REACT.JS

LUXOFT

LXFT
LISTED
NYSE

BUILT-IN HTML TAGS

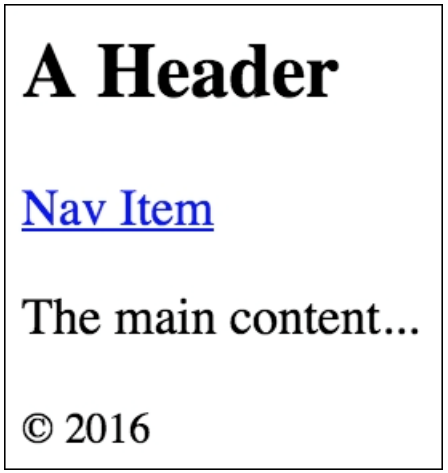
```
// React internal defines all the standard HTML tags
// that we use on a daily basis. Think of them being
// the same as any other react component.
render((
      <div>
         <button />
         <code />
         <input />
         <label />
         <p />
         <pre />
         <select />
         <table />
         <ul />
      </div>
   ),
   document.getElementById('app')
);
```

HTML TAG CONVENTIONS

```
// This renders as expected, except for the "foo"
// property, since this is not a recognized button
// property. This will log a warning in the console.
render((
      <button foo="bar">
         My Button
      </button>
   ),
   document.getElementById('app')
);

// This fails with a "ReferenceError", because
// tag names are case-sensitive. This goes against
// the convention of using lower-case for HTML tag names.
render(
   <Button />,
   document.getElementById('app')
);
```

DESCRIBING UI STRUCTURES

```
render((
    <section>
      <header>
        <h1>A Header</h1>
      </header>
      <nav>
        <a href="item">
            Nav Item</a>
      </nav>
      <main>
        <p>The main content...</p>
      </main>
      <footer>
        <small>&copy; 2016</small>
      </footer>
    </section>
  ),
  document.getElementById('app')
);
```



This JSX markup describes some fairly-sophisticated markup. Yet, it**'s easy to read,** because it's XML and XML is good **for** concisely-expressing hierarchical structure. This is how we want to think of our UI, when it needs to **change**, not as an individual element or property.

ENCAPSULATING HTML

Instead of having to type out complex markup, we just use our custom tag:

```
class MyComponent extends Component {
    render() {
        // All components have a "render()" method, which
        // returns some JSX markup. In this case, "MyComponent"
        // encapsulates a larger HTML structure.
        return (
            <section>
                <h1>My Component</h1>
                <p>Content in my component...</p>
            </section>
        );
    }
}
```

NESTED ELEMENTS

```
export default class MySection
extends Component {
   render() {
      return (
         <section>
            <h2>My Section</h2>
            {this.props.children}
         </section>
      );
   }
}
render((
      <MySection>
         <MyButton>My Button Text</MyButton>
      </MySection>
   ),
   document.getElementById('app')
);
```

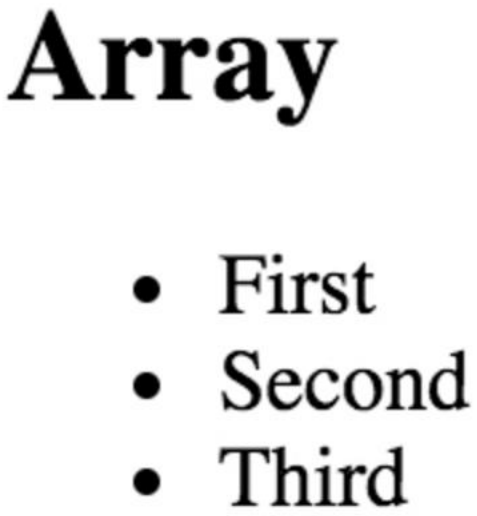
```
export default class MyButton
      extends Component {
   render() {
      return (
         <button>
            {this.props.children}
         </button>
      );
   }
}
```

DYNAMIC PROPERTY VALUES AND TEXT


```
const enabled = false;
const text = 'A Button';
const placeholder = 'input value...';
const size = 50;
render((
      <section>
         <button disabled={!enabled}>{text}</button>
         <input placeholder={placeholder} size={size} />
      </section>
   ),
   document.getElementById('app')
);
```

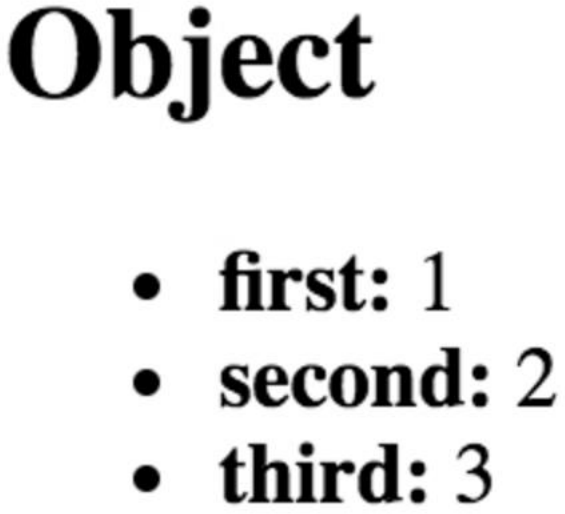
MAPPING COLLECTIONS TO ELEMENTS

```
// An array that we want to render as a list...
const array = [ 'First', 'Second', 'Third' ];
render((
  <section>
    <h1>Array</h1>

    { /* Maps "array" to an array of "<li>"s.
     Note the "key" property on "<li>".
     This is necessary for performance reasons,
     and React will warn us if it's missing. */ }
    <ul>
      {array.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>
  </section>
), document.getElementById('app') );
```

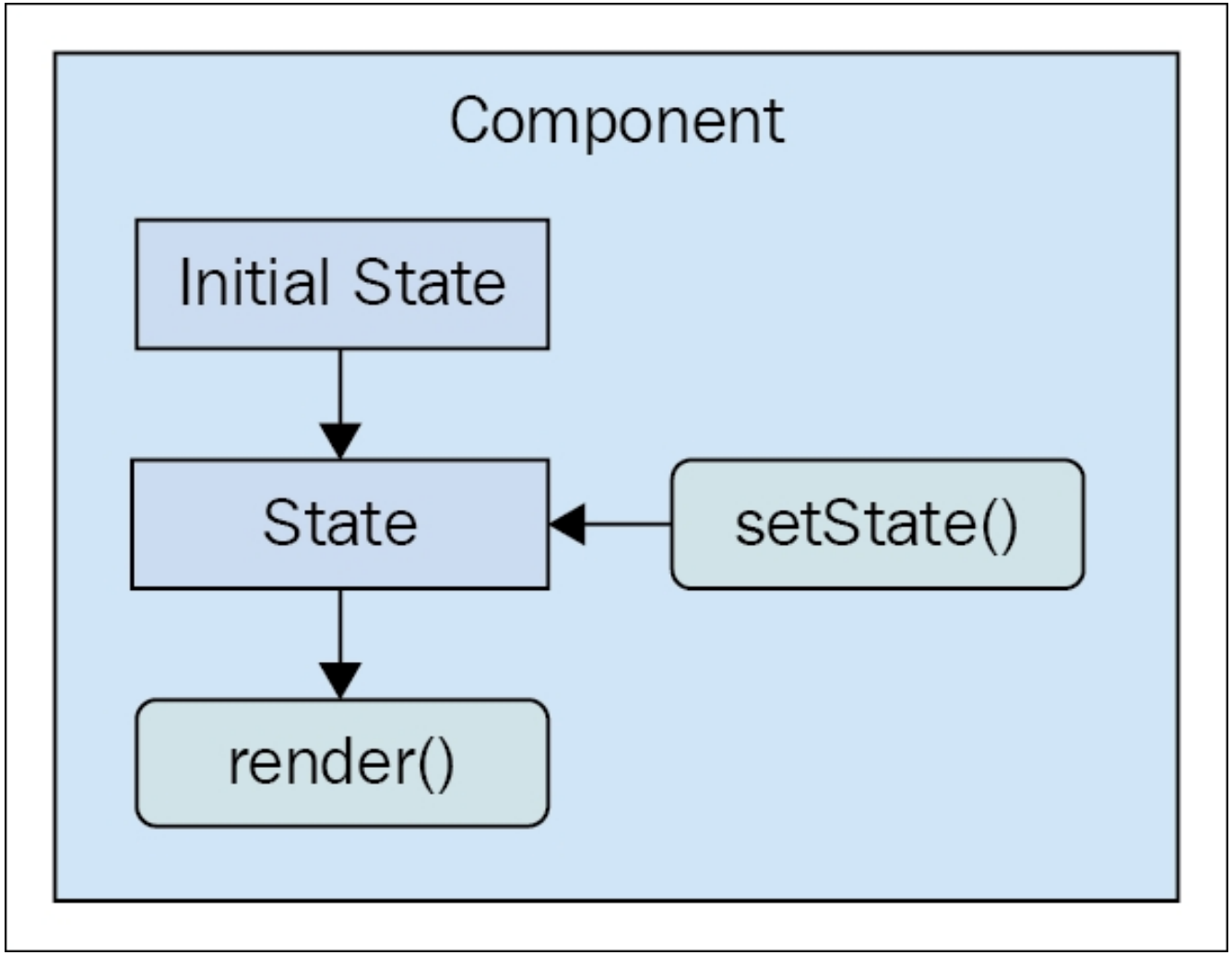


MAPPING OBJECT TO ELEMENTS

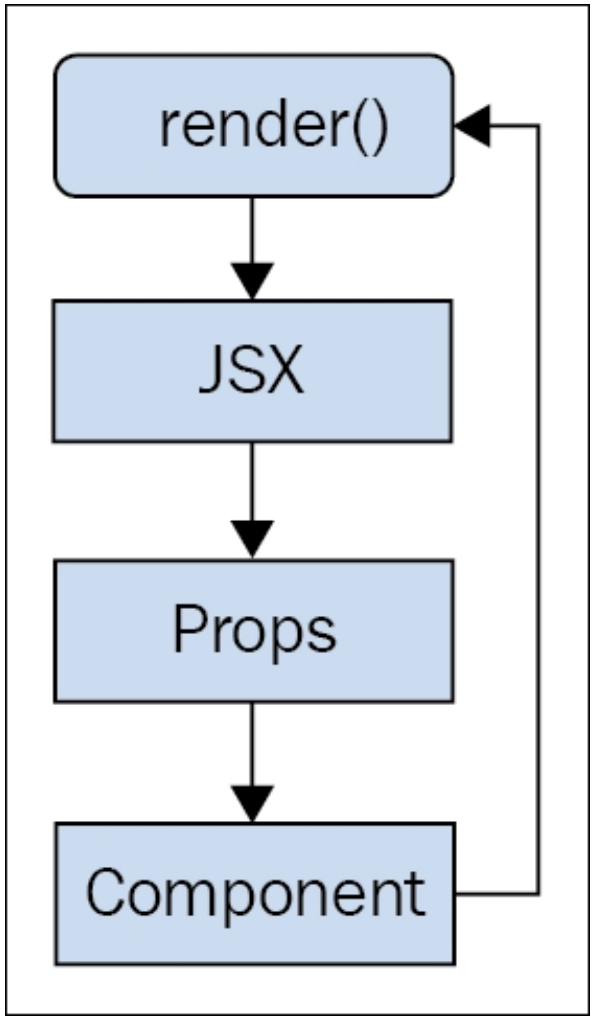
```
// An object that we want to render as a list...
const object = {  first: 1,  second: 2,  third: 3 };
render((
  <section><h1>Object</h1>
    { /* Maps "object" to an array of "<li>"s.
     Note that we have to use "Object.keys()"
     before calling "map()" and that we have
     to lookup the value using the key "i". */ }
    <ul>
      {Object.keys(object).map(i => (
        <li key={i}>
          <strong>{i}: </strong>{object[i]}
        </li>
      ))}
    </ul>
  </section>
  ), document.getElementById('app')
);
```



