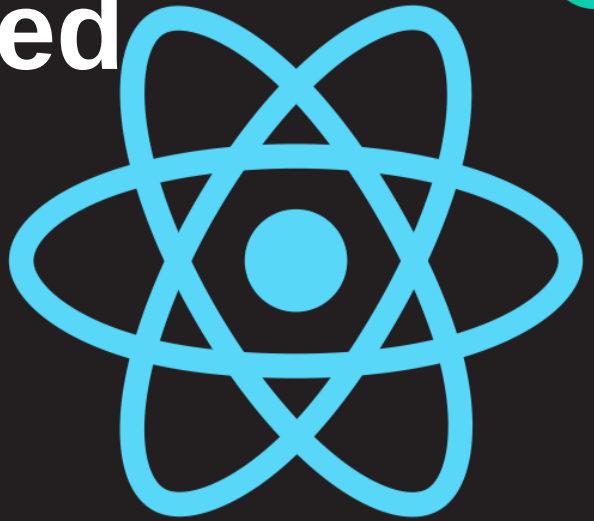
Components Revisited

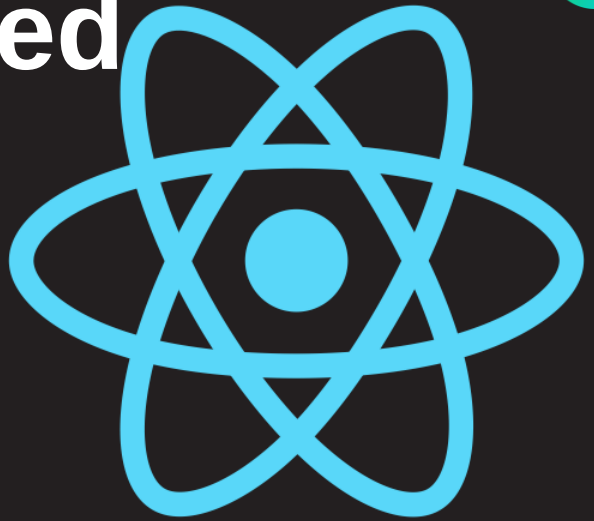


LEARNING OBJECTIVES

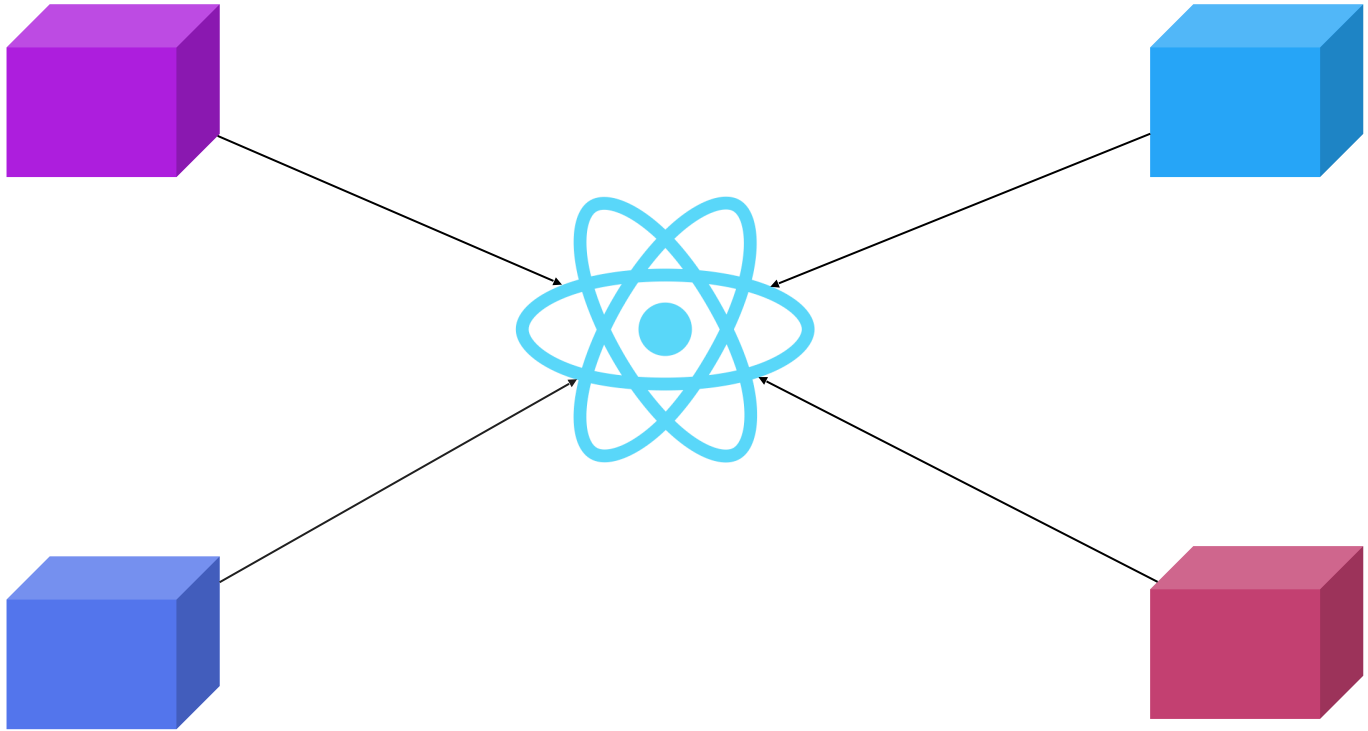
- Understand the lifecycle of a component and how it can be harnessed
- Learn to integrate side effects such as fetching data from an API, in a React component
- Learn to deal with side effects that require clean-up, such as using Timers
- Understand React's SyntheticEvent system
- Learn about managing errors gracefully using Error Boundaries

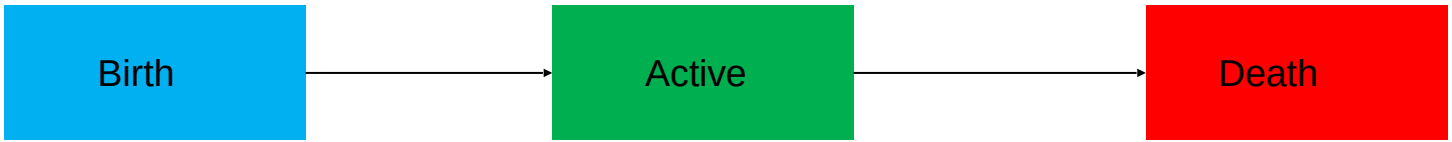
Components Revisited

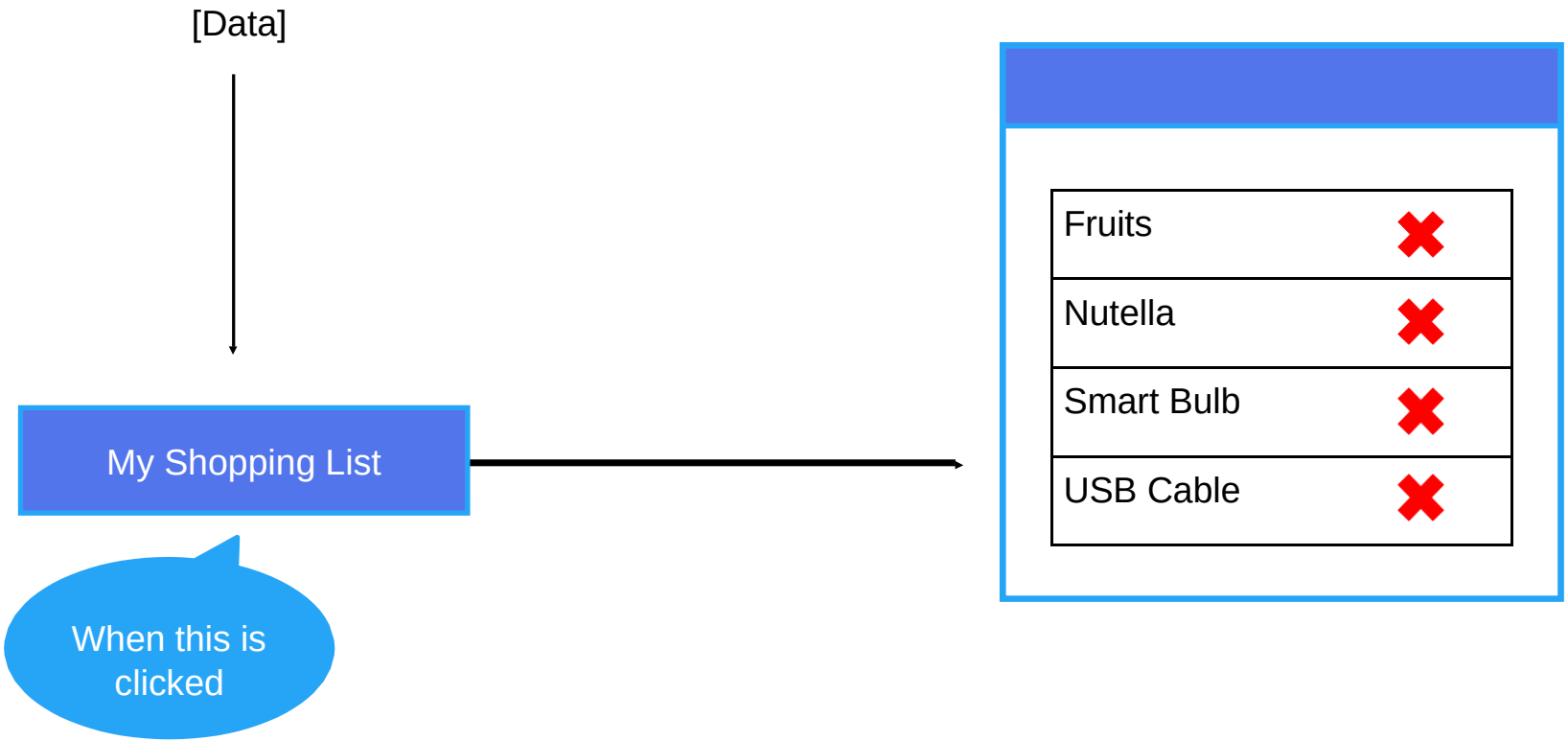
Lifecycle of A Component

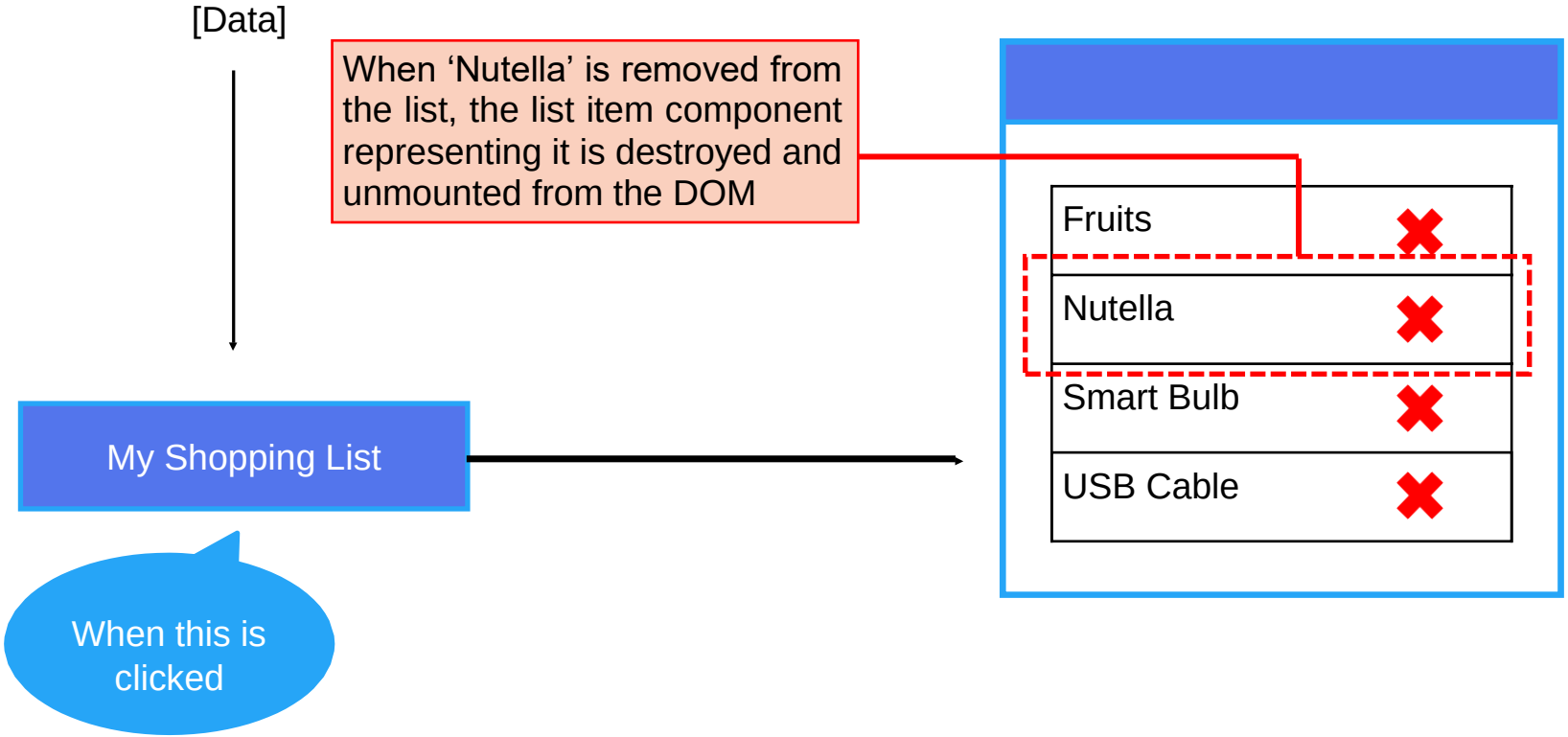


REACT APPLICATION

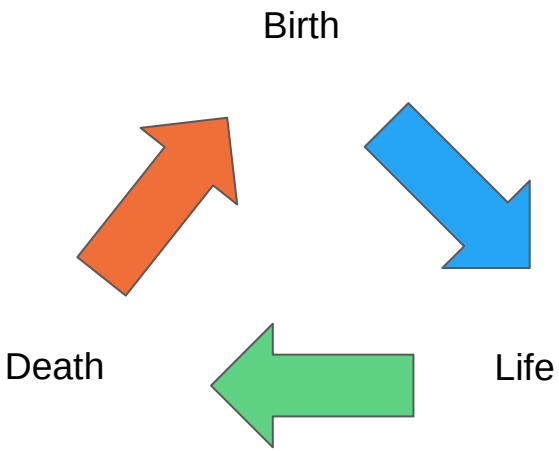



Birth
Active
Death


[Data]
My Shopping List
When this is clicked
Fruits
Nutella
Smart Bulb
USB Cable


[Data]
When ‘Nutella’ is removed from the list, the list item component representing it is destroyed and unmounted from the DOM
Fruits
Nutella
Smart Bulb
USB Cable
My Shopping List
When this is clicked

