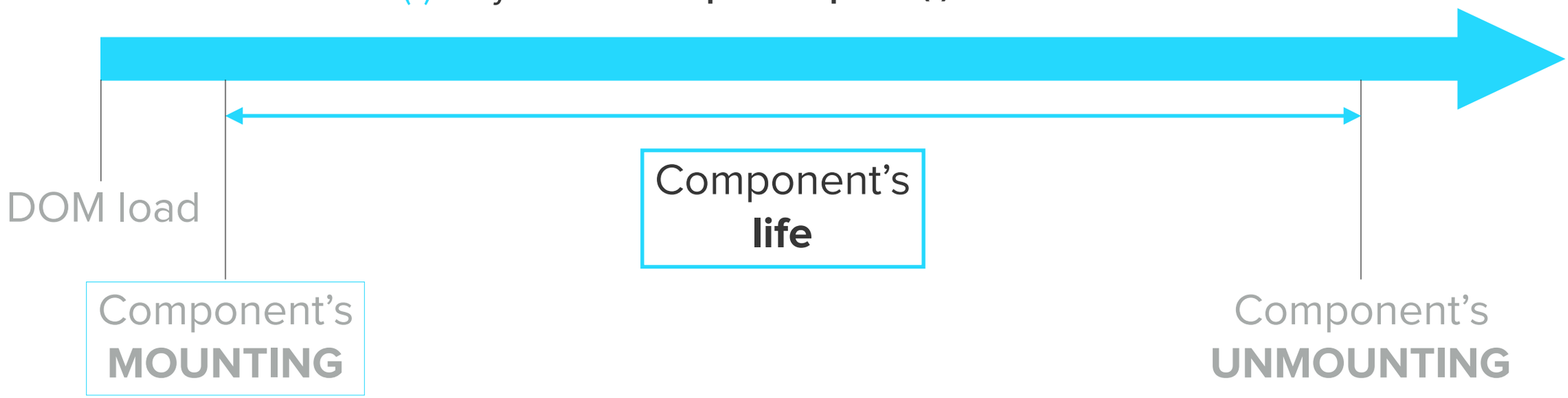


Updating state

During its life, whenever **`this.setState()`** is called:
The following methods are successively called:

```
shouldComponentUpdate()
componentWillUpdate() (*)
render() (*)
componentDidUpdate() (*)
```

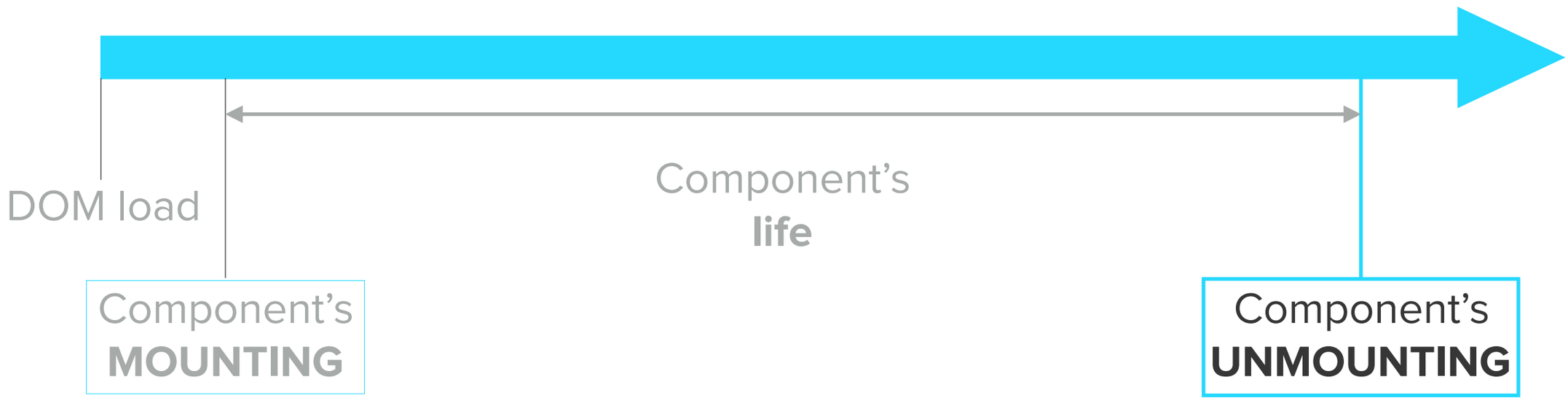
(*) only if **`shouldComponentUpdate()`** returns **`true`**!