COMPONENT STATE



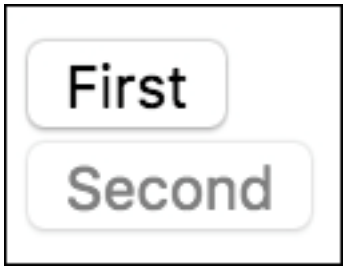
COMPONENT PROPERTIES



INITIAL COMPONENT STATE

```
export default class MyComponent extends Component {
   // The initial state is set
   // as a simple property
    state = {   first: false, second: true   }
    render() {
       const { first, second } = this.state;
       return (
          <main>
             <section>
                <button disabled={first}>First</button>
             </section>
             <section>
                <button disabled={second}>Second</button>
             </section>
          </main>
       );
    }
}
```

```
// "MyComponent" has an initial state
// Nothing is passed
// as a property when it's rendered.
render(
   (<MyComponent />),
   document.getElementById('app')
);
```



SETTING COMPONENT STATE

```
export default class MyComponent extends Component {
   // The initial state is used, until something
   // calls "setState()", at which point the state is
   // merged with this state.

   state = {
      heading: 'React Awesomesauce (Busy)',
      content: 'Loading...',
   }

   render() {
      const { heading, content } = this.state;

      return (
         <main>
            <h1>{heading}</h1>
            <p>{content}</p>
         </main>
      );
   }
}
```

CHANGING STATE

```
// The "render()" function returns a reference to the
// rendered component. In this case, it's an instance
// of "MyComponent". Now that we have the reference,
// we can call "setState()" on it whenever we want.
const myComponent = render(
   (<MyComponent />),
   document.getElementById('app')
);

// After 3 seconds, set the state of "myComponent",
// which causes it to re-render itself.
setTimeout(() => {
   myComponent.setState({
      heading: 'React Awesomesauce',
      content: 'Done!',
   });
}, 3000);
```

MERGING COMPONENT STATE

```
export default class MyComponent extends Component {
  // The initial state...
  state = { first: 'loading...', second: 'loading...',
    third: 'loading...' };
  render() {
    const { state } = this;
    // Renders a list of items from the  component state.
    return (
      <ul>
        {Object.keys(state).map(i => (
          <li key={i}>
            <strong>{i}: </strong>{state[i]}
          </li>
        ))}
      </ul>
    );
  }
}
```

MERGING COMPONENT STATE

```
// Stores a reference to the rendered component...
const myComponent = render(
    (<MyComponent />),
    document.getElementById('app')
);

// Change part of the state after 1 second...
setTimeout(() => {
    myComponent.setState({ first: 'done!' });
}, 1000);

// Change another part of the state after 2 seconds...
setTimeout(() => {
    myComponent.setState({ second: 'done!' });
}, 2000);

// Change another part of the state after 3 seconds...
setTimeout(() => {
    myComponent.setState({ third: 'done!' });
}, 3000);
```

- **first:** loading...
- **second:** loading...
- **third:** loading...

- **first:** done!
- **second:** done!
- **third:** loading...

DEFAULT PROPERTY VALUES

```
export default class MyButton extends Component {
    // The "defaultProps" values are used when the
    // same property isn't passed to the JSX element.
    static defaultProps = {
        disabled: false,
        text: 'My Button',
    }

    render() {
        // Get the property values we want to render.
        // In this case, it's the "defaultProps", since
        // nothing is passed in the JSX.
        const { disabled, text } = this.props;

        return (
            <button disabled={disabled}>{text}</button>
        );
    }
}
```

SETTING PROPERTY VALUES

```
export default class MyButton extends Component {

    // Renders a "<button>" element using values
    // from "this.props".
    render() {
        const { disabled, text } = this.props;

        return (
            <button disabled={disabled}>{text}</button>
        );
    }
}
```

SETTING PROPERTY VALUES

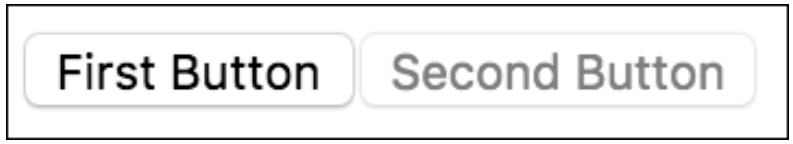
```
export default class MyList extends Component {
  render() {

    // The "items" property is an array.
    const { items } = this.props;

    // Maps each item in the array to a list item.
    return (
      <ul>
        {items.map(i => (
          <li key={i}>{i}</li>
        ))}
      </ul>
    );
  }
}
```

CHANGING PROPERTY VALUES

```
// This data changes over time, and we can pass the application data to
// components as properties.
const appState = { text: 'My Button', disabled: true,
  items: [ 'First', 'Second', 'Third' ] };
// Defines our own "render()" function. The "renderJSX()"
// function is from "react-dom" and does the actual rendering.
// We can call render() whenever there's new  application data.
function renderApp(props) {
    render ((
          <main>
            <MyButton text={props.text}
                  disabled={props.disabled} />
            <MyList items={props.items} />
          </main>
        ), document.getElementById('app')
    );
}
renderApp(appState); // Performs the initial rendering…
```

```
// After 1 second, changes
// some application data
setTimeout(() => {
    appState.disabled = false;
    appState.items.push('Fourth');
    renderApp(appState);
}, 1000);
```

PURE FUNCTIONAL COMPONENT




```
import MyButton from './MyButton';
// Renders two "MyButton" components.
function renderApp({ first, second }) {
   render((
         <main>
            <MyButton text={first.text}
                  disabled={first.disabled} />
            <MyButton text={second.text}
                  disabled={second.disabled}/>
         </main>
      ),  document.getElementById('app'));
}
// Reders the components, passing in property data.
renderApp({first: { text: 'First Button', disabled: false, },
   second: {  text: 'Second Button', disabled: true } });
```

Pure function is a function without side effects:

MyButton.js:

```
//  This function is pure because it has
// no state, and will always produce
//  the same output, given same input.
export default ({ disabled, text }) => (
   <button disabled={disabled}>
      {text}
  </button>
);
```

DEFAULTS IN FUNCTIONAL COMPONENTS

```
// The functional component doesn't care if the property
// values are the defaults, or if they're passed in from
// JSX. The result is the same.
const MyButton = ({ disabled, text }) => (
   <button disabled={disabled}>{text}</button>
);

// The "MyButton" constant was created so that we could
// attach the "defaultProps" metadata here, before
// exporting it.
MyButton.defaultProps = {
   text: 'My Button',
   disabled: false,
};
export default MyButton;
```

*When React encounters a functional component with **defaultProps**, it knows to pass in the defaults if they're not provided via JSX.*

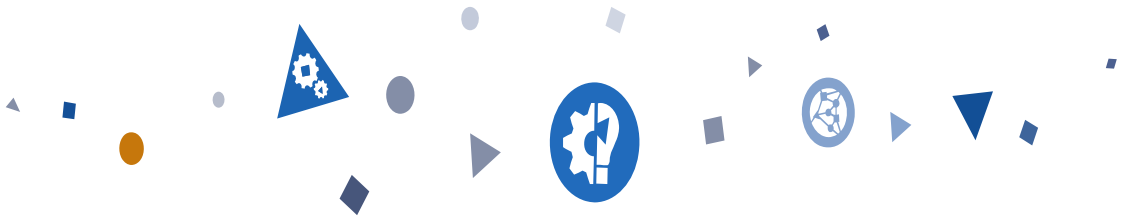
CONTAINER COMPONENTS

Basic intention: not to couple data fetching with the component that renders the data. The **container** is responsible for **fetching the data** and passing it to its child component. It **contains** the component responsible for **rendering the data**.

```
// Container components usually have state, so they
// can't be declared as functions.
export default class MyContainer extends Component {
  // Initial state will be passed down to child components
  state = { items: [] }
  componentDidMount() {
    fetchData().then(items => this.setState({ items }));
  }
  // Renders the containee, passing the container
  // state as properties, using the spread operator: "...".
  render() {
    return (
      <MyList {...this.state} />
    );
  }
}
```

- First
- Second
- Third

```
// A stateless component
export default ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    ))}
  </ul>
);

function fetchData() {
  return new Promise(
    (resolve) => {
      setTimeout(() => {resolve(
        ['First','Second','Third']);
      }, 2000);
  });
}
```



EVENTS HANDLING

DECLARING EVENT HANDLERS

```
export default class MyButton extends Component {
  // The click event handler, there's nothing much
  // happening here other than a log of the event.
  onClick() {
    console.log('clicked');
  }

  // Renders a "<button>" element with the "onClick"
  // event handler set to the "onClick()" method of
  // this component.
  render() {
    return (
      <button onClick={this.onClick}>
        {this.props.children}
      </button>
    );
  }
}
```

MULTIPLE EVENT HANDLERS

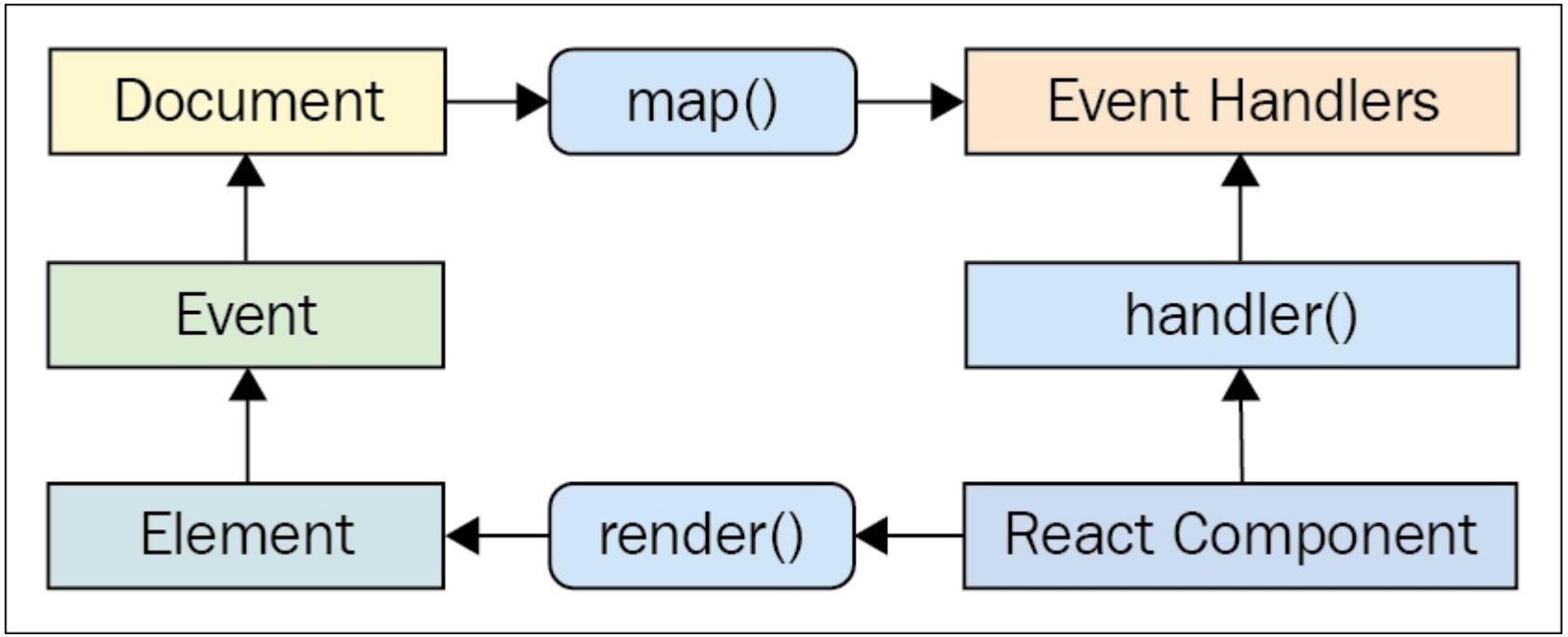
```
export default class MyInput extends Component {
   // Triggered when the value of the text input changes...
   onChange() { console.log('changed'); }

   // Triggered when the text input loses focus...
   onBlur() { console.log('blured'); }

   // JSX elements can have as many event handler
   // properties as necessary.
   render() {
      return (
         <input onChange={this.onChange} onBlur={this.onBlur} />
      );
   }
}
```