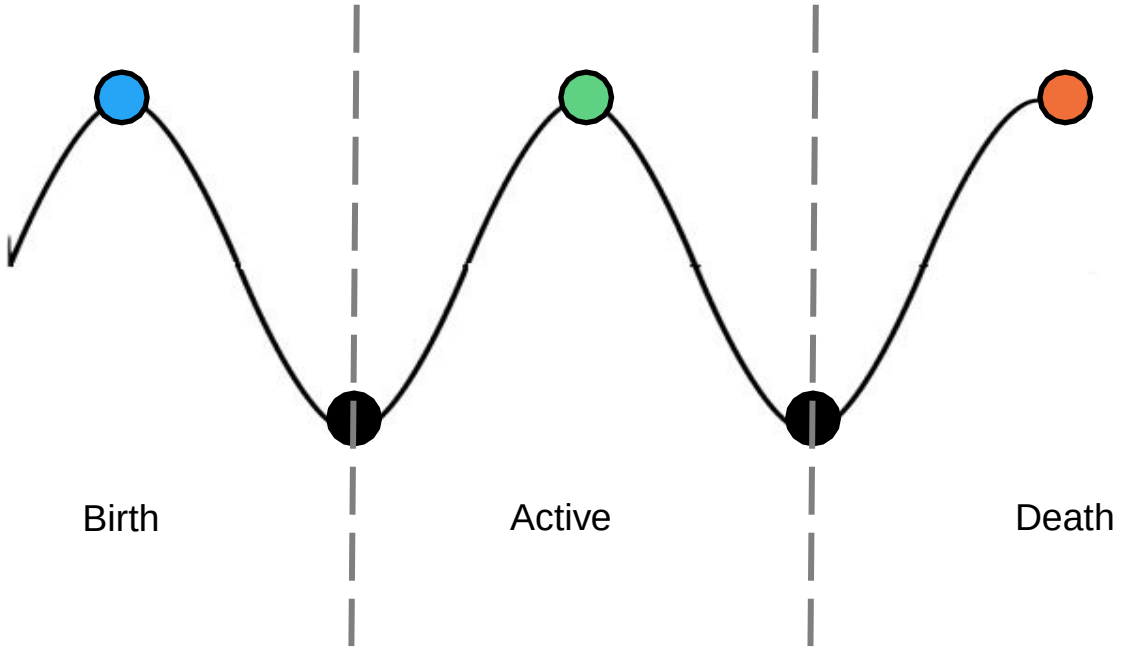
LIFECYCLE OF REACT COMPONENT



Class components provide built-in mechanism for customizing behavior through lifecycle methods.

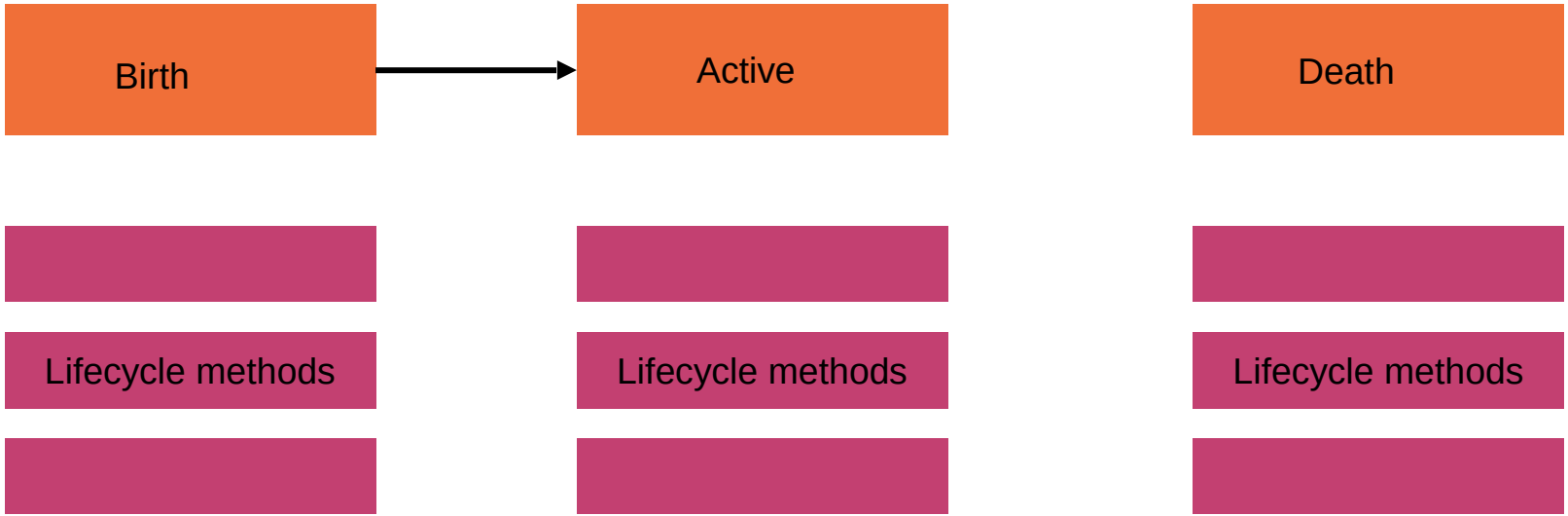
LIFECYCLE OF REACT COMPONENT

- React Milestones

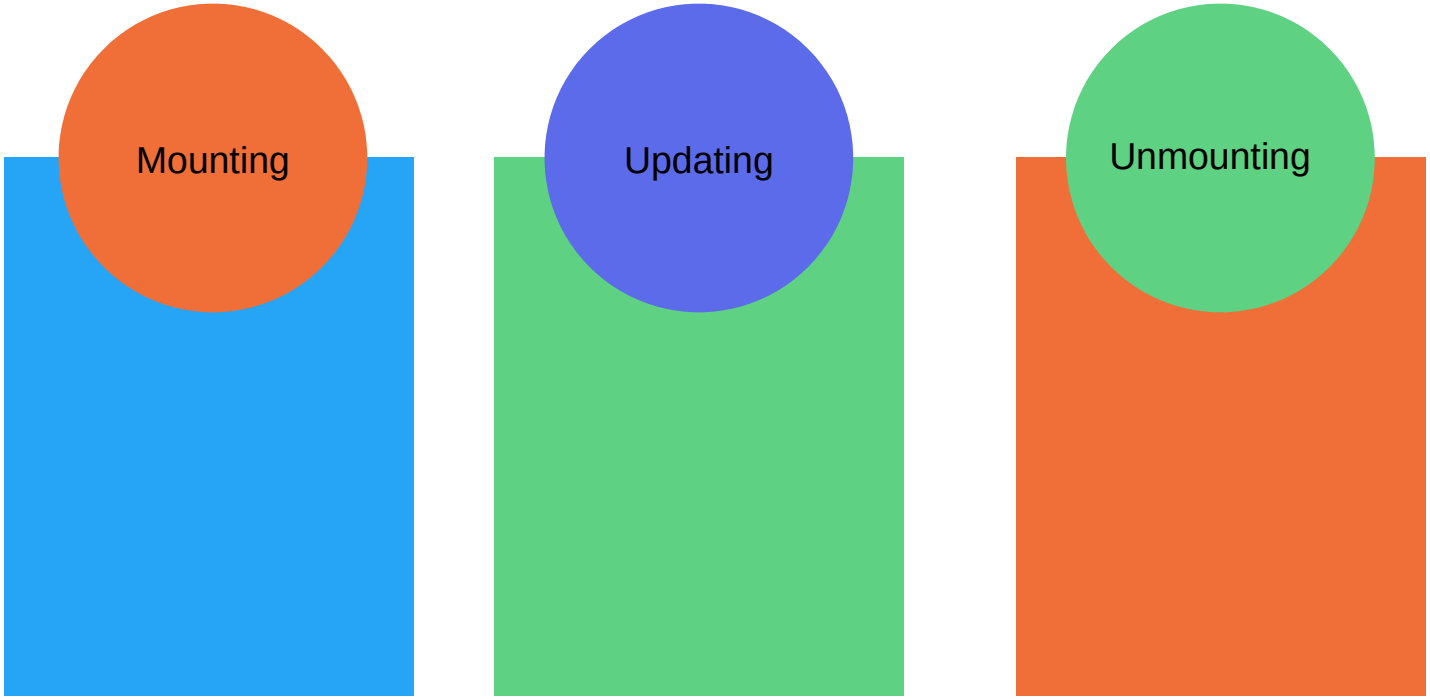


React Milestones are known as Lifecycle Methods

PHASES OF A REACT COMPONENT



PHASES OF A REACT COMPONENT



MOUNTING PHASE

Mounting phase occurs when a React component instance is created and inserted into the DOM.

1. constructor() Method
2. getDerivedStateFromProps() Method
3. render() Method
4. componentDidMount() Method

MOUNTING PHASE

Constructor() Method

- The constructor is called before the component is mounted into the DOM.
- It is used to initialize local state variables and bind class methods such as this example.

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoading: false,
      userName: "",
      shoppingList: []
    }
    this.onClickHandler =
this.onClickHandler.bind(this);
  }
}
```

MOUNTING PHASE

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoading: false,
      userName: "",
      shoppingList: []
    }
    this.onClickHandler =
this.onClickHandler.bind(this);
  }
}
```

Mandatory to call `super(props)`

Inside the constructor method, it is mandatory to call super as we are subclassing from the component class.

MOUNTING PHASE

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoading: false,
      userName: "",
      shoppingList: []
    }
    this.onClickHandler =
this.onClickHandler.bind(this);
  }
}
```

The only place where you can initialize state using `this.state`

In all other lifecycle methods you need to use this.setState().

MOUNTING PHASE

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoading: false,
      userName: "",
      shoppingList: []
    }
    this.onClickHandler =
this.onClickHandler.bind(this);
  }
}
```

If needed, you can also use the constructor to bind instance methods so their ‘this’ context refers to the class.

Binding methods to the context of the class.

MOUNTING PHASE

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoading: false,
      userName: "",
      shoppingList: []
    }
    this.onClickHandler =
this.onClickHandler.bind(this);
  }
}
```