Unmounting

When a component is being **removed** from the DOM
The following method is called:

```
componentWillUnmount()
```



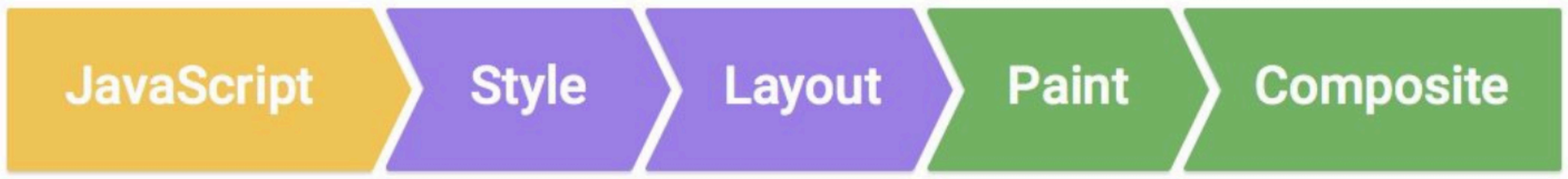
Behind the scenes:

Rendering Optimization

Problem

Whenever a **browser** is asked to perform a change in the DOM it takes an incompressible amount of time to **redraw** the screen.

The **browser's rendering pipeline** has 5 steps:



We don't want the browser going through all these steps if the result displayed to the user ends up **unchanged**.

Solution

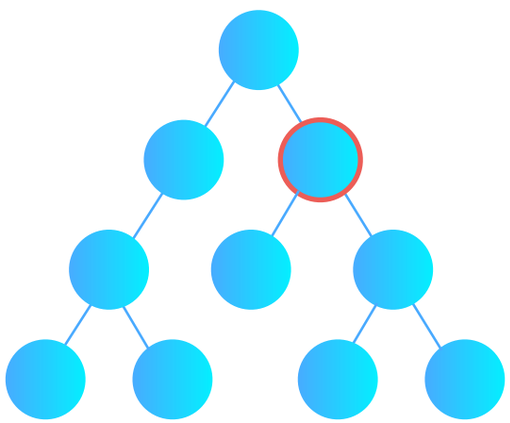
Minimize and **batch** the DOM changes that make redraws necessary.

React embraces this idea with its **Virtual DOM**

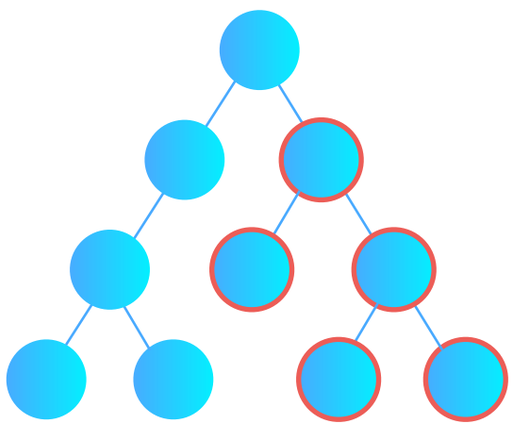
Virtual DOM

When a component receives new props or state
The **`render()`** method of the component is called.

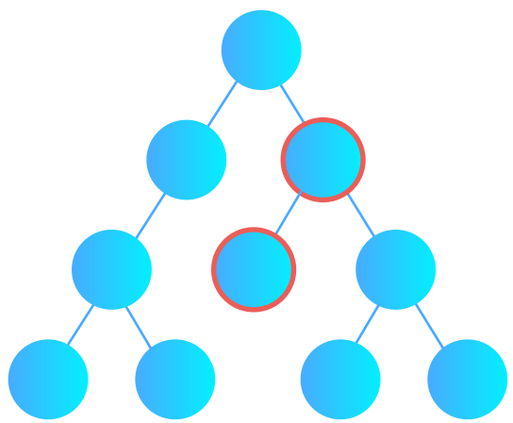
React builds a “Virtual DOM” and compares it to the DOM with a **diff algorithm** going through the nodes.