INLINE EVENT HANDLERS

```
export default class MyButton extends Component {

  // Renders a button element with an "onClick()" handler.
  // This function is declared inline with the JSX, and is
  // useful in scenarios where you need to call another
  // function.
  render() {
    return (
      <button
        onClick={e => console.log('clicked', e)}
      >
        {this.props.children}
      </button>
    );
  }
}
```

BINDING HANDLERS TO ELEMENTS

When you assign an event handler function to an element in JSX, React doesn't actually attach an event listener to the underlying DOM element. Instead, it ***adds the function to an internal mapping of functions***. There's a single event listener on the document for the page. As events bubble up through the DOM tree to the document, the React handler checks to see if any components have matching handlers. The process is illustrated here:



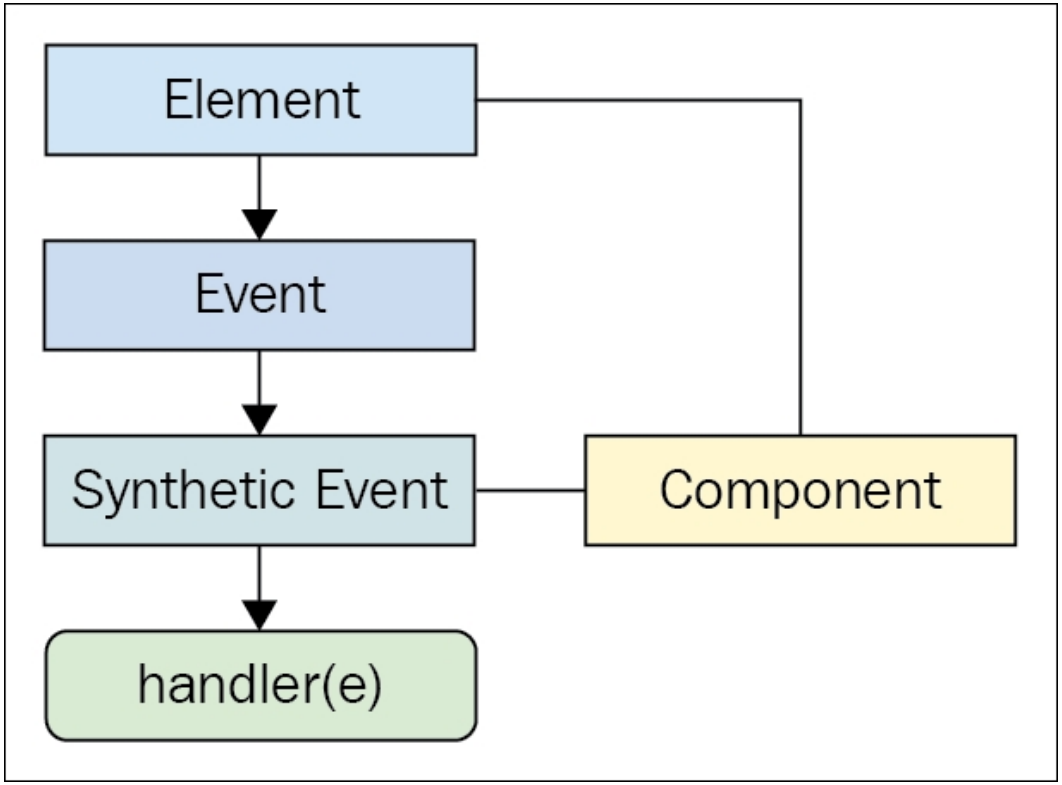
SYNTHETIC EVENT

Your event handlers will be passed instances of **SyntheticEvent**, a **cross-browser wrapper** around the browser's native event.

It has the same interface as the browser's native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers
If you find that you need the underlying browser event for some reason, simply use the nativeEvent attribute to get it.

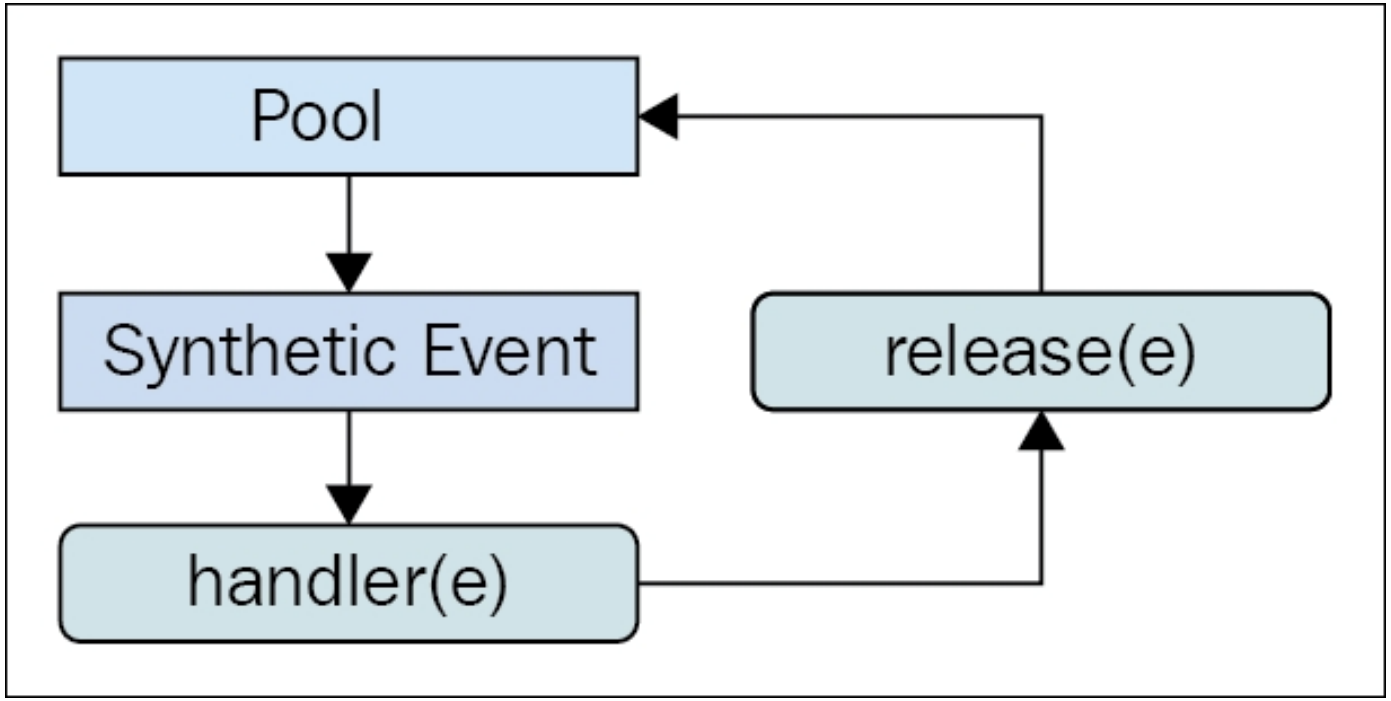
SYNTHETIC EVENT OBJECTS

When you attach an event handler function to a DOM element using the native addEventListener() function, the callback will get an event argument passed to it. Event handler functions in React are also passed an event argument, but it's not the standard Event instance. It's called **SyntheticEvent**, and it's a simple wrapper for native event instances.



EVENT POOLING

Modern applications respond to many events, even if the handlers don't actually do anything with them. This is problematic if React constantly has to allocate new synthetic event instances. React deals with this problem by allocating a **synthetic instance pool**. Whenever an event is triggered, it takes an instance from the pool and populates its properties. When the event handler has finished running, the synthetic event instance is released back into the pool, as shown here:

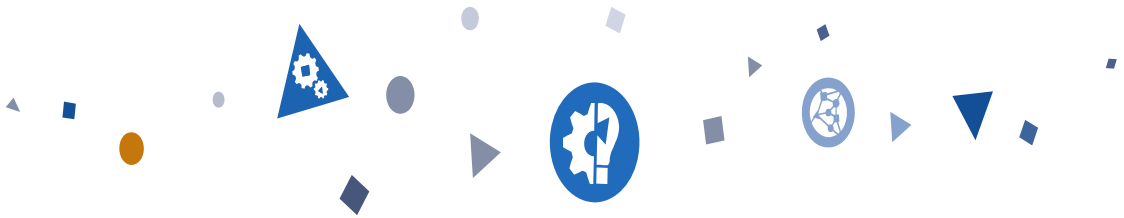


PROBLEM WITH POOLING

```
export default class MyButton extends Component {
   onClick(e) {
      console.log('clicked', e.currentTarget.style); // This works fine
      fetchData().then(() => {
         // However, trying to access "currentTarget"
         // asynchronously fails, because it's properties
         // have all been nullified so that the instance
         // can be reused.
         console.log('callback', e.currentTarget.style);
      });
   }
   render() {
      return (
         <button onClick={this.onClick}> {this.props.children} </button>
      );}
}
```

let style = e.currentTarget.style;

```
function fetchData() {
   return new Promise(
   (resolve) => {
      setTimeout(() => {
         resolve();
      }, 1000);
   });
}
```



COMPONENT VALIDATION

BASIC TYPE VALIDATION

```
const MyComponent = ({myString,myNumber,myBool,
                      myFunc,myArray,myObject}) => (
  <section>
    <p>{myString}</p>
    <p>{myNumber}</p>
    { /* Booleans are used as property values. */ }
    <p><input type="checkbox"
          defaultChecked={myBool} /></p>
    <p>{myFunc()}</p>
    <ul>
      {myArray.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>
    <p>{myObject.myProp}</p>
  </section>
);
```

```
// The "propTypes"
// specification
// for this component.
MyComponent.propTypes = {
 myString: PropTypes.string,
 myNumber:
      PropTypes.number,
 myBool: PropTypes.bool,
 myFunc: PropTypes.func,
 myArray: PropTypes.array,
 myObject: PropTypes.object,
};

export default
MyComponent;
```