```
class Greeting extends Component {
  state = {
    isLoading: false,
    userName: "",
    shoppingList: []
  };
  onClickHandler = () => {}
}
```

MOUNTING PHASE

`getDerivedStateFromProps()`
Method

- This rare lifecycle method is used only in very special cases
- This lifecycle method is used both in mounting and updating phase
- This lifecycle method allows you to perform state changes based on data in the props

This method is invoked every time a parent component re-renders.

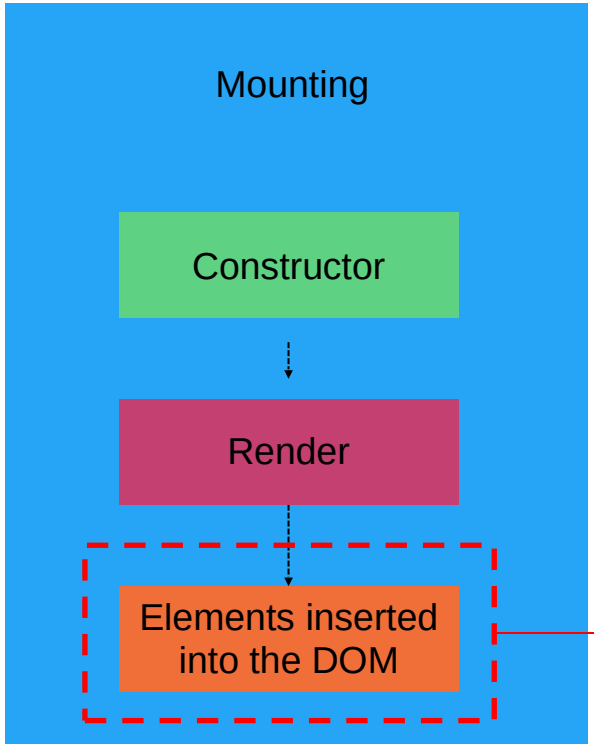
MOUNTING PHASE

`render()` Method

- This lifecycle method is only needed if you're building class components and its purpose is to return React element
- This is also why the render() method should stay pure
- The render() method must never modify the state and should always return the same predictable result.

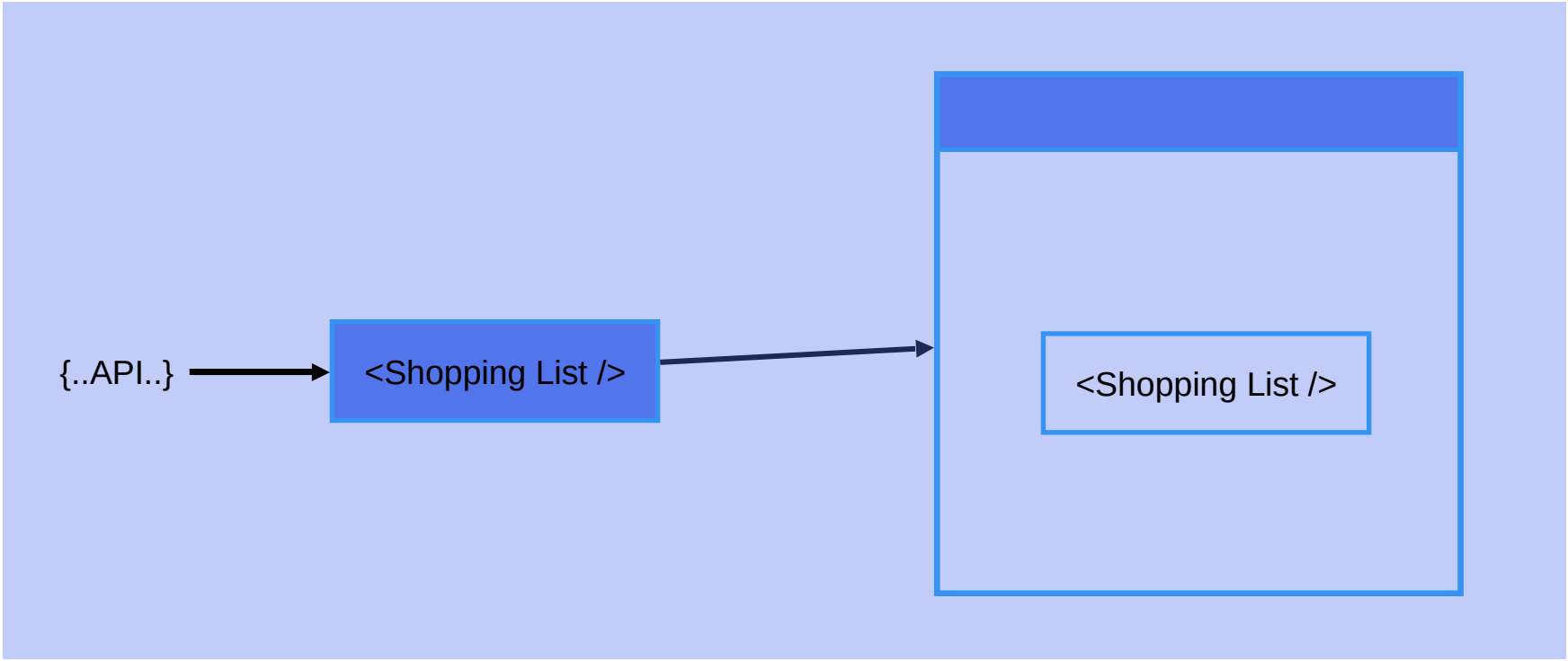
```
class Card extends Component {
  state = {
    fullName: "Johnny Doe",
    designation: "VP - Sales"
  };
  render() {
    return <div className="v-card">
      <div className="name">{this.state.fullName}</div>
      <div className="designation">{this.state.designation}</div>
    </div>
  }
}
```

MOUNTING PHASE



After the `render()` method, its initial set of contents are inserted into the DOM

MOUNTING PHASE



UPDATING PHASE

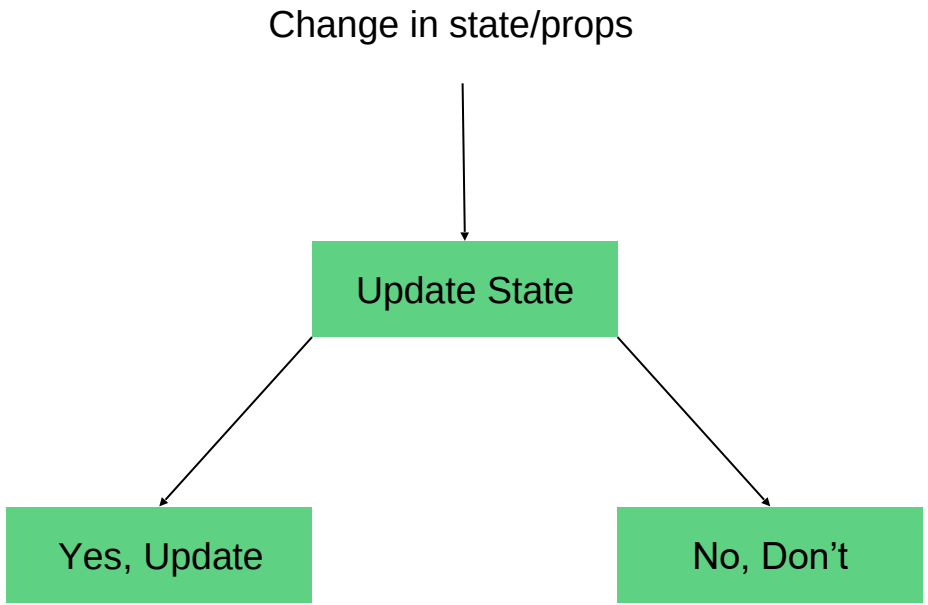
Updating phase occurs when we interact with the component and it re-renders.

- This interaction includes receiving updated data through
  - Props
  - Update to the state
- In both cases, the updating phase begins with an invocation to the `getDerivedStateFromProps()` function.

UPDATING PHASE

`getDerivedStateFromProps()` Method

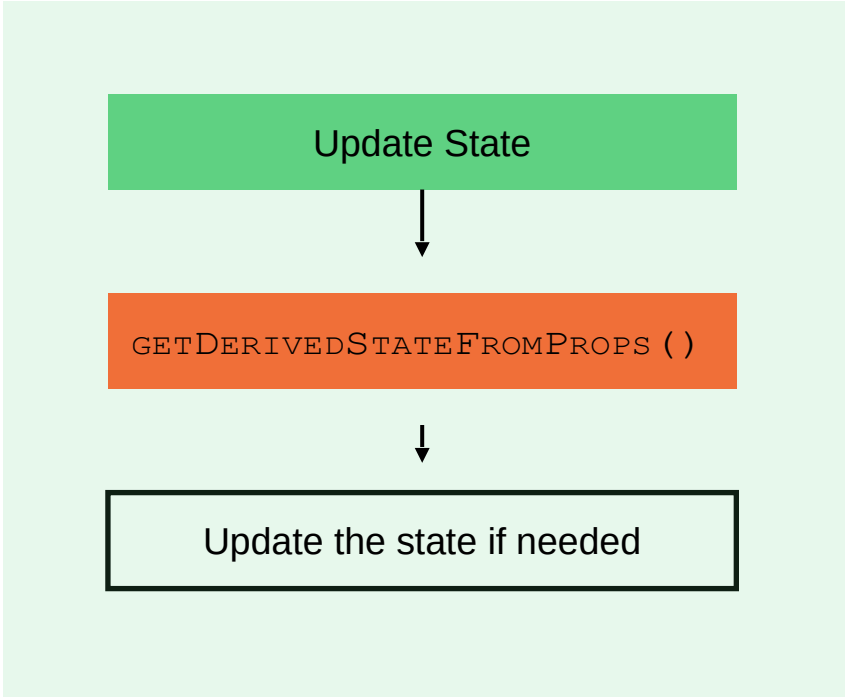
This method allows you to decide whether you want the state to be updated or not, based on the changes to data in the props.



UPDATING PHASE

The next time `getDerivedStateFromProps()` method is called in the updating phase, we can compare the data in the state with that in the props and can update the state if needed.

This helps you implement state changes that will occur only if the data in the props differs from what it was earlier.



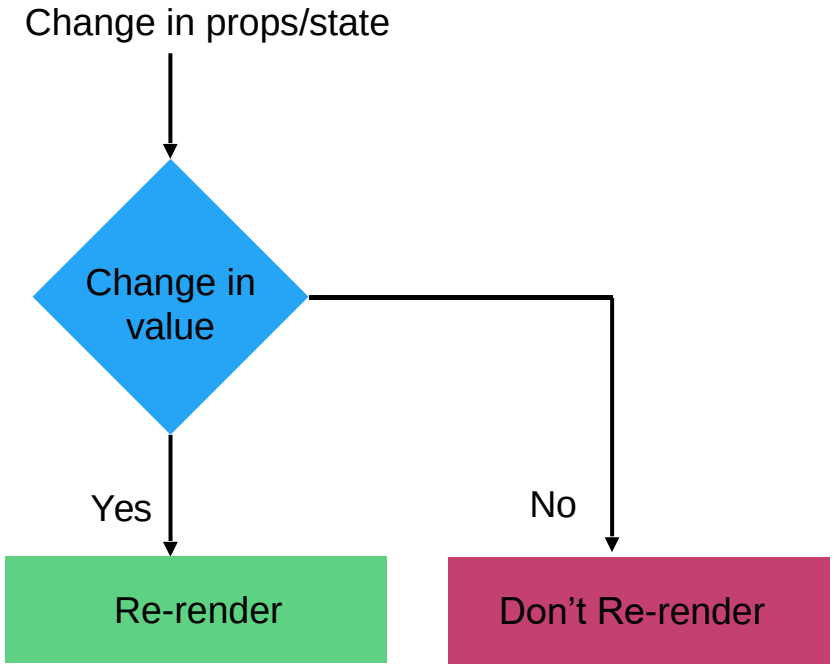
UPDATING PHASE

- In Updating phase, every update to the parent component brings an update to child components too, even if the data in the props continues to be the same.
- In some cases, you may not want to update the state in a child component if the data in the props continues to be the same as before and that's where you can use `getDerivedStateFromProps()` function.
- After this lifecycle hook, the `shouldComponentUpdate()` method can be invoked.

UPDATING PHASE

`shouldComponentUpdate() Method`

- This function lets you decide if you want the component to re-render based on changes to data in state or props
- This helps in prevention of DOM reconciliation for components if data in props and state has not changed thereby boosting performance.



UPDATING PHASE

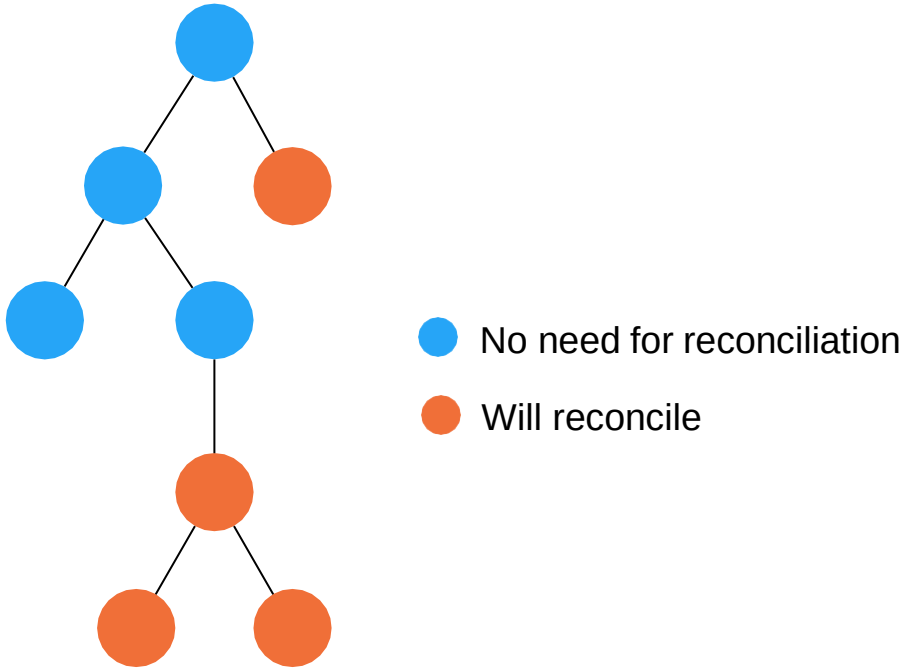
```
class Notification extends
Component {
  shouldComponentUpdate =
(nextProps, nextState) => {
    if (this.props.message !==
nextProps.message) {
      return true;
    }

    return false;
  }
  render() {
    return <div
className="notification">{this.prop
s.message}</div>
  }
}
```

By comparing existing data in props/state with new and incoming data during the updating phase, we can prevent re-renders

Returning true will cause the component to re-render and returning false will exempt the component from the reconciliation process

UPDATING PHASE

- In large applications where components are deeply nested, the efficiency and performance can be improved by using `SHOULDCOMPONENTUPDATE()` method as it prevents instances from re-rendering just because their parent component was refreshed.
- Now, instead of implementing this method for optimization, React also offers something known as a Pure Component which automates this process.



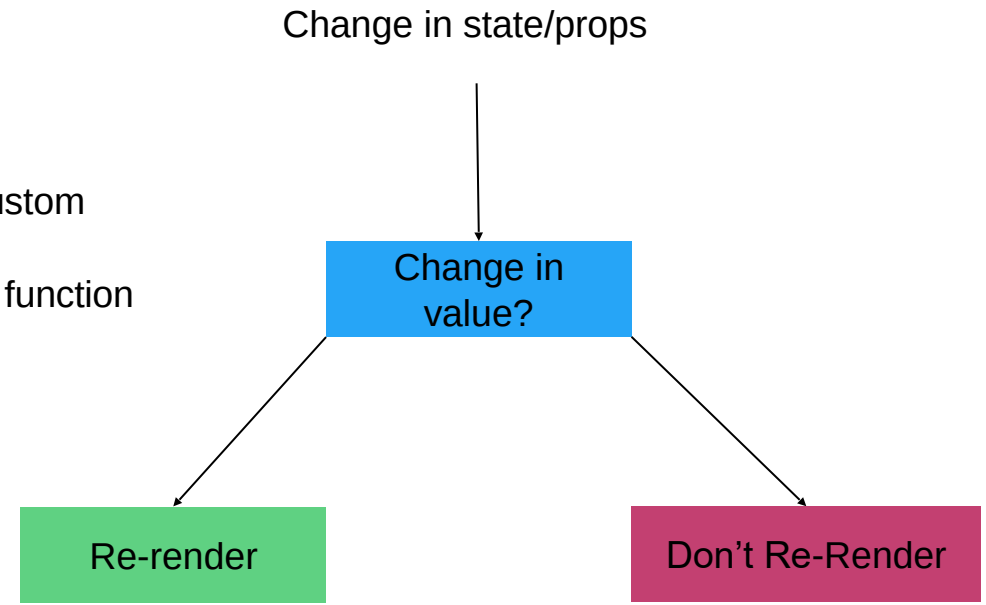
UPDATING PHASE

- A Pure Component has built-in `shouldComponentUpdate()` and doesn't need to be manually set up as in the case of a component instance.
- The build-in `shouldComponentUpdate()` performs a shallow comparison of state and props and offer no customization of behaviour.
- A Pure Component will prevent its subtrees from re-rendering.