`setState()`



`render()`

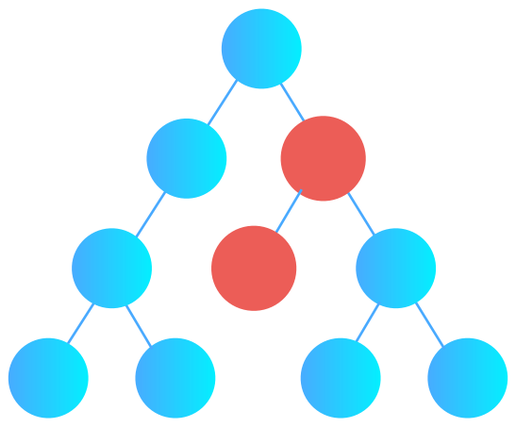


Effective diff

Patch

React applies only the **diff** as a **patch** to the actual DOM

React spares the **browser** from going through all the steps of the **pixel pipeline** for the **unchanged nodes**



Patch applied to
the DOM

Reconciliation

On a component re-render, React builds the corresponding **subtree** and goes through reconciliation.

If root **types** differ, React builds a **whole new tree** from scratch.
Old DOM nodes are destroyed, components **unmounted**.

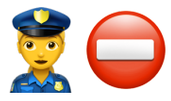
If not, it compares DOM elements based on their **attributes**.
The **key** attribute is compared first.

Go further

You can optimize even further:

By default, a **parent** component's **re-rendering** triggers the re-rendering of **all of its children**.

You can make use of the **`shouldComponentUpdate()`** lifecycle method to **skip this re-rendering process when adequate**.



Go further

Don’t re-render a gif if it has the same **`id`** as before!

```
class Gif extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    return this.props.id !== nextProps.id;
  }

  render() {
    if (!this.props.id) {
      return <p>Loading...</p>;
    }

    // [...]
    return <img alt="" src={url} className="gif" [...] />;
  }
};
```

Good to know

By default, **`shouldComponentUpdate()`** returns **`true`**
Don’t try comparing **`this.props`** with **`nextProps`**
(they’ll always be 2 different objects)
Compare **values** (or identifiers) instead

```
// DON’T 👎
shouldComponentUpdate(nextProps, nextState) {
  return this.props !== nextProps;
}
```

```
// DO 👍
shouldComponentUpdate(nextProps, nextState) {
  return this.props.id !== nextProps.id;
}
```

Take away

Under the hood, React has a pretty efficient way of optimising the re-rendering process in the browser.

React builds a **Virtual DOM** and runs a **diff algorithm** to spare memory allocation and rendering time.