REQUIRED AND ANY PROPERTIES

```
const MyComponent = ({myString,myNumber,myBool,
                      myFunc,myArray,myObject}) => (
  <section>
    <p>{myString}</p>
    <p>{myNumber}</p>
    <p><input type="checkbox”
          defaultChecked={myBool} />
    </p>
    <p>{myFunc()}</p>
    <ul>
      {myArray.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>
    <p>{myAny}</p>
  </section>
);
```

```
// The "propTypes”
// specification for this component.
// Every property is required

MyComponent.propTypes = {
 myString:
        PropTypes.string.isRequired,
 myNumber:
        PropTypes.number.isRequired,
 myBool: PropTypes.bool.isRequired,
 myFunc: PropTypes.func.isRequired,
 myArray: PropTypes.array.isRequired,
 myAny: PropTypes.any.isRequired,
};

export default MyComponent;
```

USING VALIDATION

```
// "myObject" property is missing. This will trigger a warning.
const missingProp = {
    myString: 'My String',
    myNumber: 100,
    myBool: true,
    myFunc: () => 'My Return Value',
    myArray: ['One', 'Two', 'Three'],
    myAny: 'Here we can put any object, but this is obligatory'
};
// Renders "<MyComponent>" with the given "props".
function renderApp(props) {
    render(
        (<MyComponent {...props} />),
        document.getElementById('app')
    );
}
renderApp(missingProp);
```

Warning:

Required prop `myObject` was not specified in `MyComponent`.
Cannot read property 'myProp' of undefined

REQUIRING SPECIFIC TYPES

```
const MyComponent = ({myDate, myCount, myUsers}) => (
  <section>
    { /* Requires a specific "Date" method. */ }
    <p>{myDate.toLocaleString()}</p>
    { /* Number or string works here. */ }
    <p>{myCount}</p>
    <ul>
      { /* "myUsers" is expected to be an array of
       "MyUser" instances. So we know that it's
       safe to use the "id" and "name" property. */ }
      {myUsers.map(i => (
        <li key={i.id}>{i.name}</li>
      ))}
    </ul>
  </section>
);
```

```
MyComponent.propTypes = {
 myDate: PropTypes.
      instanceOf(Date),
 myCount:
      PropTypes.oneOfType([
   PropTypes.string,
   PropTypes.number,
 ]),
 myUsers: PropTypes.arrayOf(
      PropTypes.
      instanceOf(MyUser)),
};
```

REQUIRING SPECIFIC VALUES

```
const MyComponent =
   ({level,user}) => (
   <section>
      <p>{level}</p>
      <p>{user.name}</p>
      <p>{user.age}</p>
   </section>
);

MyComponent.propTypes = {
   level: PropTypes.oneOf(levels),
   user: PropTypes.shape(userShape),
};
```

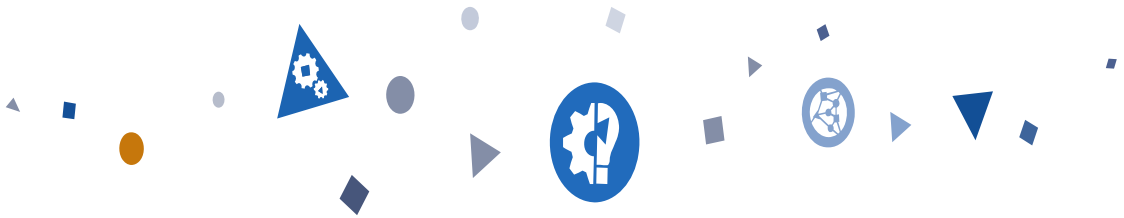
```
// Any one of these
//  is a valid "level"
// property value.
const levels = new Array(10)
   .fill(null)
   .map((v, i) => i + 1);

// This is the "shape"
// of the object we expect
// to find in the "user"
// property value.
const userShape = {
   name: PropTypes.string,
   age: PropTypes.number,
};
```

CUSTOM VALIDATOR

```
const MyComponent =
  ({myArray,myNumber}) => (
  <section>
    <ul>
      {myArray.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>
    <p>{myNumber}</p>
  </section>
);

{ /* Both custom validators fail... */ }
<MyComponent myArray={[]}
myNumber={100}/>
```

```
MyComponent.myArray: expecting
non-empty array
MyComponent.myNumber: expecting
number between 1 and 99
```

```
MyComponent.propTypes = {
  // Expects a property named
  //"myArray" with a non-zero
  // length. If this passes, we return null..
  myArray: (props, name, component) =>
    (Array.isArray(props[name]) &&
    props[name].length) ? null : new Error(
      `${component}.${name}:
        expecting non-empty array`
    ),
  // Expects property named "myNumber" that's
  // greater than 0 and less than 99.
  myNumber: (props, name, component) =>
    (Number.isFinite(props[name]) &&
    props[name] > 0 &&
    props[name] < 100) ? null : new Error(
      `${component}.${name}: ` +
      `expecting number between 1 and 99`
    ),
};
```



ROUTING

ROUTING

http://localhost:3000/shop#**/books**

BOOKS GAMES ALBUMS APPS

Books:

1. ReactJS
2. JavaScript
3. Redux

http://localhost:3000/shop**/books/1**

<head><base href="/shop"></head>

ROUTES

Routing enables you to show different content depending on what route is chosen. A route is specified in the URL.

http://localhost:3000/shop**/books**	books-> BookComponent
http://localhost:3000/shop**/albums**	albums->AlbumsComponent
http://localhost:3000/shop#**/games**	games->GamesComponent
http://localhost:3000/shop#**/apps**	apps->AppsComponent

route: loads mapped component in placehoder in index.html

Advantages:

- better app structure;
- forward/backward in browser;
- app state can be shared as URL (to search engine, social framework, favorites, messengers, etc…)

INSTALLATION AND USE OF ROUTER 4

npm install react-router-dom

```
import { BrowserRouter } from 'react-router-dom'
ReactDOM.render((
   <BrowserRouter>
      <App />
   </BrowserRouter>
), document.getElementById('root'))
```

```
<Route path='/roster'/>
vs.
<Route exact path='/roster'/>
```

```
const App = () => (
   <div>
      <Header />
      <Main />
   </div>
)
```

```
const Main = () => (
   <main>
      <Switch>
         <Route exact path='/' component={Home}/>
         <Route path='/roster' component={Roster}/>
         <Route path='/schedule' component={Schedule}/>
      </Switch>
   </main>
)
```

NESTED ROUTES

The player profile route /roster/:number is not included in the previous <Switch>. Instead, it will be rendered by the <Roster> component, which is rendered whenever the pathname begins with /roster:

```
const Roster = () => (
  <div>
    <h2>This is a roster page!</h2>
    <Switch>
      <Route exact path='/roster' component={FullRoster}/>
      <Route path='/roster/:number' component={Player}/>
    </Switch>
  </div>
)
```

/roster — This should only be matched when the pathname is exactly /roster, so we should also give that route element the exact prop.
/roster/:number — This route uses a path param to capture the part of the pathname that comes after /roster.

ROUTER PARAMS

The :number part of the path /roster/:number means that the part of the pathname that comes after /roster/ will be captured and stored as **match.params.number**. For example, the pathname **/roster/6** will generate a **params** object : **{ number: '6' }**

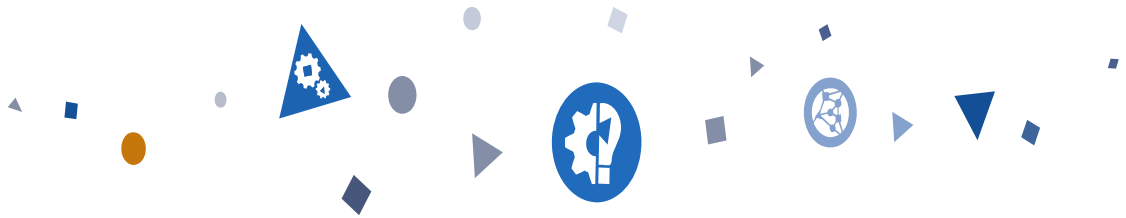
```
const Player = (props) => {
  const player = PlayerAPI.get(
    parseInt(props.match.params.number, 10)
  )
  if (!player) {
    return <div>Sorry, but the player was not found</div>
  }
  return (
    <div>
      <h1>{player.name} (#{player.number})</h1>
      <h2>{player.position}</h2>
    </div>
  )
```

LINKS

```
const Header = () => (
  <header>
    <nav>
      <ul>
        <li><Link to='/'>Home</Link></li>
        <li><Link to='/roster'>Roster</Link></li>
        <li><Link to='/schedule'>Schedule</Link></li>
      </ul>
    </nav>
  </header>
)
```

Parameter **to** can either be a string or a **location object** (containing a combination of pathname, search, hash, and state properties). When it is a string, it will be converted to a location object.

```
<Link to={{ pathname: '/roster/7' }}>Player #7</Link>
```



IMMUTABLE.JS

IMMUTABILITY

IMMUTABLE.JS

Much of what makes application development difficult is **tracking mutation** and **maintaining state**. Developing with immutable data encourages you to think differently about how data flows through your application.

```
<script src="immutable.min.js"></script>
<script>
   var map1 = Immutable.Map({a:1, b:2, c:3});
   var map2 = map1.set('b', 50);
   map1.get('b'); // 2
   map2.get('b'); // 50
   assert(map1.equals(map2) === true);
   const map3 = map1.set('b', 50);
   assert(map1.equals(map3) === false);
</script>
```

Immutable collections should be treated as **values** rather than **objects**. While **objects** represent some thing which could **change over time**, a **value** represents the **state** of that thing at a **particular instance of time**.

CREATING COPY

If an object is immutable, it can be "copied" simply by making another reference to it instead of copying the entire object. Because a reference is much smaller than the object itself, this results in memory savings and a potential boost in execution speed for programs which rely on copies (such as an undo-stack).

```
const map1 = Map({ a: 1, b: 2, c: 3 })
const clone = map1;
```

IMMUTABLE COLLECTIONS

Immutable collections **will not mutate** the collection with methods like **push**, **set**, **unshift** or **splice**. It will instead **return a new immutable** collection.

```
const { List } = require('immutable')
const list1 = List([ 1, 2 ]);
const list2 = list1.push(3, 4, 5);
const list3 = list2.unshift(0);
const list4 = list1.concat(list2, list3);
assert(list1.size === 2);
assert(list2.size === 5);
assert(list3.size === 6);
assert(list4.size === 13);
assert(list4.get(0) === 1);
```

Almost all of the methods on Array will be found on Immutable.List, those of Map found on Immutable.Map, and those of Set found on Immutable.Set:

```
const { Map } = require('immutable')
const alpha = Map({ a: 1, b: 2, c: 3, d: 4 });
alpha.map((v, k) => k.toUpperCase()).join();
```

WORKING WITH JAVASCRIPT OBJECTS

Designed to inter-operate with your existing JavaScript, Immutable.js accepts plain JavaScript Arrays and Objects anywhere a method expects an Collection:

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3, d: 4 })
const map2 = Map({ c: 10, a: 20, t: 30 })
const obj = { d: 100, o: 200, g: 300 }
const map3 = map1.merge(map2, obj);
// Map { a: 20, b: 2, c: 10, d: 100, t: 30, o: 200, g: 300 }
```

Keep in mind, when using JS objects to construct Maps, JS Object properties are always strings:

```
const { fromJS } = require('immutable’)
const obj = { 1: "one" }
Object.keys(obj) // [ "1" ]
obj["1"] // "one"
obj[1]   // "one”

const map = fromJS(obj)
map.get("1") // "one"
map.get(1)   // undefined
```

CONVERTING BACK TO JAVASCRIPT

All Immutable.js Collections can be converted to plain JavaScript Arrays and Objects shallowly with **toArray()** and **toObject()** or deeply with **toJS()**. All Immutable Collections also implement **toJSON()**.