```
import React, {PureComponent} from
"react";

class Notification extends
PureComponent {
  render() {
    return <div
className="notification">{this.pro
ps.message}</div>
  }
}
```

UPDATING PHASE

For complex data structures and custom behavior,
use `shouldComponentUpdate()` function with a standard Component



UPDATING PHASE

```
class Notification extends
Component {
  shouldComponentUpdate =
(nextProps, nextState) => {
    if (this.props.message !==
nextProps.message) {
      return true;
    }

    return false;
  }
  render() {
    return
<div
className="notification">{this.pro
ps.message}</div>
  }
}
```

- At this point in updating phase, we've decided if we want to re-render or not.
- Then the render method is called which returns a revised set of nodes to be updated in DOM.
- Right before the changes are updated in DOM, we get to use `getSnapshotBeforeUpdate()` function.

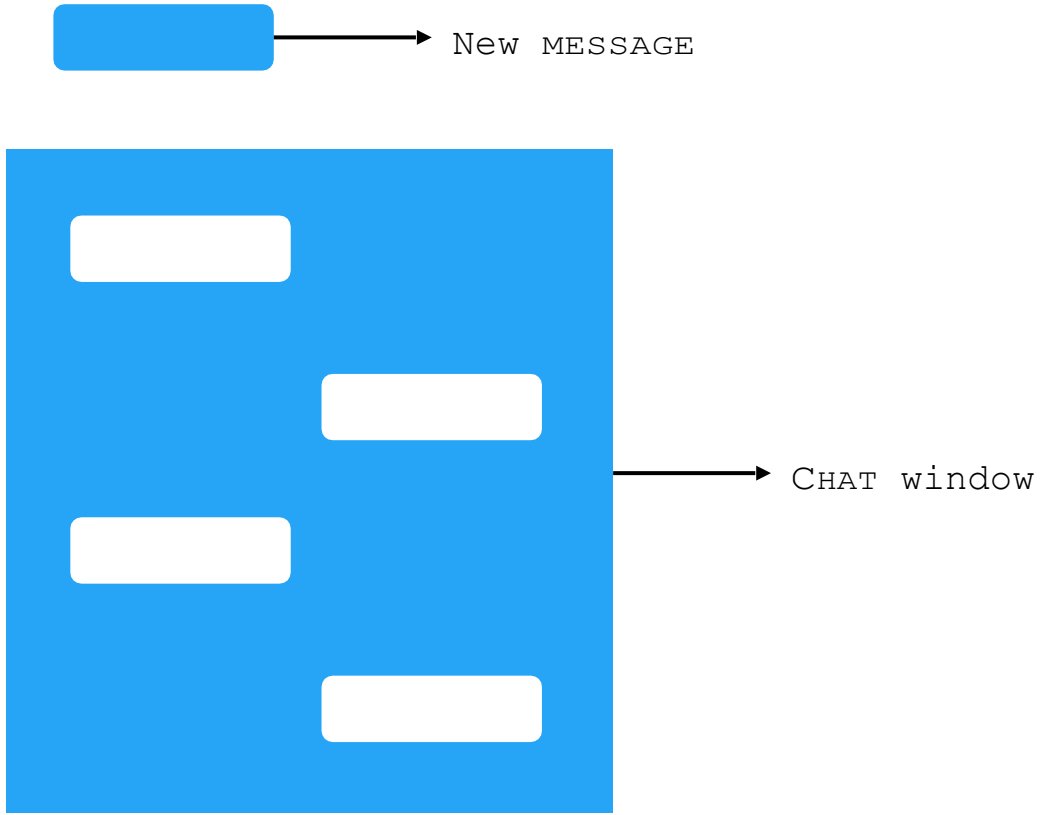
DOM

UPDATING PHASE

`getSnapshotBeforeUpdate()` function

- This function lets you compare current and previous values in state and props but this time we have access to DOM nodes before React implements them.
- This means you can access and update DOM properties using React references.

UPDATING PHASE



UPDATING PHASE

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    // Are we adding new items to the list?
    // Capture the scroll position so we can adjust scroll later.
    if (prevState.messages.length < this.state.messages.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }
```

```
componentDidUpdate(prevProps, prevState, snapshot) {
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }
```

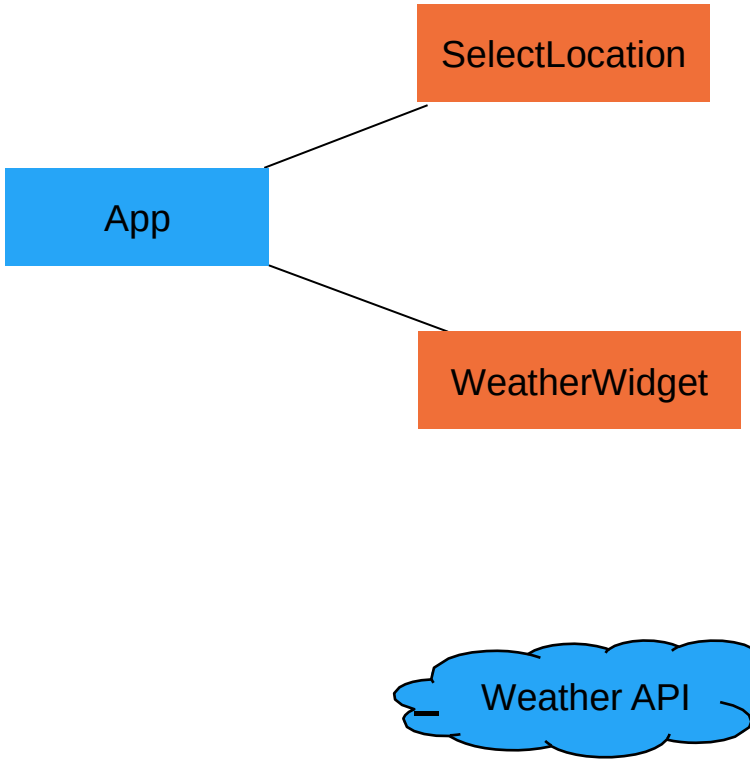
Implementing the scroll

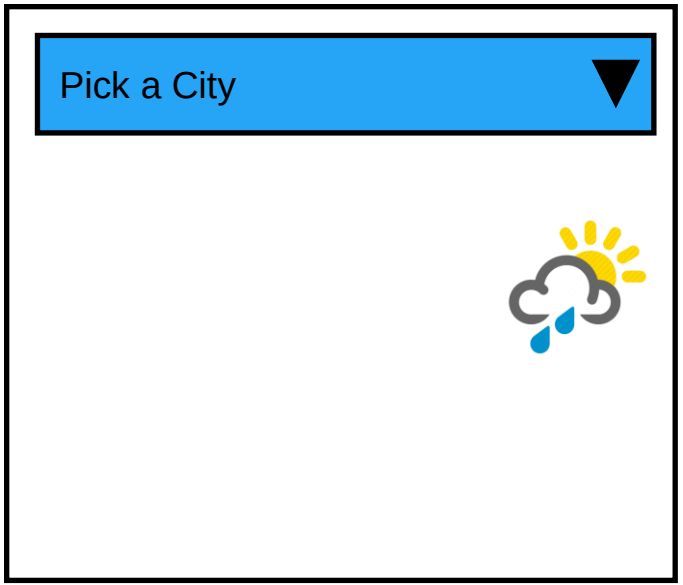
UPDATING PHASE

ComponentDidUpdate() lifecycle hook

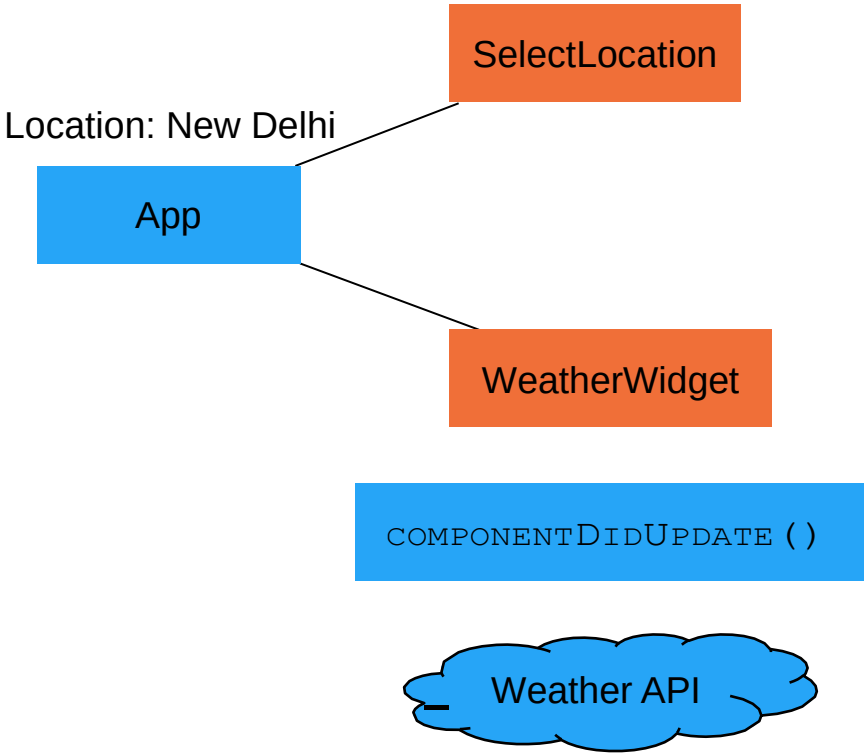
- This function is useful when a component needs to decide if it needs to perform side effects such as querying an API for more data on the basis of change in state or props.
- This method is not invoked on the first render of a component and is only available when a component updates.

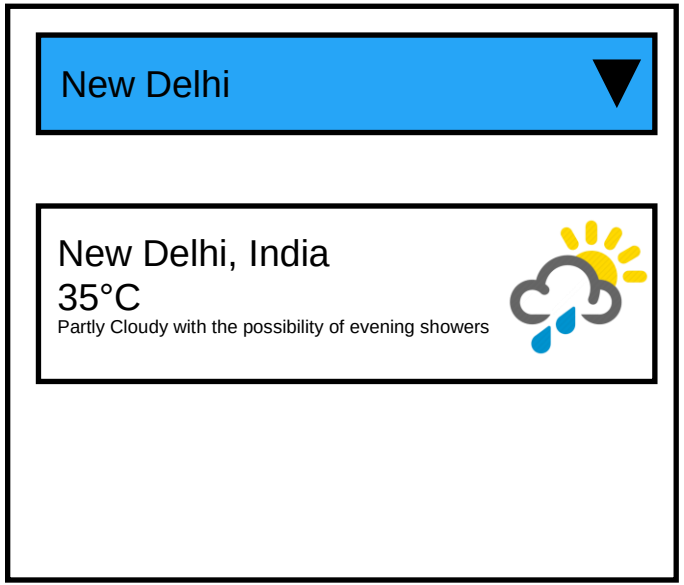
UPDATING PHASE





UPDATING PHASE





UPDATING PHASE

```
class WeatherApp extends Component {
  componentDidUpdate() {
    // Query Weather API and update
the state
  }
  render() {
    // Render the UI
  }
}
```

This will run perpetually as the component is updated/re-rendered!

UPDATING PHASE

```
class WeatherApp extends Component {
  componentDidUpdate(prevProps,
prevState) {
    if (prevProps.location !==
this.props.location) {
      // Query Weather API and
update the state
    }
  }
  render() {
    // Render the UI
  }
}
```

Perform side-effects only if the value in the location prop is different from what it was before the update.

UPDATING PHASE

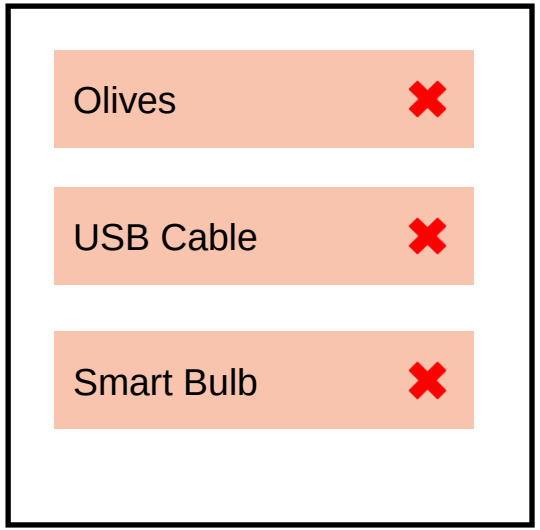
Unmounting Phase is the final phase that the component goes through.

- This phase offers one single method, known as componentWillUnmount() which occurs right before the compoent is destroyed
- This method is useful for cleanup such as removing timer instances and detaching or unsubscribing from real time data sources such as Websockets, when a component is removed
- This method also lets you free up memory when the component is destroyed and unmounted

UPDATING PHASE

State

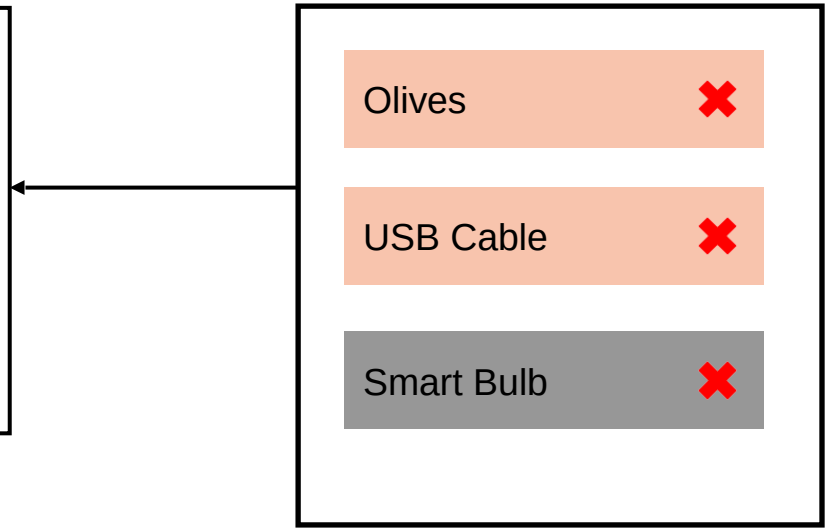
```
[{
  prodId: 0102,
  prodName: "Olives"
}, {
  prodId: 0103,
  prodName: "USB Cable"
}, {
  prodId: 0104,
  prodName: "Smart Bulb"
}]
```