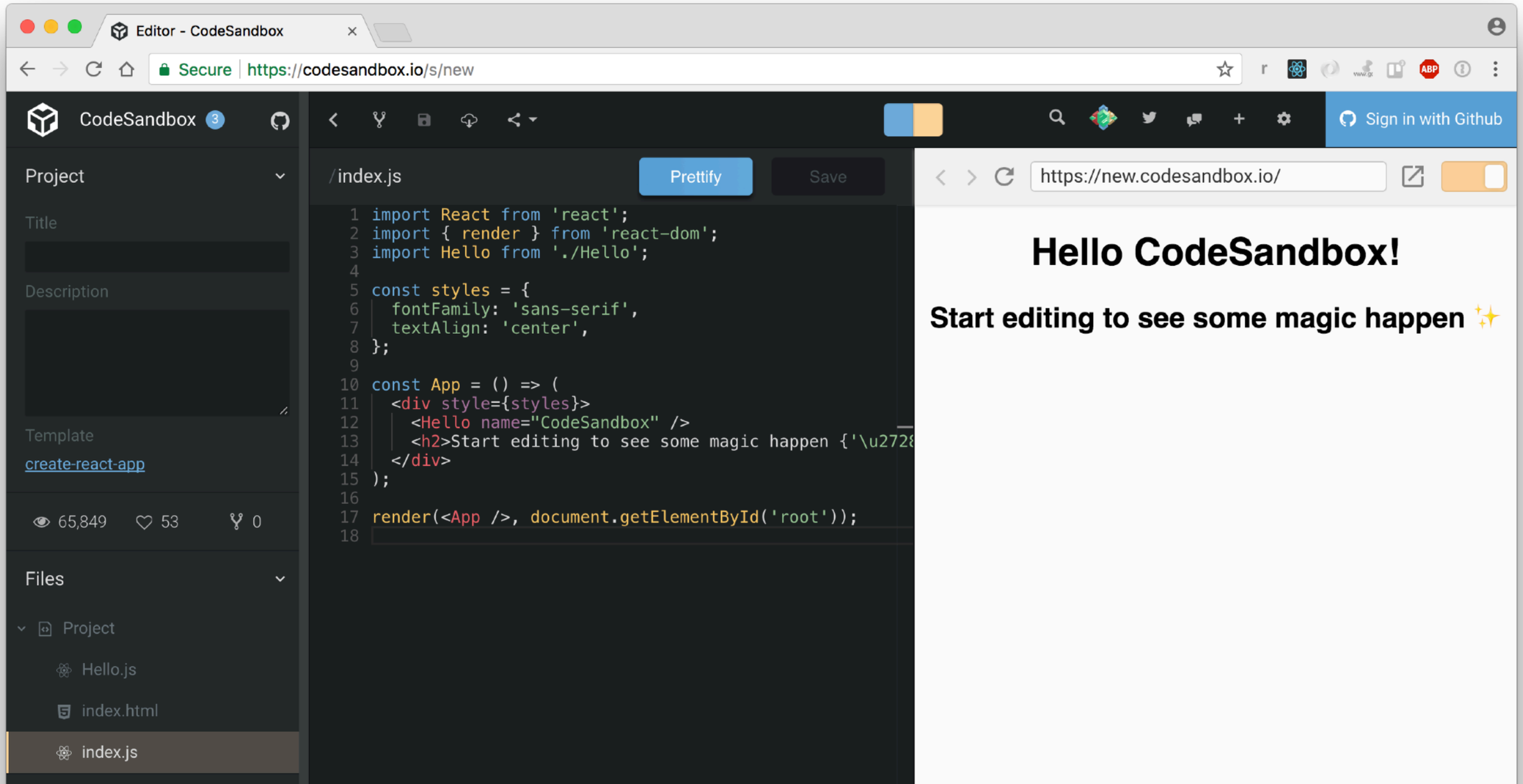
Its efficiency mainly lies in the early comparison of DOM nodes' **types** and **keys**

You can go further and skip this process by overriding the **`shouldComponentUpdate()`** lifecycle method

Tools

Code sandbox

https://codesandbox.io/s/new



create-react-app

A package to generate new React apps out of the box

```
yarn global add create-react-app
# gives you a create-react-app binary
```

```
cd ~/code/<github_username>
create-react-app my-app && cd $_
# creates the app with a given configuration
# cd into the project
```

```
yarn start
# launches webpack-dev-server
# open your browser at http://localhost:3000
```

```
yarn eject
# to override default config -- cannot be undone ⚠️
# will appear in your package.json
```

Production

To deploy your front-end app on **gh-pages**:

```
yarn add gh-pages --dev
# add the module in your project
```

```
webpack -p
# create your production bundle
```

```
gh-pages -d dist
# deploy
```

Go to https://<github_username>.github.io/<project>

Scripts

To enjoy simpler yarn commands:

```
// package.json
// [...]
  "scripts": {
    "start": "webpack-dev-server",
    "deploy": "webpack -p && gh-pages -d dist"
  },
// [...]
```

gives you

```
yarn start  # to start a server on localhost:8080
yarn deploy # to deploy on GitHub pages
```

Your turn!