```
const { Map, List } = require('immutable')

const deep = Map({ a: 1, b: 2, c: List([ 3, 4, 5 ]) })

deep.toObject() // { a: 1, b: 2, c: List [ 3, 4, 5 ] }
deep.toArray() // [ 1, 2, List [ 3, 4, 5 ] ]
deep.toJS() // { a: 1, b: 2, c: [ 3, 4, 5 ] }

JSON.stringify(deep) // '{"a":1,"b":2,"c":[3,4,5]}'
```

NESTED STRUCTURES

The collections in Immutable.js are intended to be nested, allowing for deep trees of data, similar to JSON.

```
const nested = fromJS({ a: { b: { c: [ 3, 4, 5 ] } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ] } } }

const nested2 = nested.mergeDeep({ a: { b: { d: 6 } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 6 } } }

nested2.getIn([ 'a', 'b', 'd' ]) // 6

const original = { x: { y: { z: 123 }}}
const res = setIn(original, ['x', 'y', 'z'], 456) // res == { x: { y: { z: 456 }}}

const nested3 = nested2.updateIn([ 'a', 'b', 'd' ], value => value + 1)
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 7 } } }

const nested4 = nested3.updateIn([ 'a', 'b', 'c' ], list => list.push(6))
// Map { a: Map { b: Map { c: List [ 3, 4, 5, 6 ], d: 7 } } }
```

LAZY SEQ

Seq describes a lazy operation, allowing them to efficiently chain use of all the sequence methods (such as map and filter).

Seq is **immutable** — Once a Seq is created, it cannot be changed, appended to, rearranged or otherwise modified. Instead, any mutative method called on a Seq will return a new Seq.

Seq is **lazy** — Seq does as little work as necessary to respond to any method call.

For example, the following does not perform any work, because the resulting Seq is never used:

```
const { Seq } = require('immutable')
const oddSquares = Seq([ 1, 2, 3, 4, 5, 6, 7, 8 ])
   .filter(x => x % 2)
   .map(x => x * x)
```

Once the Seq is used, it performs only the necessary work.
In this example, **no intermediate arrays** are ever **created**, filter is called three times, and map is only called once:

```
console.log(oddSquares.get(1)); // 9
```

LAZY SEQ

Any collection can be converted to a lazy Seq with .toSeq().

```
const { Map } = require('immutable')
const seq = Map({ a: 1, b: 2, c: 3 }).toSeq()
```

With Immutable it's possible to express logic that would otherwise seem memory-limited:

```
const { Range } = require('immutable')
Range(1, Infinity)
   .skip(1000)
   .map(n => -n)
   .filter(n => n % 2 === 0)
   .take(2)
   .reduce((r, n) => r * n, 1);
// 1006008
```

COMPARING FOR EQUALITY

Immutable.js provides equality which treats immutable data structures as pure data, performing a deep equality check if necessary.

```
const { Map, is } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = Map({ a: 1, b: 2, c: 3 })
assert(map1 !== map2) // two different instances
assert(is(map1, map2)) // have equivalent values
assert(map1.equals(map2)) // alternatively use the equals method
```

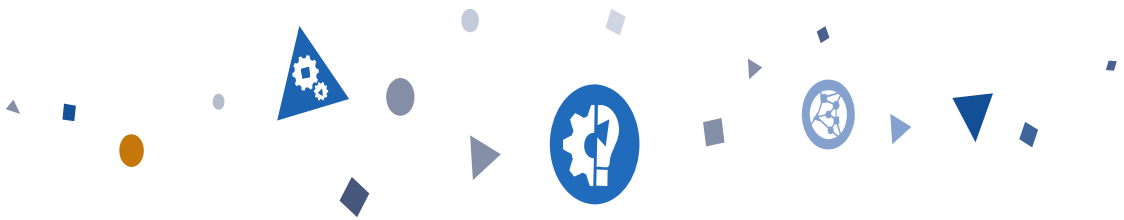
Immutable.is() uses the same measure of equality as Object.is including if both are immutable and all keys and values are equal using the same measure of equality.

BATCHING MUTATIONS

Apply several mutations to immutable => **overhead**
To make it perfomant, use withMutations():

```
const { List } = require('immutable')
const list1 = List([ 1, 2, 3 ]);
const list2 = list1.withMutations(function (list) {
   list.push(4).push(5).push(6);
});
assert(list1.size === 3);
assert(list2.size === 6);
```

NOTE: Only a select few methods can be used in withMutations including **set**, **push** and **pop**.
Other methods like **map**, **filter**, **sort**, and **splice** will always return **new immutable** data-structures and never mutate a mutable collection.



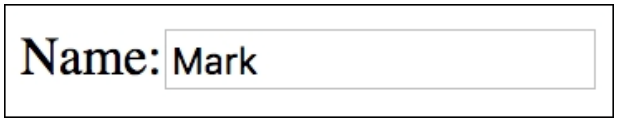
COMPONENT INHERITANCE

INHERITING STATE: BASE CLASS

```
export default class BaseComponent extends Component {
    state = {
        data: fromJS({ name: 'Mark', enabled: false, placeholder: '' }),
    }

    // Getter for "Immutable.js" state data...
    get data() { return this.state.data; }

    // Setter for "Immutable.js" state data...
    set data(data) { this.setState({ data });   }

    // The base component doesn't actually render anything,
    // but it still needs a render method.
    render() {
        return null;
    }
}
```

INHERITING STATE: CHILD CLASS

```
export default class MyComponent extends BaseComponent {
  // This is our chance to build on the initial state.
  // We change the "placeholder" text and mark it as
  // "enabled".
  componentWillMount() {
    this.data = this.data
      .merge({
        placeholder: 'Enter a name...',
        enabled: true,
      });
  }

  render() {
    const {enabled,name,placeholder} =
      this.data.toJS();
    return (
      <label htmlFor="my-input">
        Name:
        <input
          id="my-input"
          disabled={!enabled}
          defaultValue={name}
          placeholder={placeholder}
        />
      </label>
    );
  }
}
```

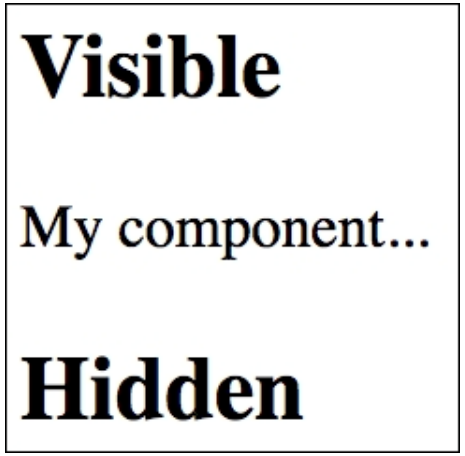


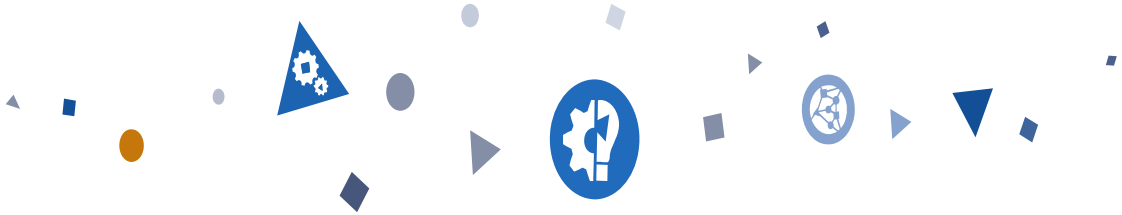
If you delete the default text in the input element, you can see that the placeholder text:

CONDITIONAL COMPONENT RENDERING

```
let MyComponent = () => <p>My component...</p>;

// A minimal higher-order function is all it
// takes to create a component repeater. Here, we're
// returning a function that calls "predicate()".
// If this returns true, then the rendered
// "<Component>" is returned.
let cond = (Component, predicate) =>
    props =>
    predicate() && (<Component {...props} />);

import MyComponent from './MyComponent';

const ComposedVisible = cond(MyComponent, () => true);
const ComposedHidden = cond(MyComponent, () => false);
```

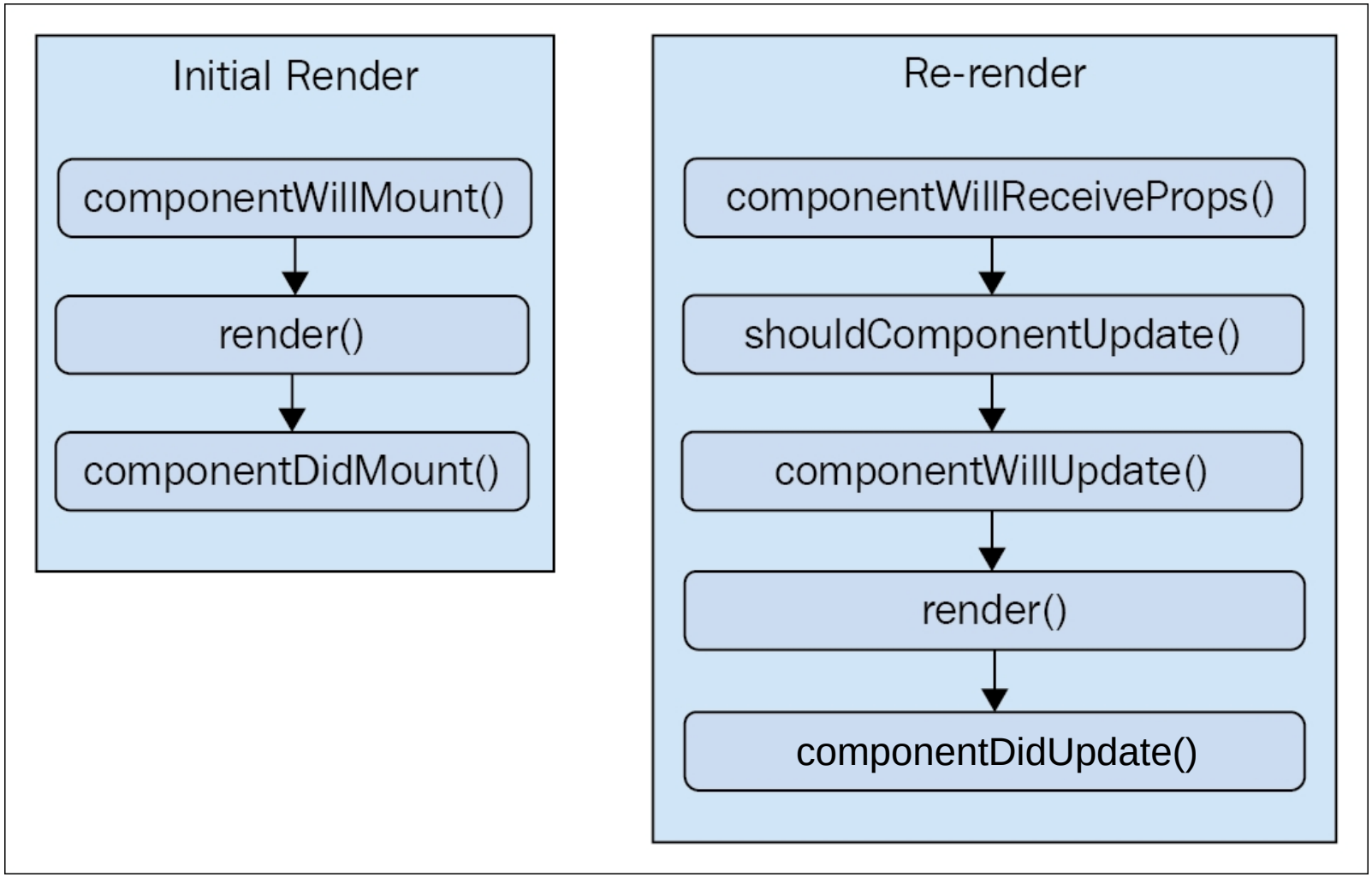
```
render((
    <section>
        <h1>Visible</h1>
        <ComposedVisible />
        <h2>Hidden</h2>
        <ComposedHidden />
    </section>
),
document.
    getElementById('app')
```





COMPONENTS LIFECYCLE

COMPONENT LIFECYCLE



INITIALIZING STATE WITH PROPERTIES