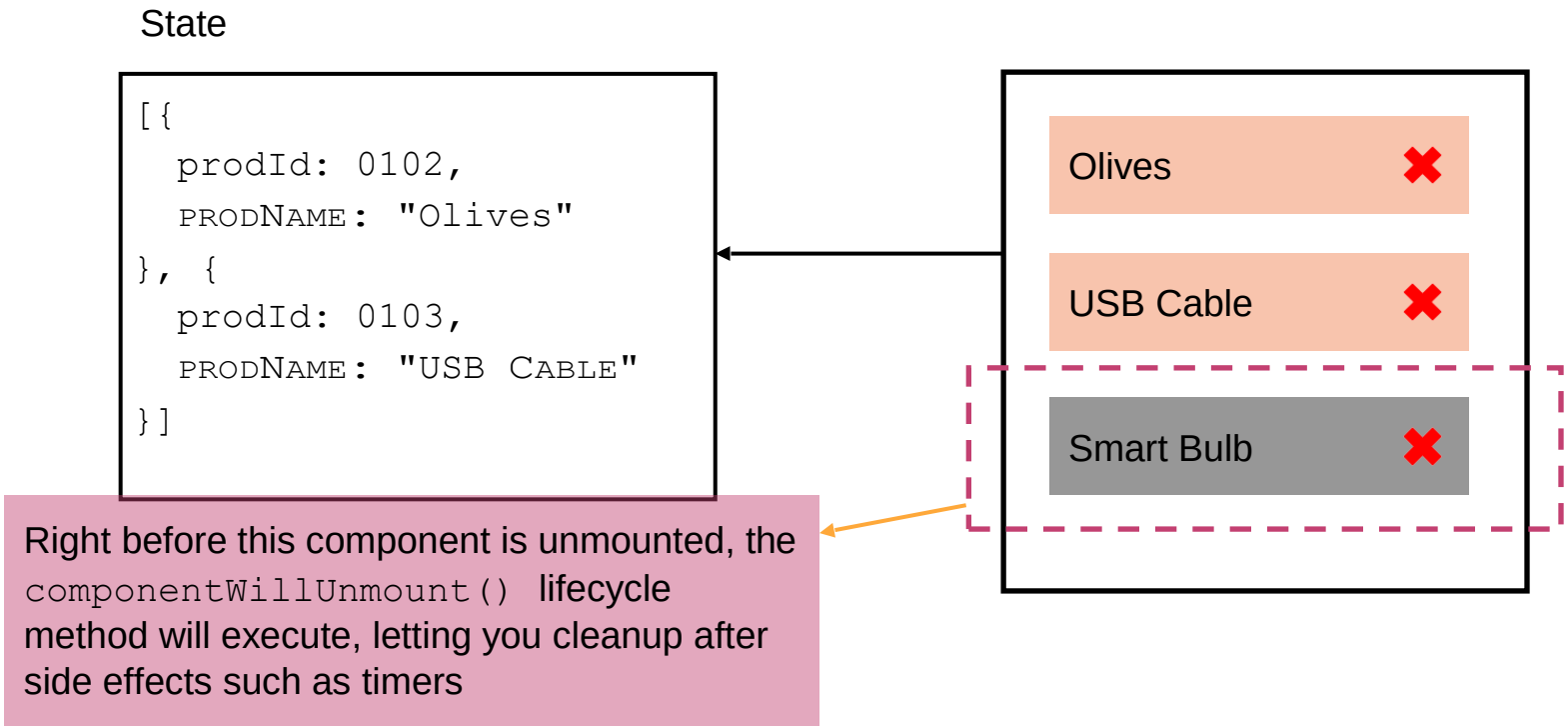
UPDATING PHASE

State

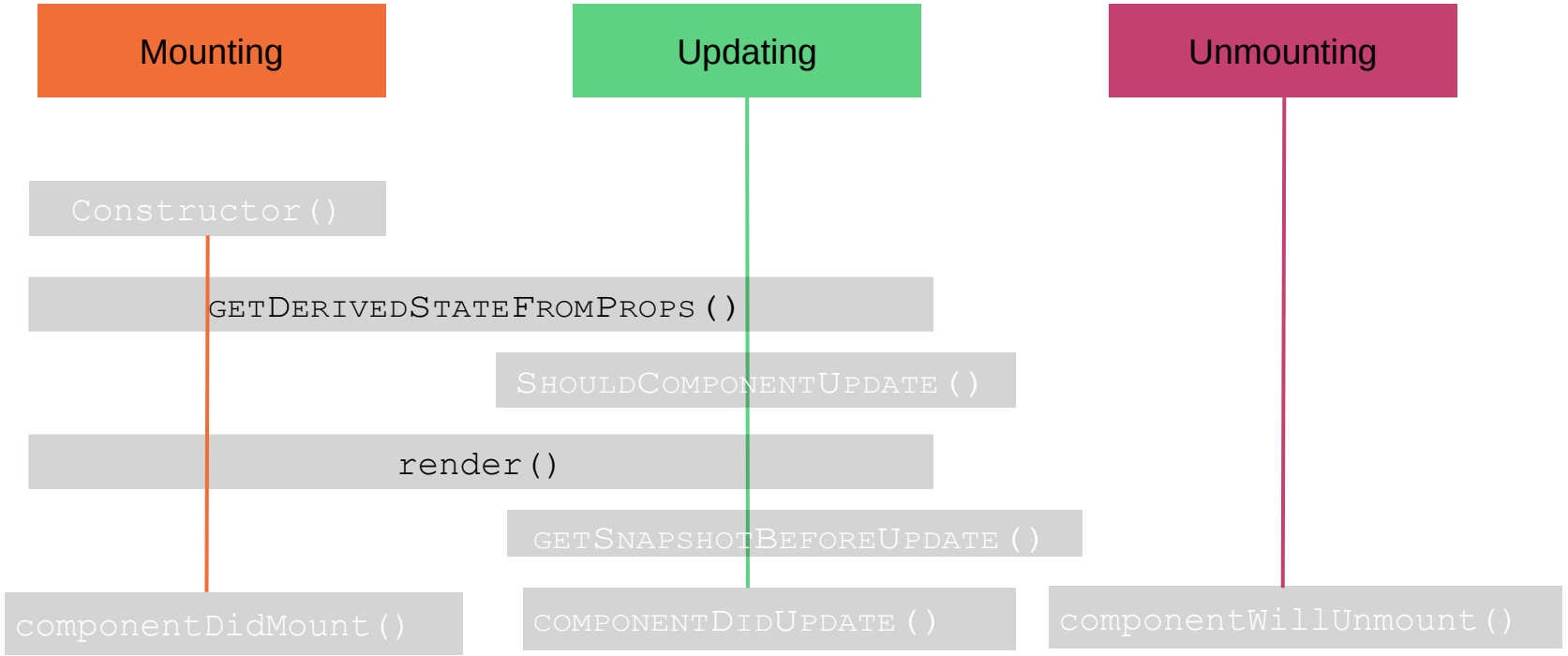
```
[{
  prodId: 0102,
  prodName: "Olives"
}, {
  prodId: 0103,
  prodName: "USB Cable"
}]
```



UPDATING PHASE



LIFECYCLE OF REACT COMPONENT



LIFECYCLE METHODS

How and where would you perform side effects such as querying a third-party API, in a class component?

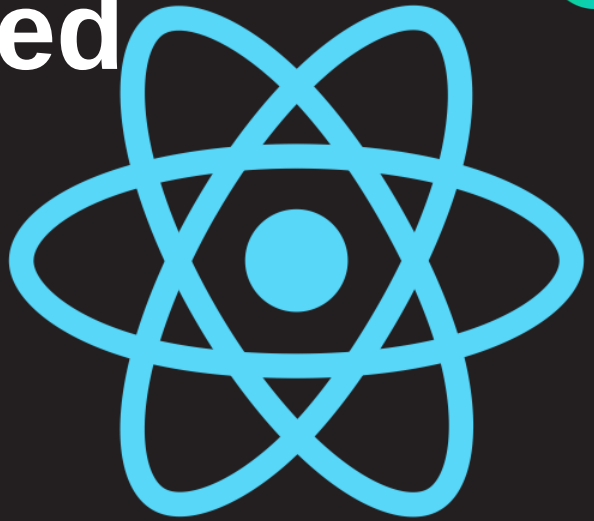
```
const NOTIFICATION = ({MESSAGE}) => {
  return <div
CLASSNAME="NOTIFICATION">{MESSAGE}</
div>
}
```

Hooks API

- Hooks allow you to incorporate a lifecycle-like workflow with function components
- They’re not exactly lifecycle methods and require a slightly different approach and thought process

Components Revisited

Side Effects & Lifecycle



SIDE EFFECTS

How and where would you perform side effects such as querying a third-party API, in a class component?

SIDE EFFECTS

- There are two kinds of side effects that we have to deal with.
- Side effects that do not require cleanup
- Side effects that require cleanup

SIDE EFFECTS

Do not require cleanup

Fetching data from an API using an HTTP request

Do not require cleanup

Using timers such as setInterval()

Using subscription-based services such as when using Websockets

SIDE EFFECTS

SIDE EFFECTS

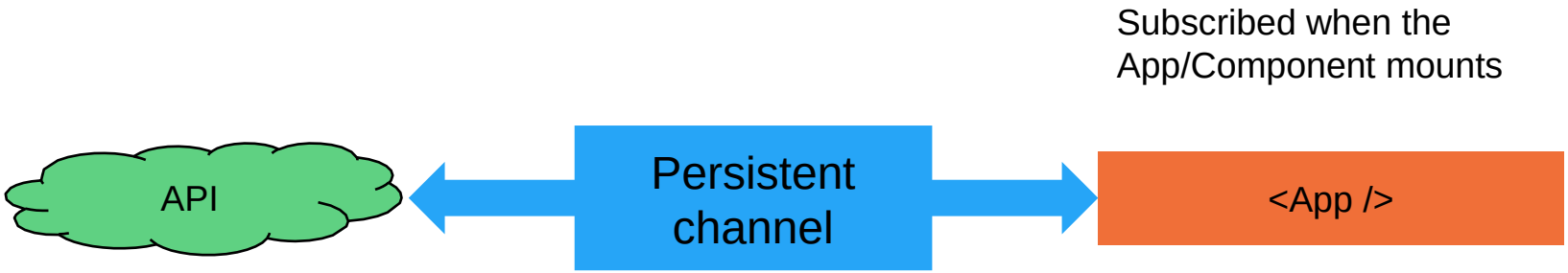
Unlike WebSocket based services, we're talking about regular HTTP based services that do not require a persistent connection or subscription.

SIDE EFFECTS

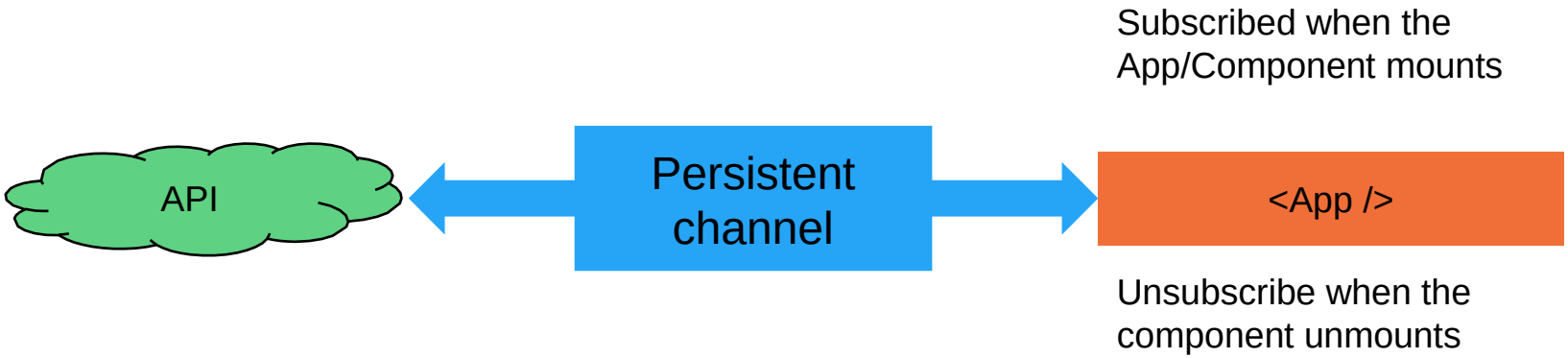
Unlike WebSocket based services, we're talking about regular HTTP based services that do not require a persistent connection or subscription.

No need to unsubscribe or perform a cleanup to prevent memory leaks.

SIDE EFFECTS



SIDE EFFECTS



SIDE EFFECTS

How to implement side effects which do not require any cleanup, such as fetching data from an external API.

SIDE EFFECTS

`componentDidMount()`

Side effects when the component mounts for the first time

`componentDidUpdate()`

Side effects when the component updates such as when data in the props update

SIDE EFFECTS

```
class Weather extends Component {
  state = {
    temperature: 0,
    humidity: 0,
    conditions: "",
    feelsLike: ""
  };
  componentDidMount = () => {
    weatherApi(this.props.location).then(data => {
      this.setState({
        temperature: data.temp,
        humidity: data.humidity,
        conditions: data.conditions,
        feelsLike: data.feels
      });
    });
  };
  render() {
    return <div>...</div>
```

- `componentDidMount()` function, is called after our component has been mounted into the DOM for the first time.
- You can query an API to fetch data and update the state.
- You can also subscribe to web sockets here.
- You can also access DOM elements using refs.

Hands-On

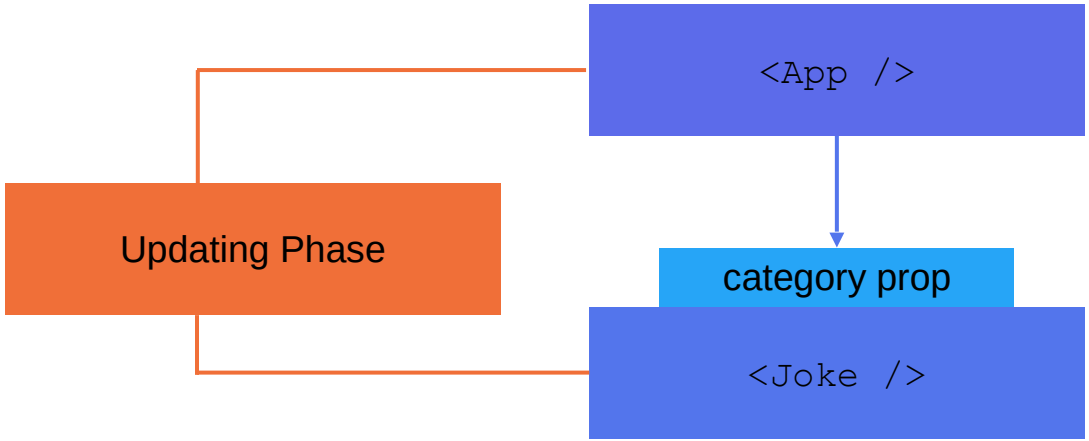
SIDE EFFECTS

With class components and side effects that do not require cleanup, you can use the `componentDidMount()` lifecycle method to perform API or network requests among other things.

```
componentDidMount = () =>

this.getJoke();
```

Hands-On

SIDE EFFECTS



When you change the category of jokes, the subtree re-renders

MOUNTING PHASE componentDidMount = () => this.getJoke;

Since componentDidMount() doesn't run in the updating phase, we have no way to fetch a new joke based on a chosen category.

SIDE EFFECTS

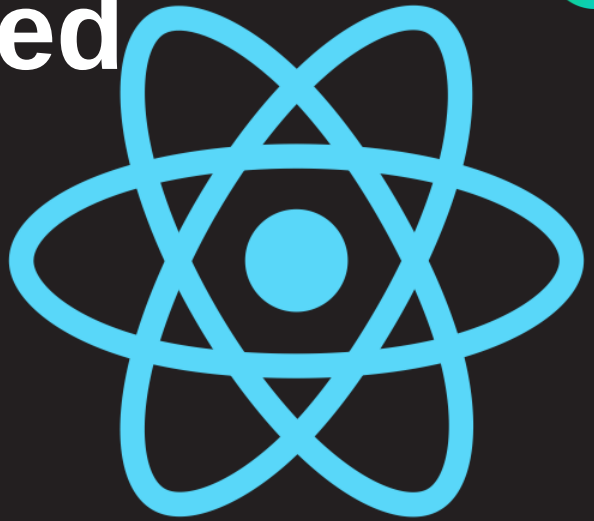
So how do we run Side Effects in the ‘Updating Phase’ of the component?

SIDE EFFECTS

The answer is `componentDidUpdate()` lifecycle hook.

Components Revisited

Managing Clean-up



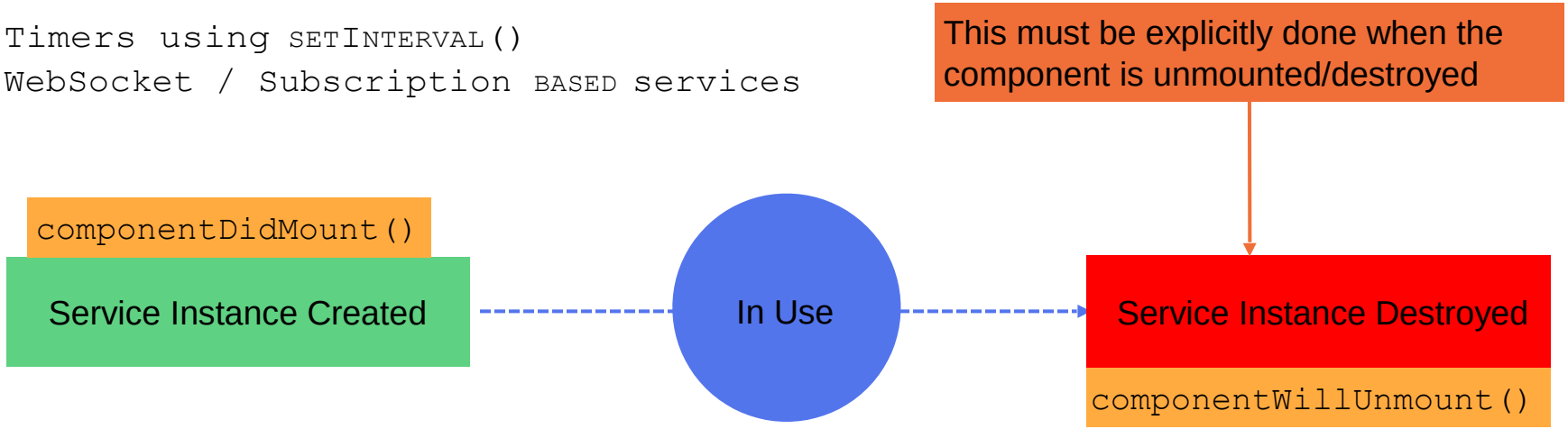
MANAGING CLEANUP

```
class Joke extends Component {
  state = {
    joke: {},
    isLoading: false
  };
  getJoke = async category => {…};
  componentDidMount = () => this.getJoke(this.props.category);
  componentDidUpdate = prevProps => {
    if (this.props.category !== prevProps.category) {
      this.getJoke(this.props.category);
    }
  };
  render() {
    return (
      <>
        <div className={this.state.isLoading ? "title title-pulse" : "title"}>
          Joke Machine
        </div>
        <div className="joke-panel">
          <div className="joke-setup">{this.state.joke.setup}</div>
          <div className="joke-punchline">{this.state.joke.punchline}</div>
        </div>
      </>
    );
```

Fetching data from an API

MANAGING CLEANUP

- Timers using setInterval()
- WebSocket / Subscription based services



MANAGING CLEANUP

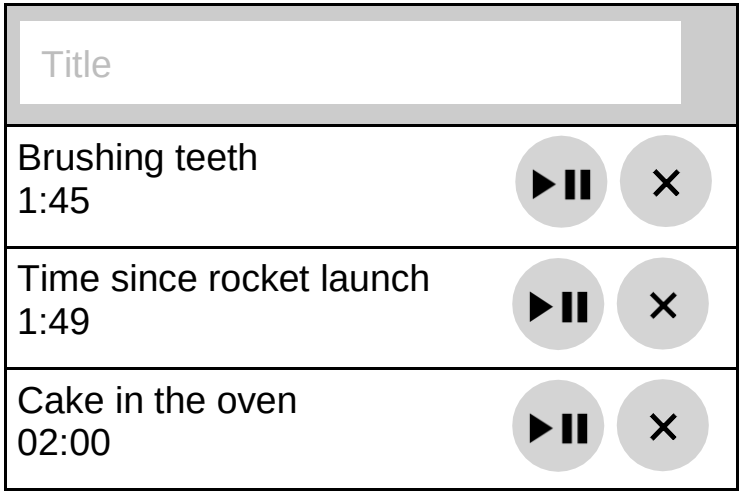
```
class GetStockRates extends Component {
  state = {
    rates:[]
  }
  componentDidMount() {
    // Subscribe to timers, realtime data services
  }
  componentWillUnmount() {
    // Teardown and unsubscribe
  }
  render() {}
}
```

`ComponentWillUnmount()`

- Unsubscribe from services
- Detach event listeners
- Remove timers

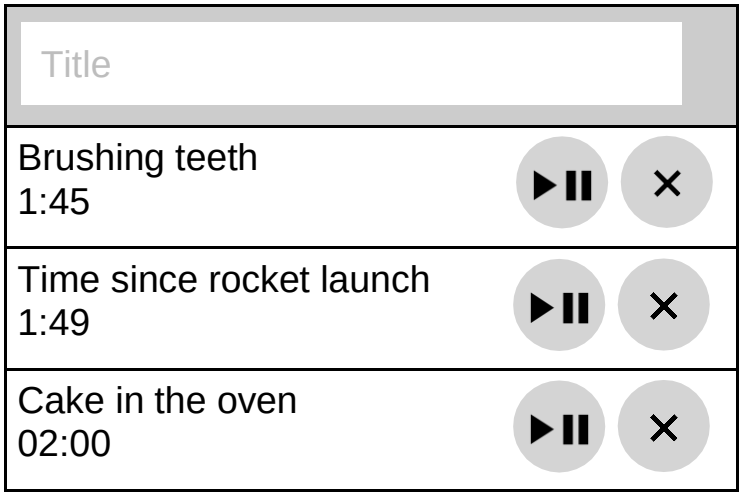
Hands-On

MANAGING CLEANUP



MANAGING CLEANUP

```
componentDidMount = () => {
    this.timerInstance =
setInterval(() => {
      if (!this.state.isPaused) {
        this.setState({
          seconds:
this.state.seconds + 1
        });
      }
    }, 1000);
  };
componentWillUnmount = () =>
clearInterval(this.timerInstance);
```

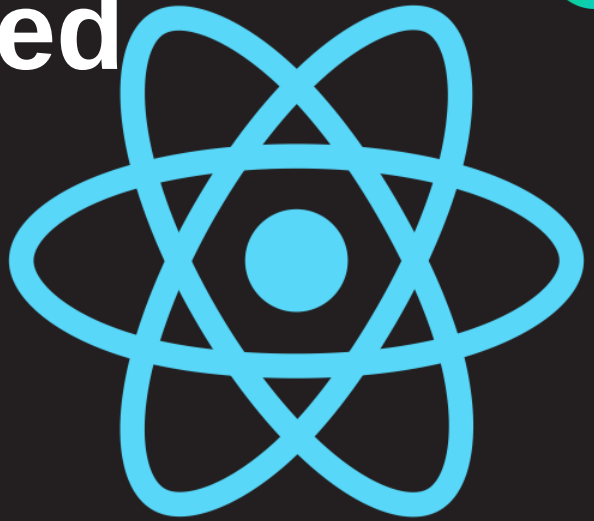


MANAGING CLEANUP

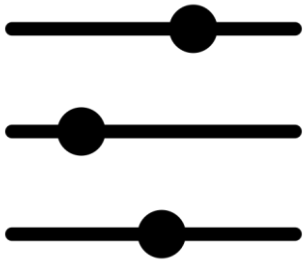
Function Components can achieve similar functionality as class components using the Hooks API.

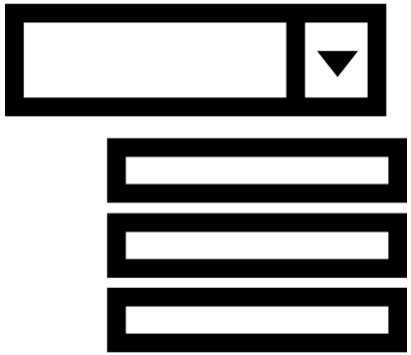
Components Revisited

Events

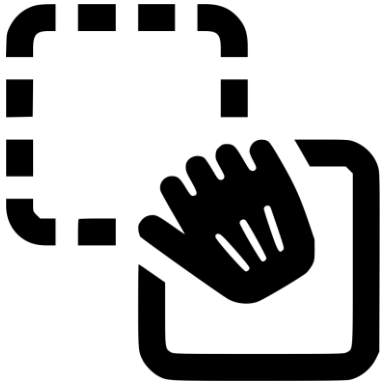


INTERACTION AND EVENTS





INTERACTION AND EVENTS



REACT EVENTS

Handling events is an essential feature of an application.

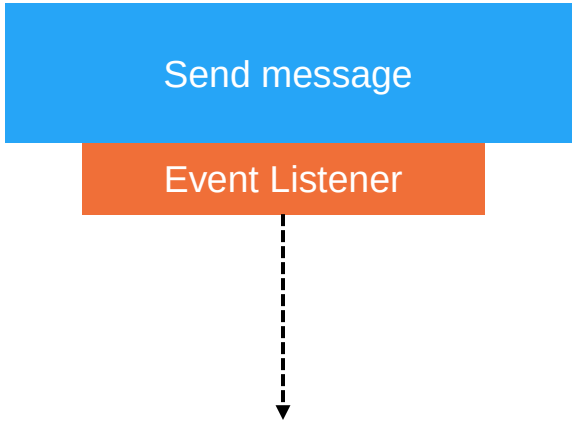
Implements an event driven model

- Browsers
- Node.js

REACT EVENTS

- Events are actions that are fired when something happens, such as when a user clicks on a button or types in an input field.
- These actions can then be used to run functions that perform a certain task.

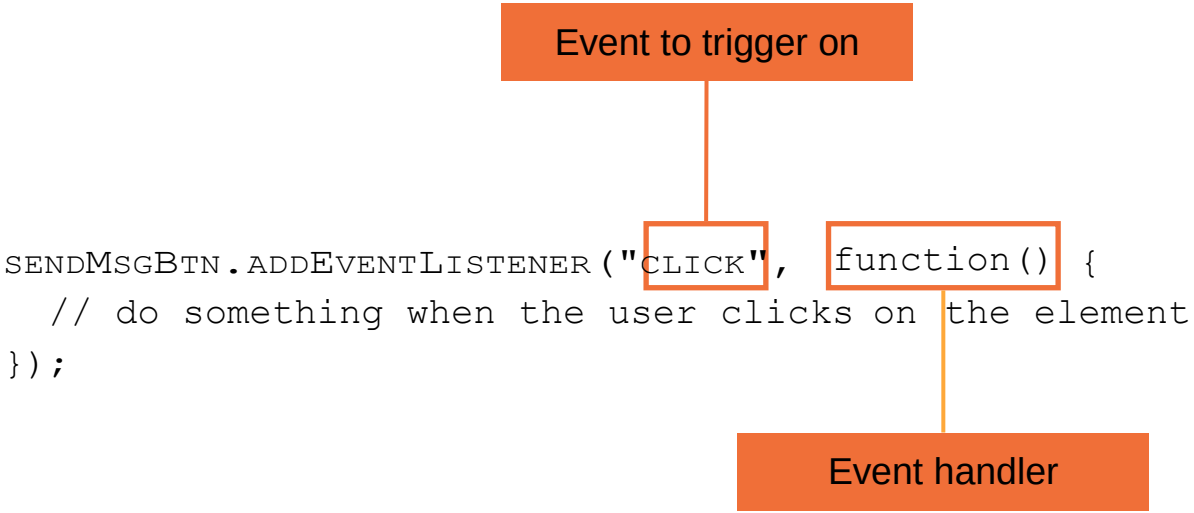
REACT EVENTS




```
eventHandler() {
  // do something when the event occurs
}
```

Events are intercepted by event listeners, which then invoke event handler functions

REACT EVENTS



REACT EVENTS

```
inputFld.addEventListener("keyup", function() {
  // do something when the user presses & releases a key
})
```

REACT EVENTS

```
inputFld.addEventListener("keyup", function(event) {
  if (event.keyCode === 13) {
    // the user pressed & released the enter key
  }
})
```

REACT EVENTS

Using event.target.value to fetch the contents of an input field

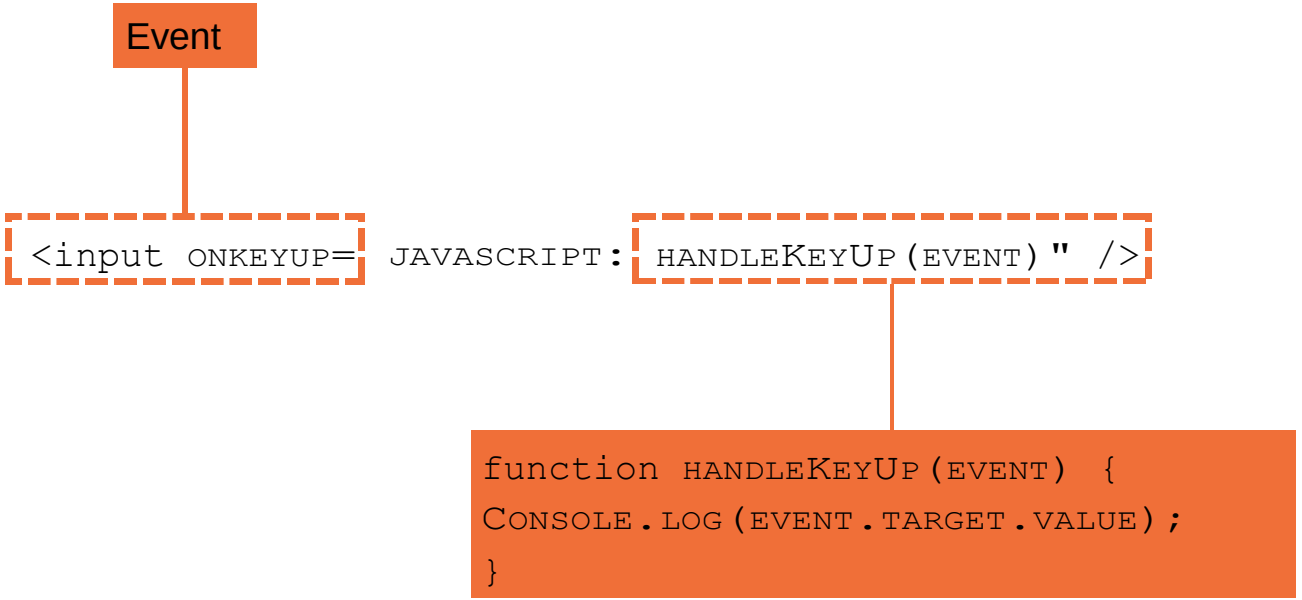
```
const inputFld =
document.getElementById("inp");


inputFld.addEventListener("keyup",
function(event) {
  console.log(event.target.value); //
Hello there...
});
```

```
Hello there…
```

REACT EVENTS

HTML also offers a rudimentary event handling mechanism

REACT EVENTS

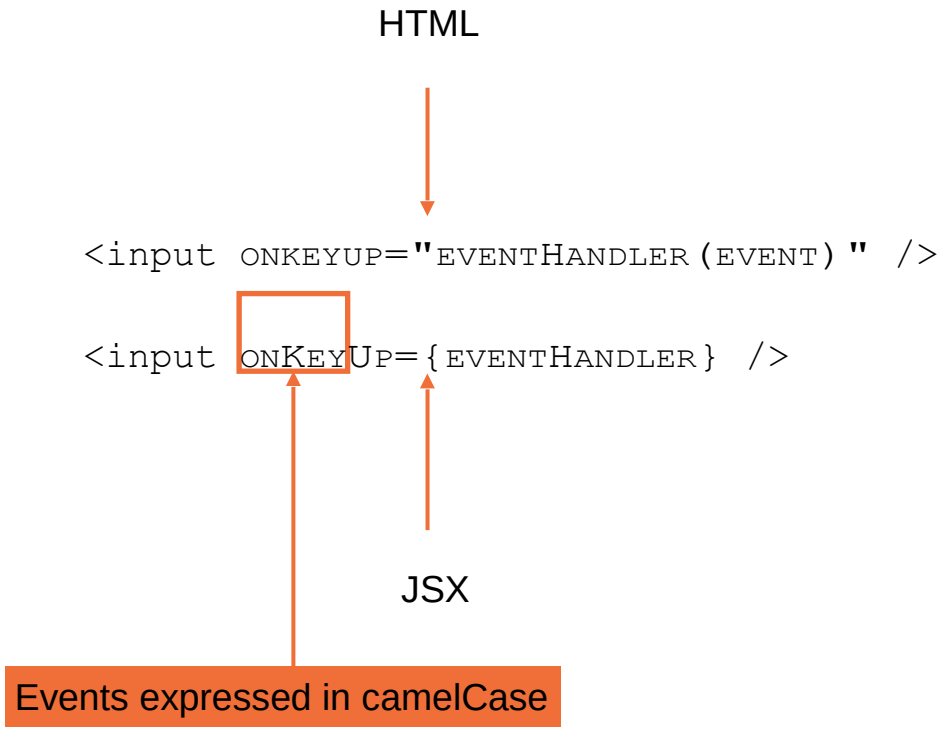


REACT EVENTS

Events should not be handled in HTML because it is difficult to manage and debug in practice.