```
export default class UserListContainer extends Component {
  constructor() { // fromJS() deeply converts JS to Immutable Maps and Lists
      this.state = { data: fromJS({ error: null, loading: 'loading...', users: [] }) };
  }
  get data() { return this.state.data; } // Getter&setter for "Immutable.js" data
  set data(data) { this.setState({ data }); }

  componentWillMount() { // Called before component is mounted into DOM
    this.data = this.data.set('loading', this.props.loading);
  }
  // This is where we should perform asynchronous
  // behavior that will change the state of the component.
  componentDidMount() {
    loadUsers().then(result => {
        this.data = this.data.set('loading', null)
          .set('error', null)
          .set('users', fromJS(result.users));
      }, error => {
        this.data = this.data.set('loading', null)
          .set('error', fromJS(error));
      });  } }
```

loading...

- First
- Second
- Third

```
render() {
  return (
    <UserList
      {...this.data.toJS()} />
  );
}
```

UPDATING STATE WITH PROPERTIES

MyButton.js:

```
export default ({clicks, disabled, text, onClick }) => (
   <section>
      <p>{clicks} clicks</p> { /* Renders the number of button clicks */ }
      <button disabled={disabled} onClick={onClick} > {text} </button>
   </section>
);

class MyFeature extends Component {
 constructor() { super();
      this.state = {data: fromJS({clicks: 0,
            disabled: false,  text: ''})}
       this.onClick = () => {this.data = this.data
          .update('clicks', c => c + 1); }}}

 componentWillMount() {
      this.data = this.data.set('text', this.props.text); }

 componentWillReceiveProps({ disabled }) {
      this.data = this.data.set('disabled', disabled); }

 render() { return (
     <MyButton onClick={this.onClick} {...this.data.toJS()} />
 )}
```

```
let disabled = true;
function renderApp() {
   disabled = !disabled;
   render(
      (<MyFeature
            {...{ disabled }} />),
      document.
            getElementById('app')
   );
}
// Re-render the "<MyFeature>"
// component every 3 seconds,
// toggling the "disabled" button
setInterval(renderApp, 3000);
renderApp();
```

TO RENDER OR NOT TO RENDER

```
export default class MyList extends Component {
  constructor() { super();
    this.data = fromJS({items: new Array(5000)
          .fill(null).map((v, i) => i)})   }

  // If this method returns false, the component
  // will not render. Since we're using an Immutable.js
  // data structure, we simply need to check for equality.
  shouldComponentUpdate(props, state) {
    return this.data !== state.data;   }

  // Renders the complete list of items, even if it's huge.
  render() {
    const items = this.data.get('items');
    return (
      <ul>
        {items.map(i => (
          <li key={i}>{i}</li>
        ))}
      </ul>
    ); }
}
```

```
function renderApp() {
  const myList = render(
    (<MyList />),
    document.
      getElementById('app')
  );
  // Immutable.js recognizes
  // that nothing changed,
  // and instead of
  // returning a new object,
  // it returns the same
  // "myList.data" reference.
  myList.data = myList.data
    .setIn(['items', 0], 0);
}
// Instead of performing
// 500,000 DOM operations,
// "shouldComponentUpdate()"
// turns this into
// 5000 DOM operations.
for (let i = 0; i < 100; i++) {
  renderApp();
}
```

USING METADATA TO OPTIMIZE RENDERING

```
export default class MyUser extends Component {
   state = { modified: new Date(), first: 'First', last: 'Last' };

   // The "modified" property is used to determine
   // whether or not the component should render.
   shouldComponentUpdate(props, state) {
      return +state.modified > +this.state.modified;
   }

   render() {
      const {modified,first,last} = this.state;
      return (
         <section>
            <p>{modified.toLocaleString()}</p>
            <p>{first}</p>
            <p>{last}</p>
         </section>
      );
   }
}
```

```
12/30/2016, 8:33:42 AM

First1

Last1
```

```
// initial rendering
const myUser = render(
   (<MyUser />),
   document.
      getElementById('app')
);

// modified
myUser.setState({
   modified: new Date(),
   first: 'First1',
   last: 'Last1',
});

// not modified
myUser.setState({
   first: 'First2',
   last: 'Last2',
});
```

WRAPPING JQUERY UI WIDGET INTO REACT COMPONENT

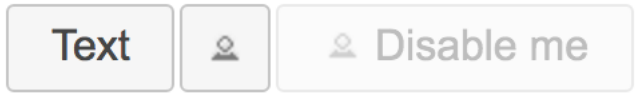
```
import $ from 'jquery';
import 'jquery-ui/ui/widgets/button';
import 'jquery-ui/themes/base/all.css';
export class MyButton extends Component {
  componentDidMount() { //initialize the widget
    $(this.button).button(this.props);
  }
  componentDidUpdate() {
    $(this.button).button('option', this.props);
  }
  render() {
    return (
      <button onClick={this.props.onClick}
              ref={button => { this.button = button; }} />
    ); }
}
MyButton.defaultProps = { onClick: () => {} };
export default MyButton;
```

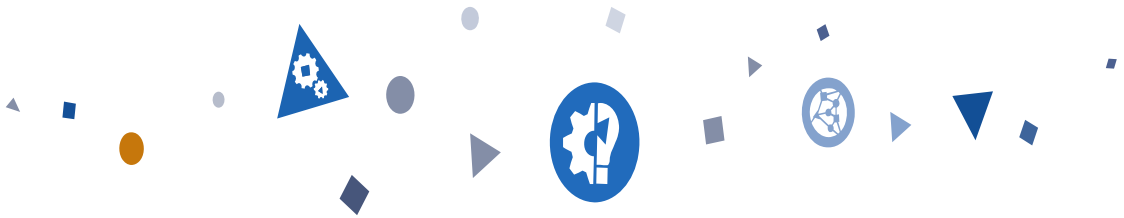
```
render((
<div>
  <MyButton label="Text" />

  <MyButton
    label="My Button"
    icon="ui-icon-person"
    showLabel={false}
  />
</div>
), document
.getElementById('app'));
```

CHANGING PROPS WITH RE-RENDERING

```
render((
      <div>
        <MyButton label="Text" />
        <MyButton
          label="My Button"
          icon="ui-icon-person"
          showLabel={false}
        />
        <span>
          <MyButton
            label="Disable me"
            icon="ui-icon-person"
            onClick={onClick}
          />
        </span>
      </div>
   ), document.getElementById('app')
);
```

```
// Simple button event handler
// that changes "disabled"
// state when clicked.
function onClick() {
   let options = Object.assign(
      {}, this.props, {disabled:true});

   render(
      <MyButton {...options} />,
      this.button.parentElement
   );
}
```

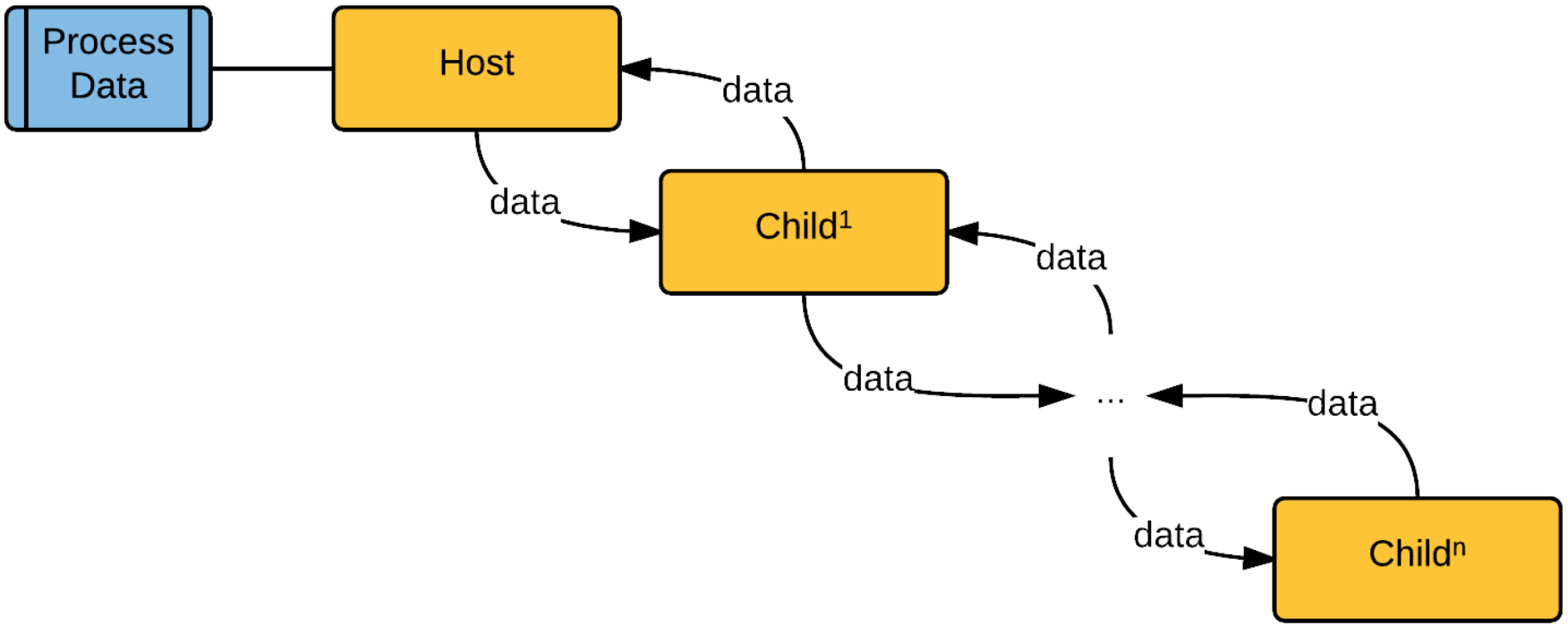




FLUX ARCHITECTURE

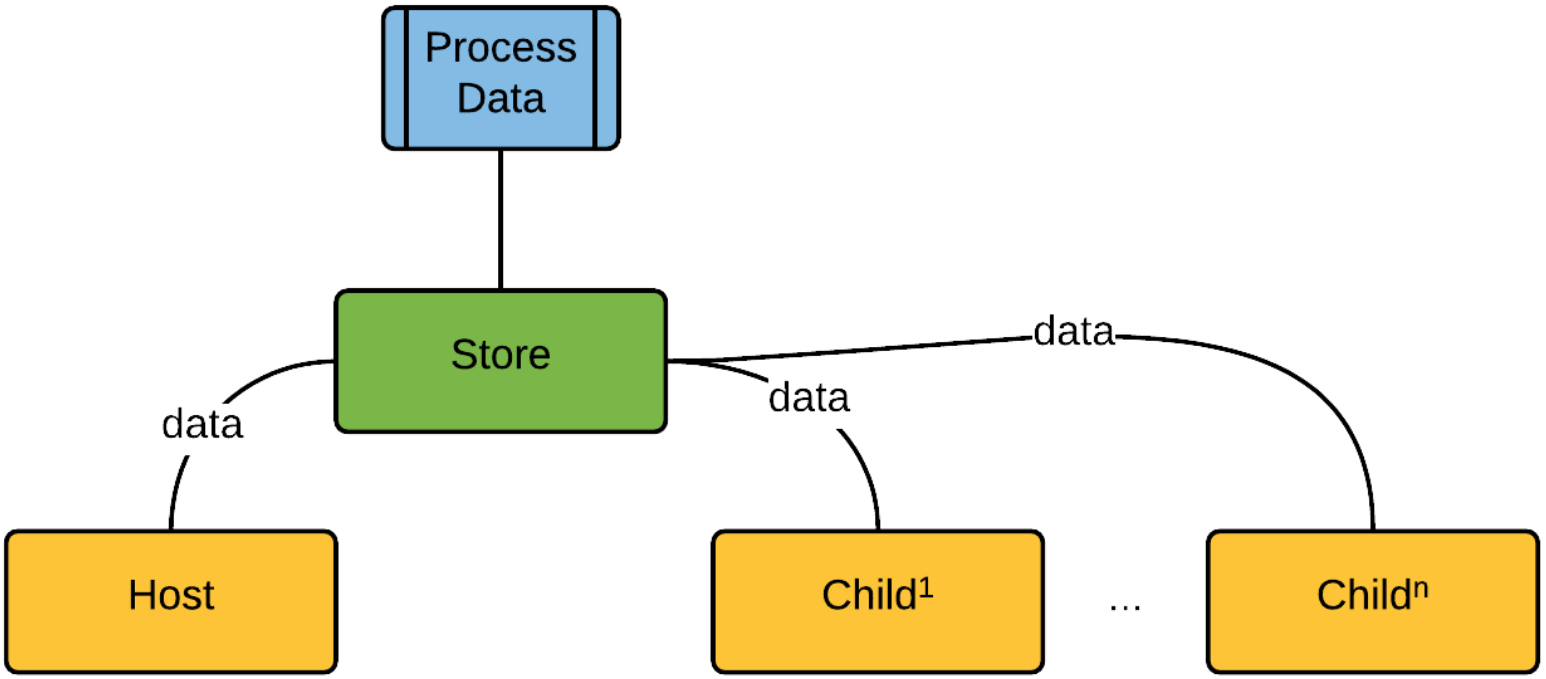
TRADITIONAL DATA-BINDING

Traditional data-binding (may it one-way, or two-way) leads to a chain of re-passing data down- and upwards the component tree:



THE STORE

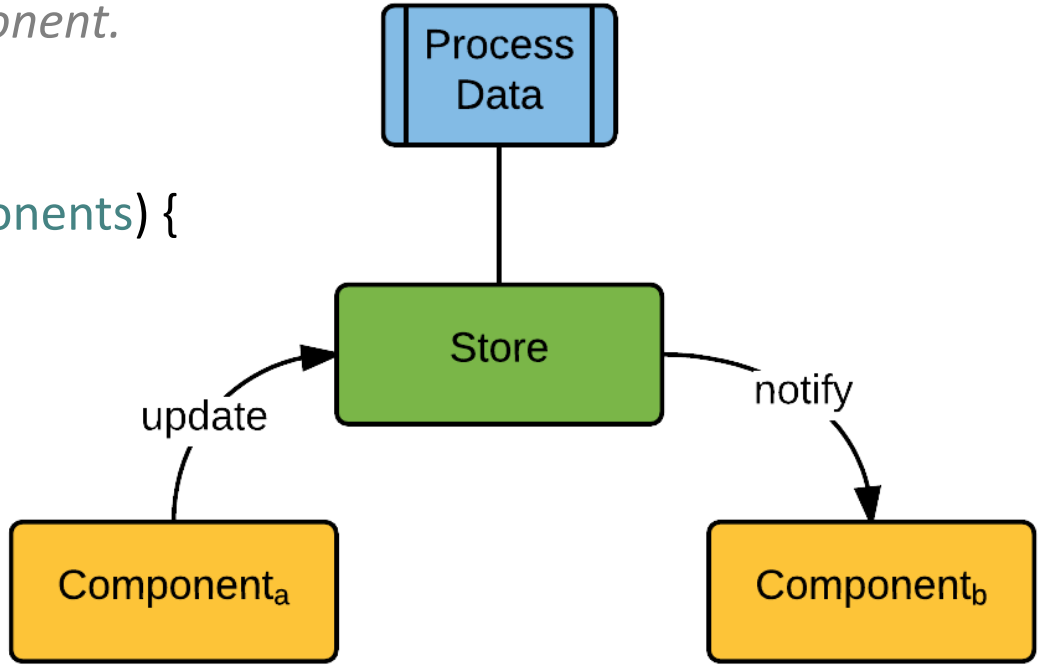
Store maintains one or more models, that's one reason why it is called Store and not model



Data stored in a store represents an **application state**.
You can think of an application state as **everything that need to be visualized directly or indirectly**.

USING STORE AS SHAREABLE DATA

```
// The components that are connected to this store.
let components = fromJS([]);

// The state store itself, where application data is kept.
let store = fromJS({});

// Sets the state of the store, then sets the
// state of every connected component.
export function setState(state) {
   store = state;
   for (const component of components) {
      component.setState({
         data: store,
      });
   }
}
// Returns the state of the store.
export function getState() {
   return store;
}
```

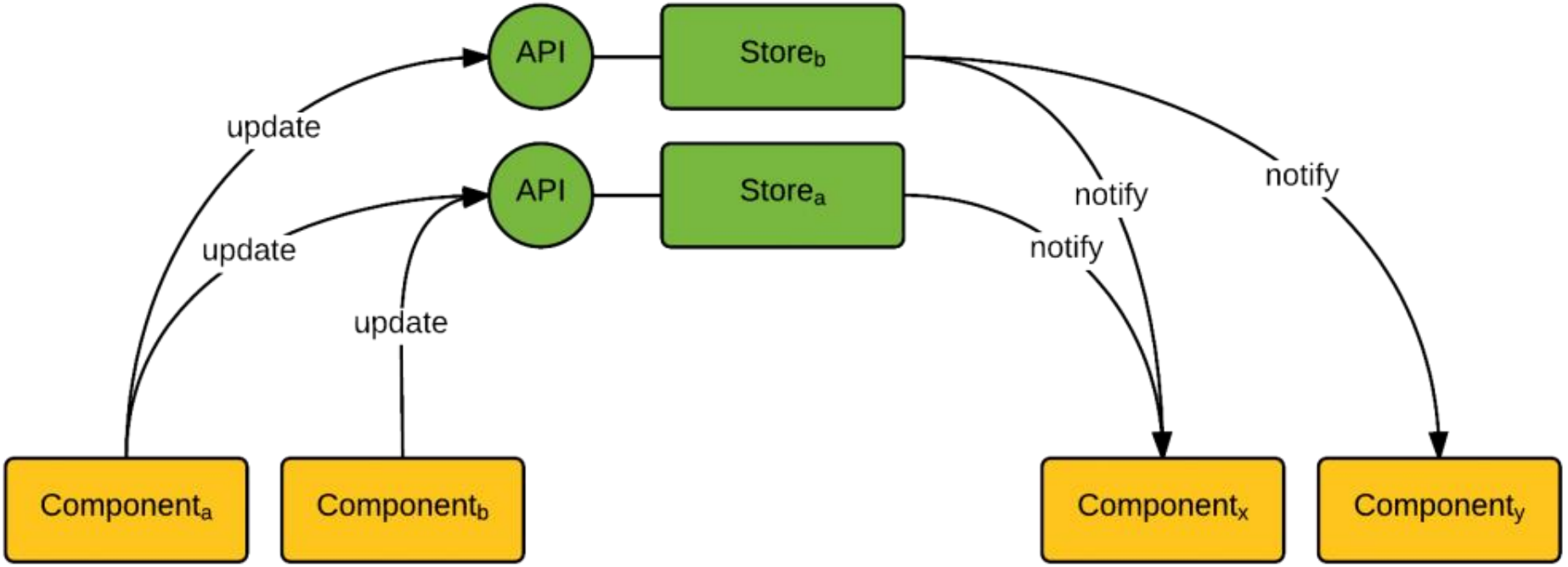


CONNECT FUNCTION TO CONNECT COMPONENT TO THE STORE

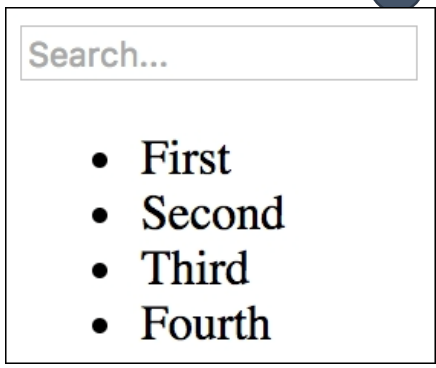
```
// Returns a higher-order component that's connected to the "store".
export function connect(ComposedComponent) {
  return class ConnectedComponent extends Component {
    state = { data: store }
    // When the component is mounted, add it to "components", so that it will
    // receive updates when the store state changes.
    componentWillMount() {
      components = components.push(this);
    }
    // Deletes this component from "components" when it is unmounted from DOM
    componentWillUnmount() {
      const index = components.findIndex(this);
      components = components.delete(index);
    }
    // Renders "ComposedComponent", using the "store" state  as properties.
    render() {
      return (<ComposedComponent {...this.state.data.toJS()} />);
    }
  }; }
```

STORES

Commonly, a Store provides a set of methods, for example add(), update(), delete() to update its models. So, each Store may have its own interface.



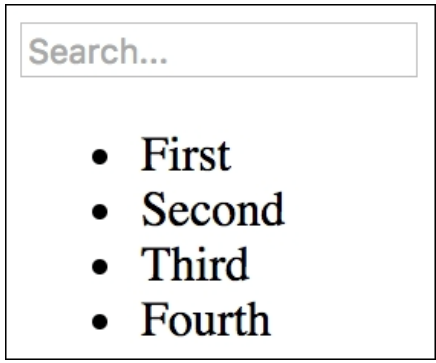
EXAMPLE: LIST WITH FILTER




```
// Renders a simple input element to filter a list.
const MyInput = ({ value, placeholder }) => (
  <input
    autoFocus
    value={value}
    placeholder={placeholder}
    onChange={onChange}
  />
);

MyInput.propTypes = {
  value: PropTypes.string,
  placeholder: PropTypes.string,
};
```

```
// Renders an item list...
const MyList = ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    ))}
  </ul>
);