REACT EVENTS

JSX in React uses a similar syntax as HTML for adding event listeners.

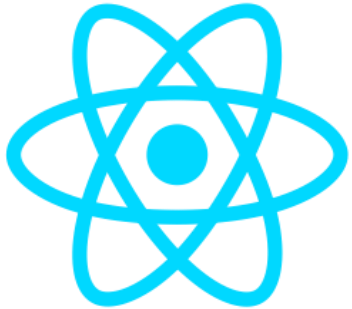


REACT EVENTS



Browsers may have differences when it comes to their native event handling mechanisms.

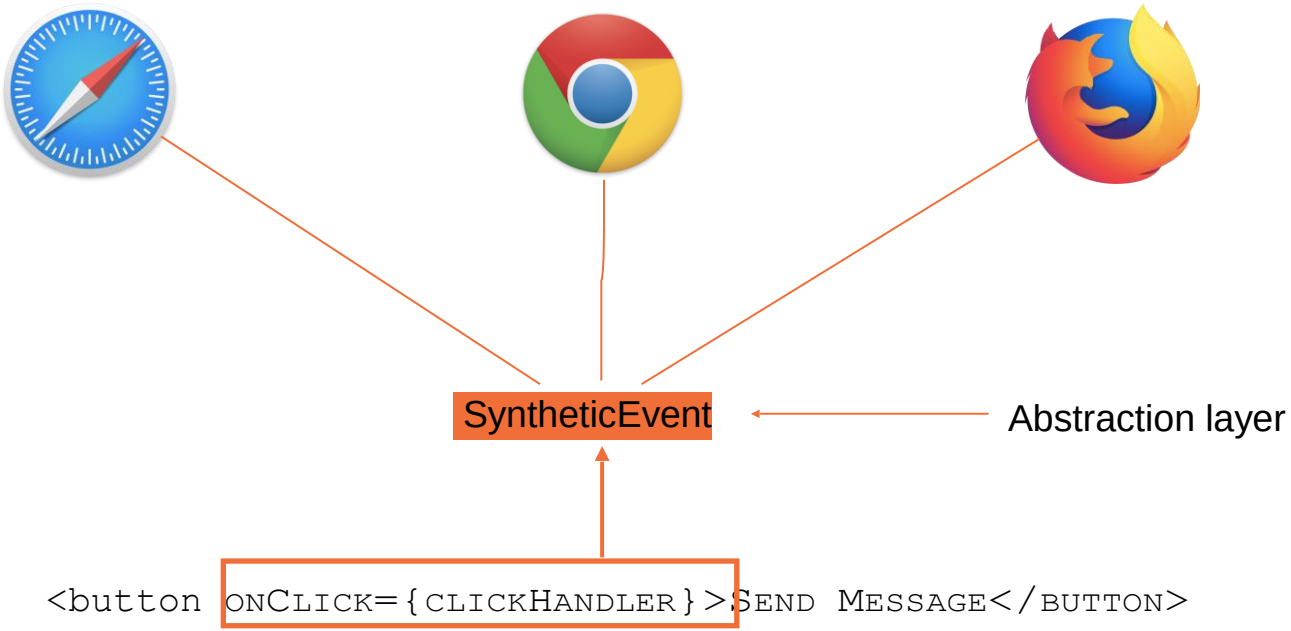
REACT EVENTS



SyntheticEvent

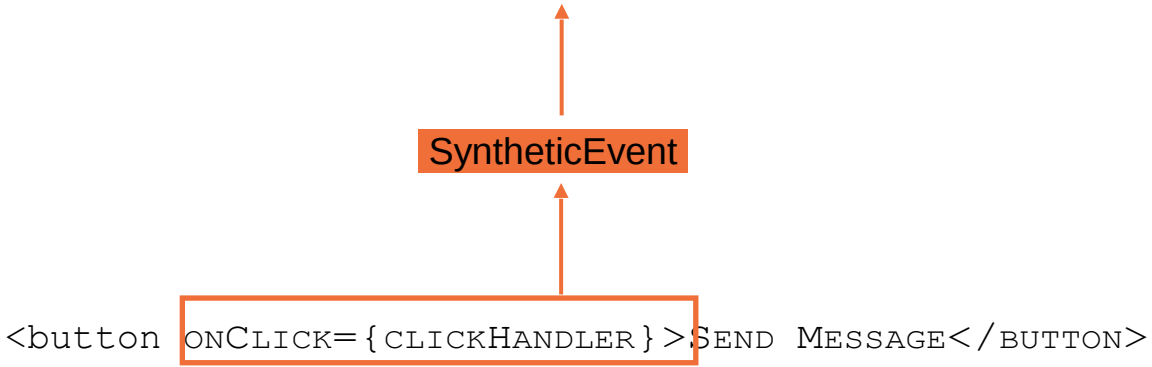
An abstraction layer on the browser’s native event system

REACT EVENTS



REACT EVENTS

```
clickHandler(event) {
    event.preventDefault();
    event.stopPropagation();
}
```

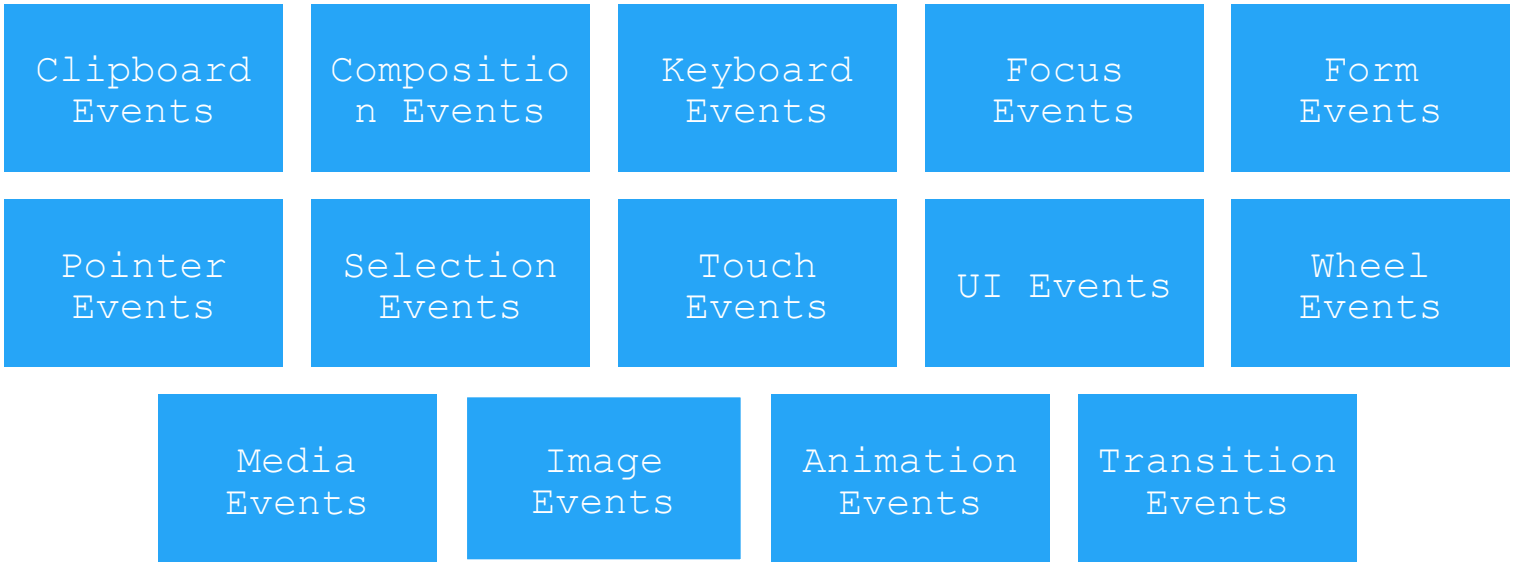


REACT EVENTS

- The developer gets to use a common and consistent API for managing events irrespective of the underlying browser.
- React manages browser level implementation
- Your app works brilliantly across all browsers!

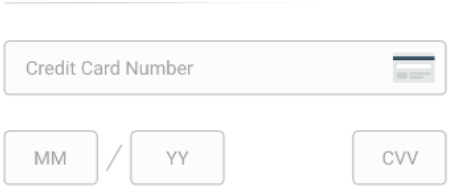
REACT EVENTS

The SyntheticEvent Wrappers | Categories

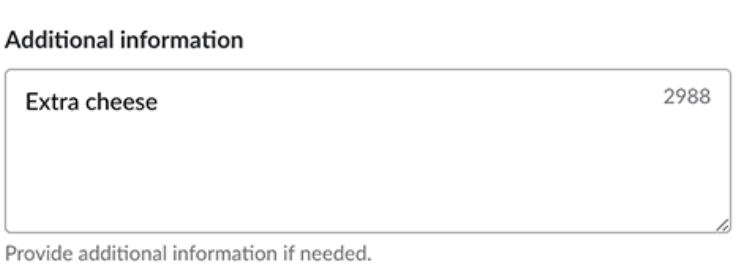


REACT EVENTS

Dropdowns



Input



Text area

```
<input onChange={onChangeHandler} />
```

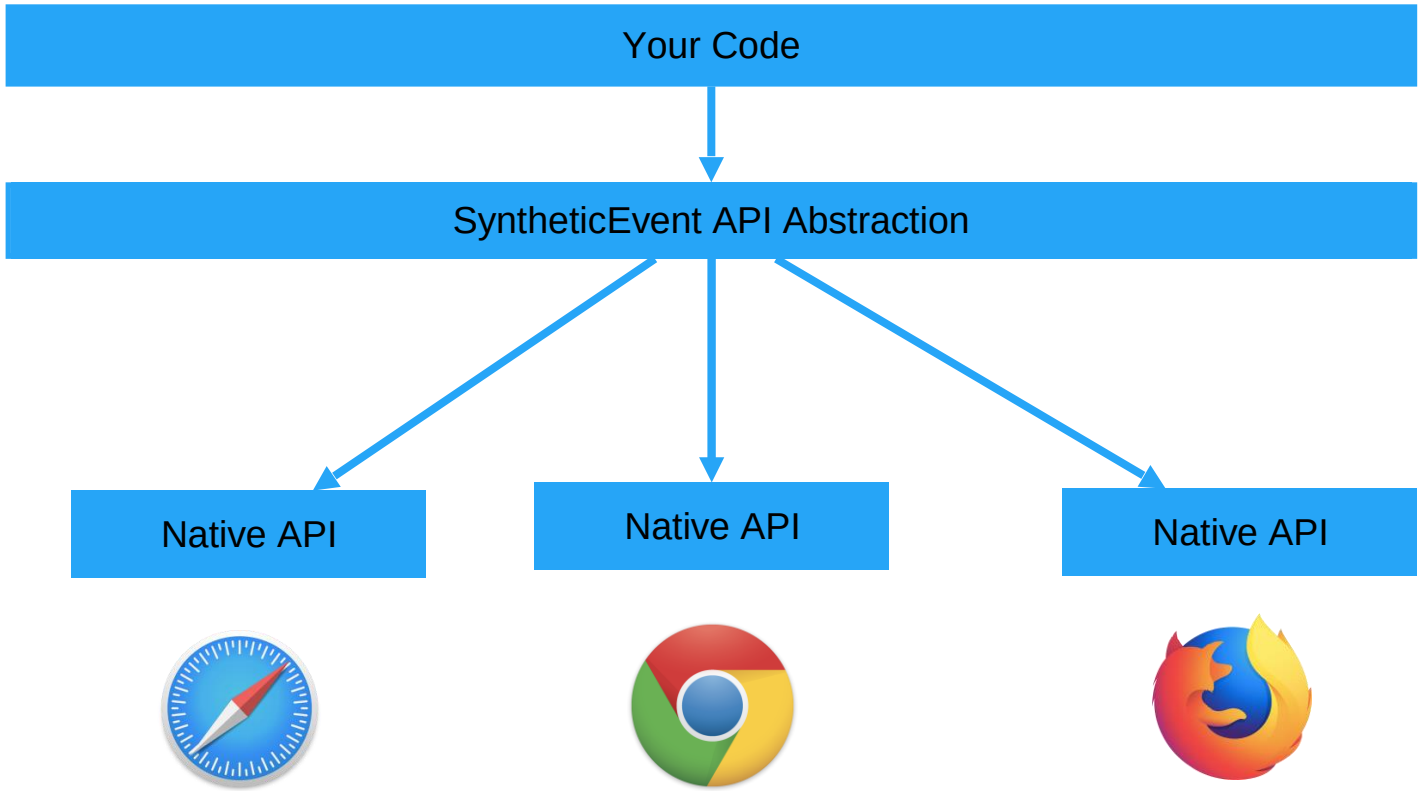
REACT EVENTS

```
<button onClick={onClickHandler}>Send Message</button>
```

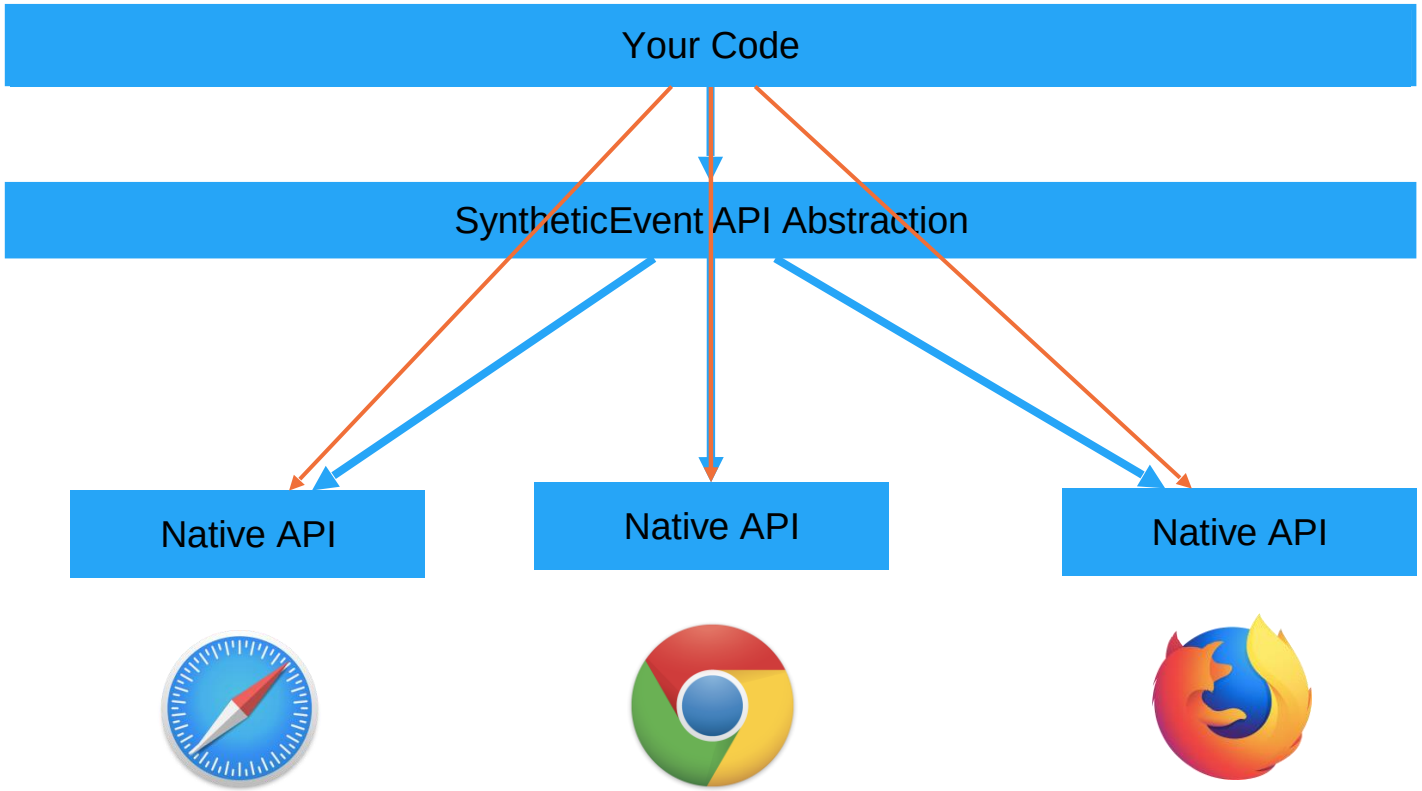
```
<img src={imgSource} onDoubleClick={openEditor} alt="Double-
click to edit" />
```

```
<div className="print-icon" onDrag={onDragHandler} />
```

REACT EVENTS



REACT EVENTS



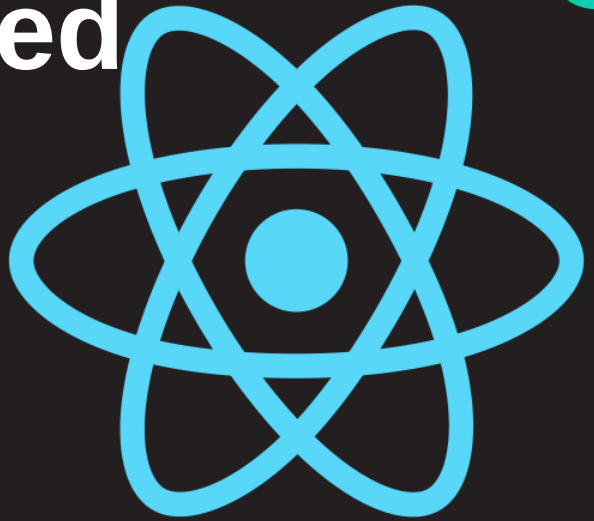
Hands-On

REACT EVENTS

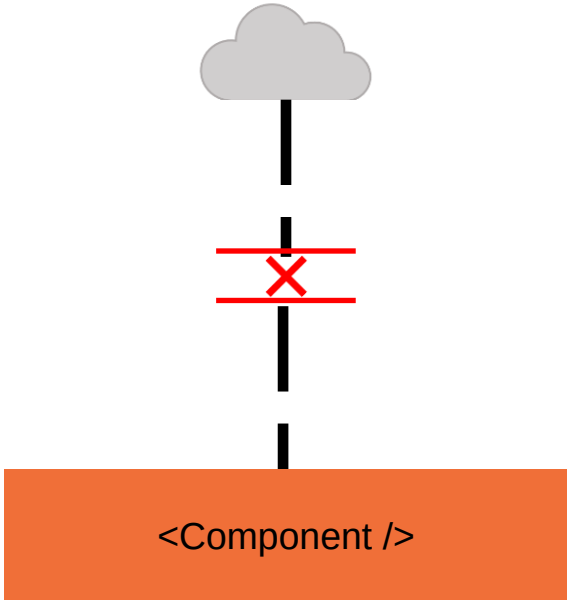
- SyntheticEvent is part of an event system that normalizes browser specific event APIs and enables writing apps that work seamlessly across all browsers.
- For special cases, where browser’s native event system access is needed, React provides easy and simple access.

Components Revisited

Error Management



ERROR MANAGEMENT



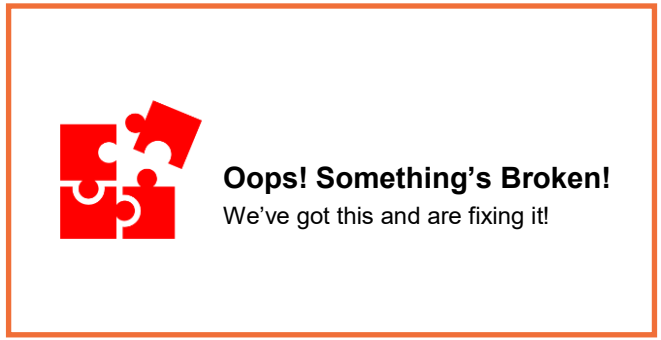
Errors when an API request fails to yield data

ERROR MANAGEMENT

```
class DataTable extends Component {
  state = {
    data: []
  }
  componentDidMount() {
    fetchDataFromApi()
      .then(data => this.setState({data}))
      .catch(error => logErrors(error));
  }
  render() {}
}
```

ERROR MANAGEMENT

Stable App



Fallback UI when an error occurs

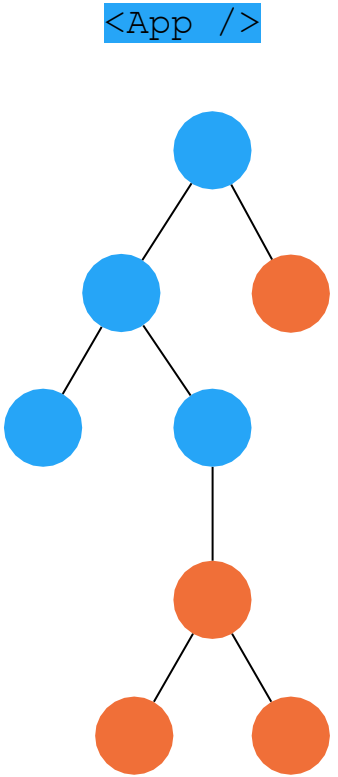
How do we handle runtime errors?

ERROR MANAGEMENT

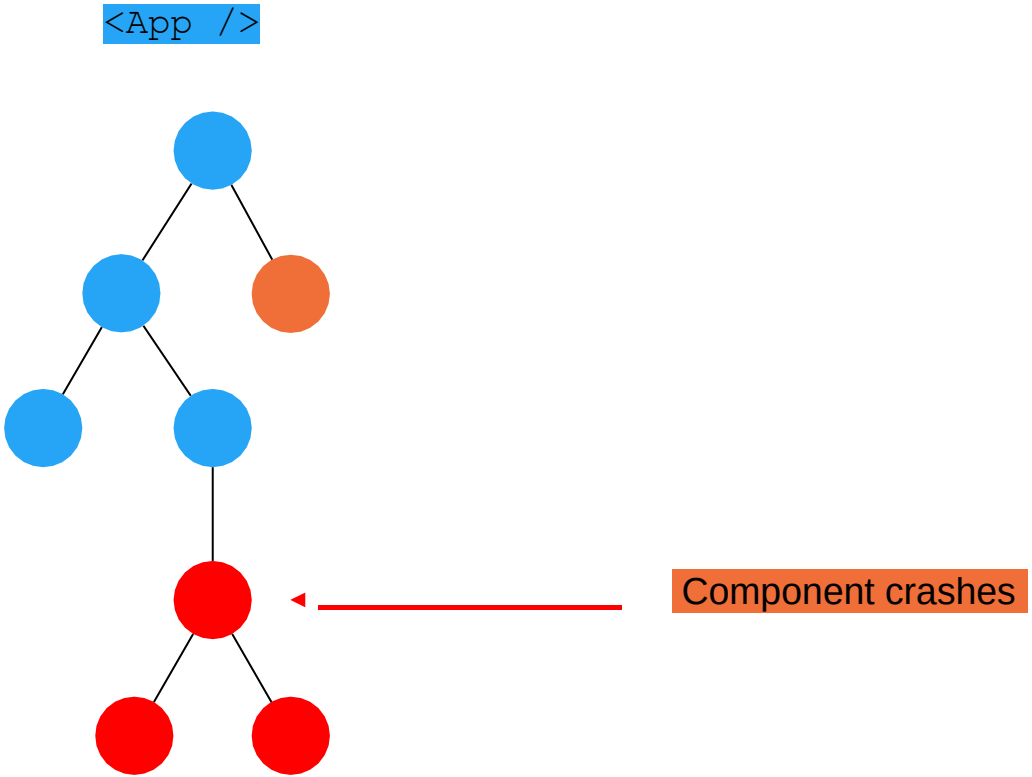
- If a component produces a run time error, it can produce unpredictable results, especially when local state is involved
- Components crashes, in fact, can crash the entire application in the browser
- These kinds of errors are unpredictable and needs to be handled.

ERROR MANAGEMENT

- This problem becomes even more complicated when you have a lot of nested components
- For instance, a hypothetical tree of components such as these poses a challenge when it comes to exception handling.



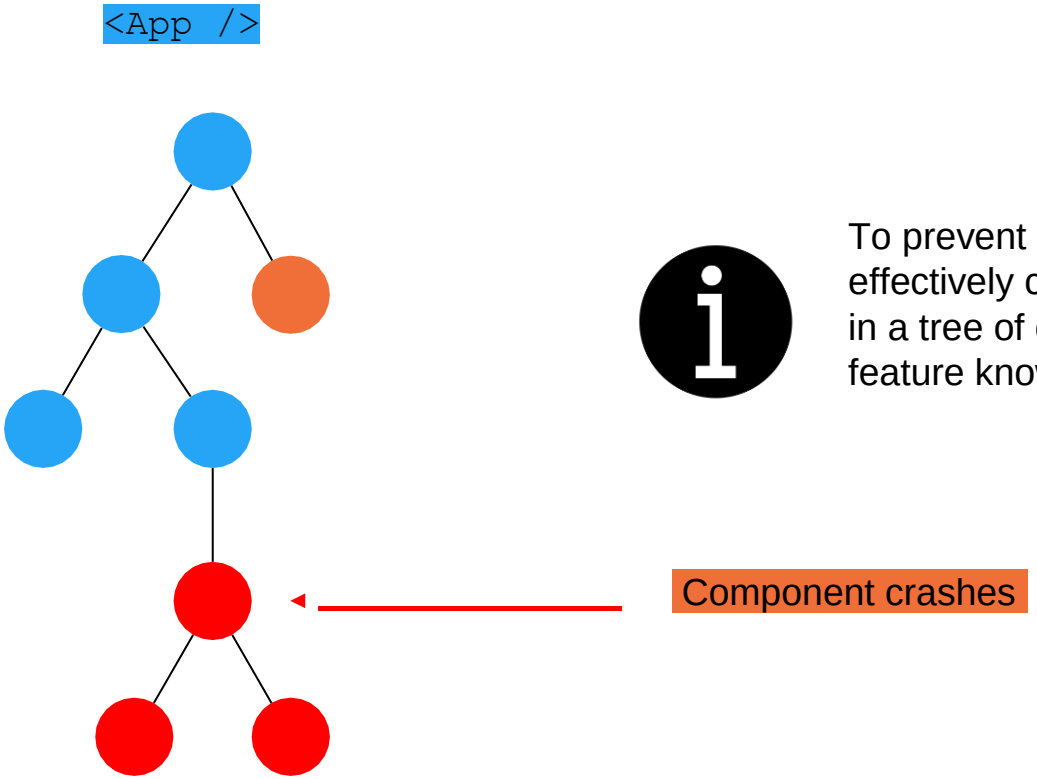
ERROR MANAGEMENT



Component crashes

ERROR MANAGEMENT

Application Crash



To prevent such a situation and to effectively capture and manage errors in a tree of components, React offers a feature known as error boundaries.

Component crashes

Hands-On

ERROR MANAGEMENT

A Class Component becomes an Error Boundary if it implements the getDerivedStateFromError() and/or componentDidCatch() methods.

```
class ErrorManager extends Component {
  state = {
    error: false
  };
  static getDerivedStateFromError() {
    return {
      error: true
    };
  }
  componentDidCatch(error, info) {
    logger(error, info);
  }
  render() {
    if (this.state.error) {
      return (
        <div className="error">
          <img src="./failWhale.jpg" alt="There was an error!" />
          <div
            className="reset-btn"
            onClick={() => {
              this.setState({
                error: false
              });
              this.props.onClose();
            }}
          >
            Close
          </div>
        </div>
      );
    }
    return this.props.children;
```

ERROR MANAGEMENT

- Errors in event handlers
- Errors in async code such as when handling side-effects or callbacks
- Server rendered React code (SSR)

ERROR MANAGEMENT

```
class FetchData extends Component
  { state = {
    data: [],
    error: false,
    errorData:
    ""
  };
  componentDidMount = async () => {
    try {
  const fetchData = await getDataFromApi();
      this.setState({
        data: fetchData
      });
    } catch ({errorInfo}) {
      this.setState({
        error: true,
        errorData: errorInfo
      });
    }
  };
  render() {    return error ? <FallBack /
> : <div className="fetch-data">...</div>
  }
```

Error management is critically important when building React apps and Error boundaries go a long way in streamlining exception management.

ERROR MANAGEMENT

```
}
```

thank you!