MyList.propTypes = {
  items:
    PropTypes.array.isRequired,
};
```

IMPLEMENTING ONCHANGE




```
// When the filter input value changes.
function onChange(e) {
  const state = getState(); // The state that we're working with...
  const items = state.get('items');
  const tempItems = state.get('tempItems');
  // The new state that we're going to set on the store.
  let newItems;
  let newTempItems;
  if (e.target.value.length === 0) { // If input value is empty, restore from tempItems
    newItems = tempItems;
    newTempItems = fromJS([]);
  } else {
    if (tempItems.isEmpty()) newTempItems = items;
    else newTempItems = tempItems;
    // Filter and set "newItems".
    const filter = new RegExp(e.target.value, 'i');
    newItems = items.filter(i => filter.test(i));
  }
```

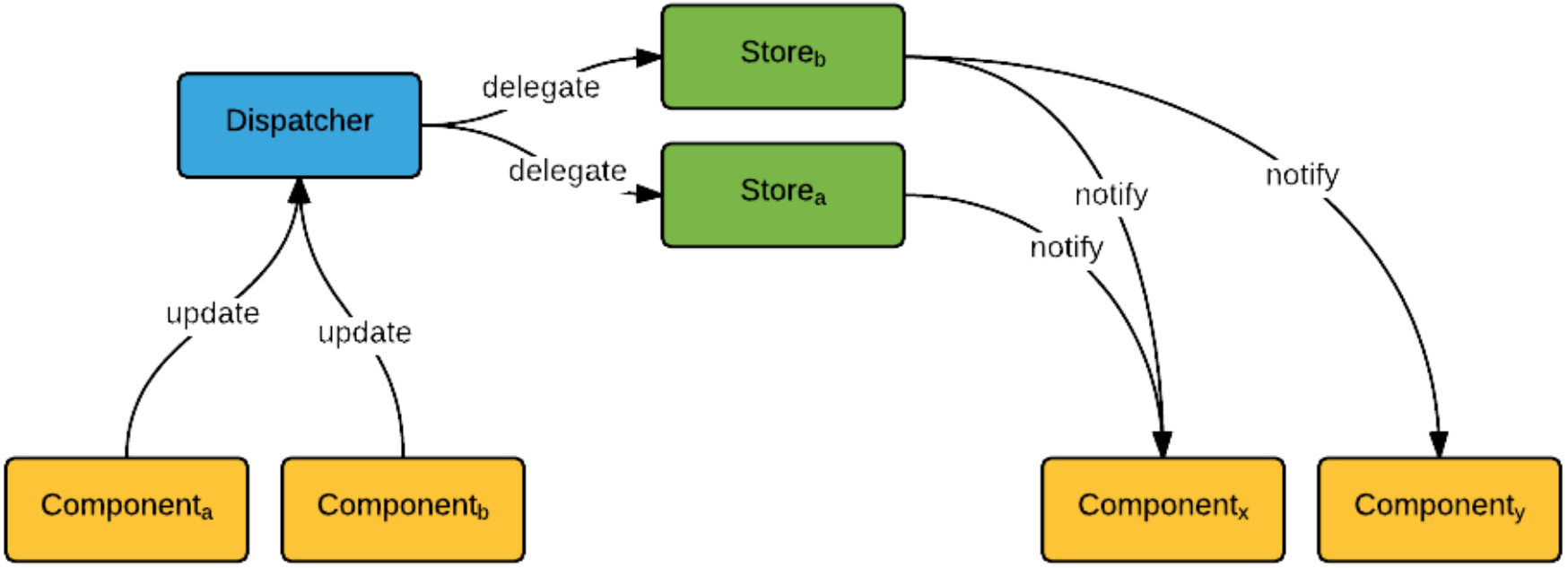
```
// Updates the state of the store.
setState(state.merge({
  items: newItems,
  tempItems: newTempItems,
}));
```

RENDERING

```
// Compose the "connected" versions of "MyInput" and
// "MyList", so that they automatically receive updates
// when the store changes state.
const ConnectedInput = connect(MyInput);
const ConnectedList = connect(MyList);
// Setup the default store state...
setState(getState().merge({
   placeholder: 'Search...',
   items: ['First', 'Second', 'Third', 'Fourth'],
   tempItems: [],
}));
render((
      <section>
         <ConnectedInput />
         <ConnectedList />
      </section>
   ), document.getElementById('app')
);
```

USING DISPATCHER

The Dispatcher delegates/propagates the update actions to the related Store.



DISPATCHER

```
let appDispatcher = new Dispatcher();

export function filterList( value ) {
    let payload = {
        type: 'FILTER_LIST',
        filter: value
    }
    appDispatcher.dispatch(payload);
}
```

in MyInput.js:

```
// When the filter input value changes.
function onChange(e) {
    filterList(e.target.value);
}
```

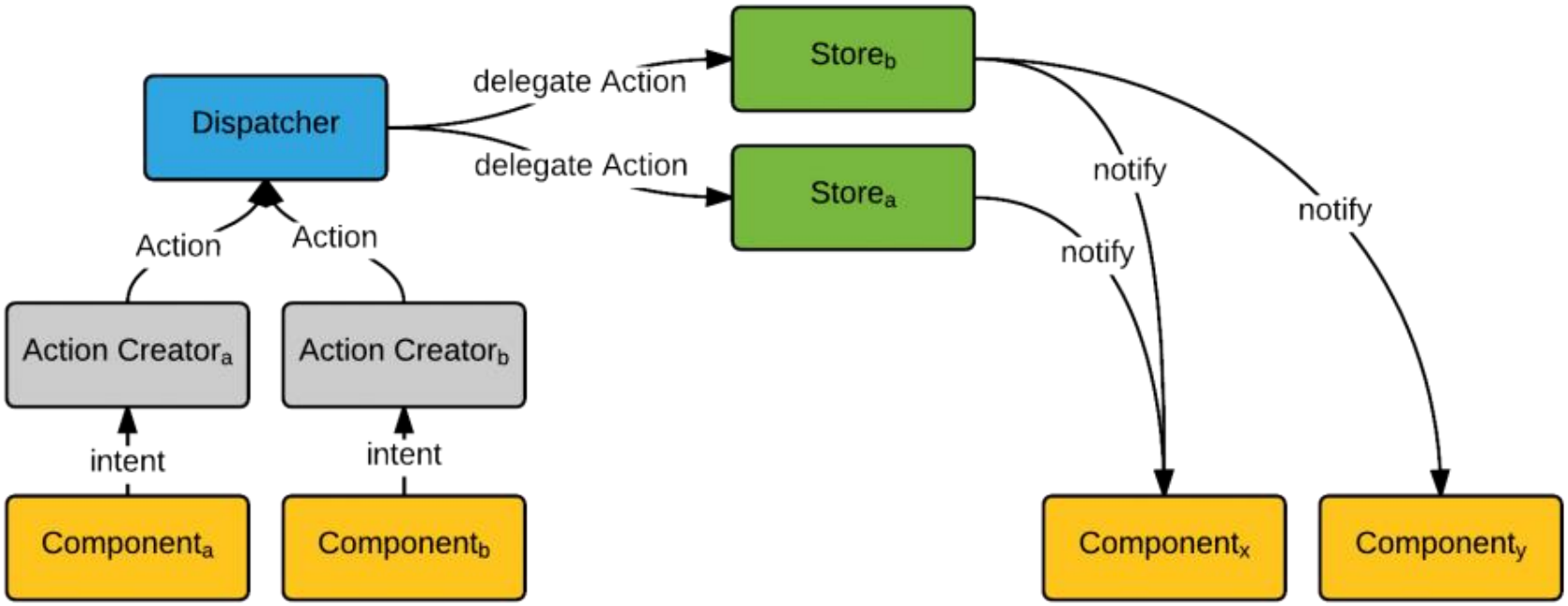
```
export class Dispatcher {
    callbacks = {};
    lastId = 0;
    register(callback) {
        let id = this.lastId;
        this.lastId++;
        this.callbacks[id] = callback;
    }
    unregister(id) {
        delete this.callbacks[id];
    }
    dispatch(payload) {
        for (let id in this.callbacks) {
            let callback =
                this.callbacks[id];
            callback(payload);
        }
    }
}
```

DISPATCHER CALLBACK PROCESSING FILTER_LIST

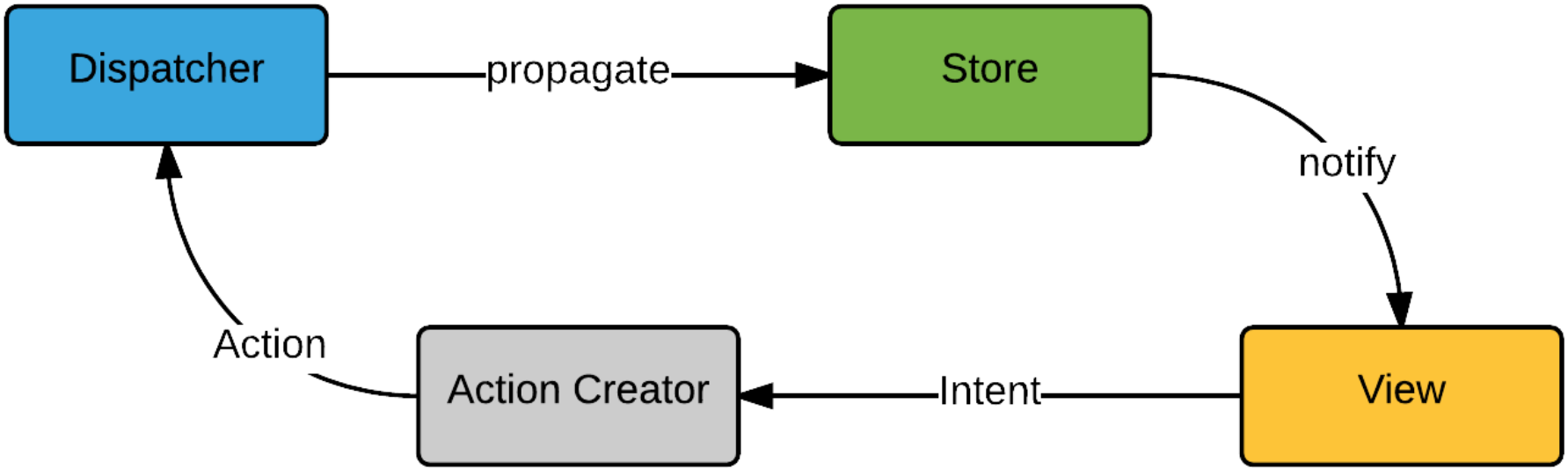
```
appDispatcher.register(function(payload) {
    switch(payload.type) {
        case "FILTER_LIST":
            const state = getState();
            const items = state.get('items');
            const tempItems = state.get('tempItems');
            let newItems, newTempItems;

            if (payload.filter.length === 0) { // restore from tempItems
                newItems = tempItems;
                newTempItems = fromJS([]);
            } else {
                if (tempItems.isEmpty()) newTempItems = items;
                else newTempItems = tempItems;
                // Filter and set "newItems".
                const filter = new RegExp(payload.filter, 'i');
                newItems = items.filter(i => filter.test(i));
            }
          setState(getState().merge({ // Updates state of the store
                items: newItems,
                tempItems: newTempItems,
            }));
            break;
        }});
```

ACTIONS AND ACTION CREATOR

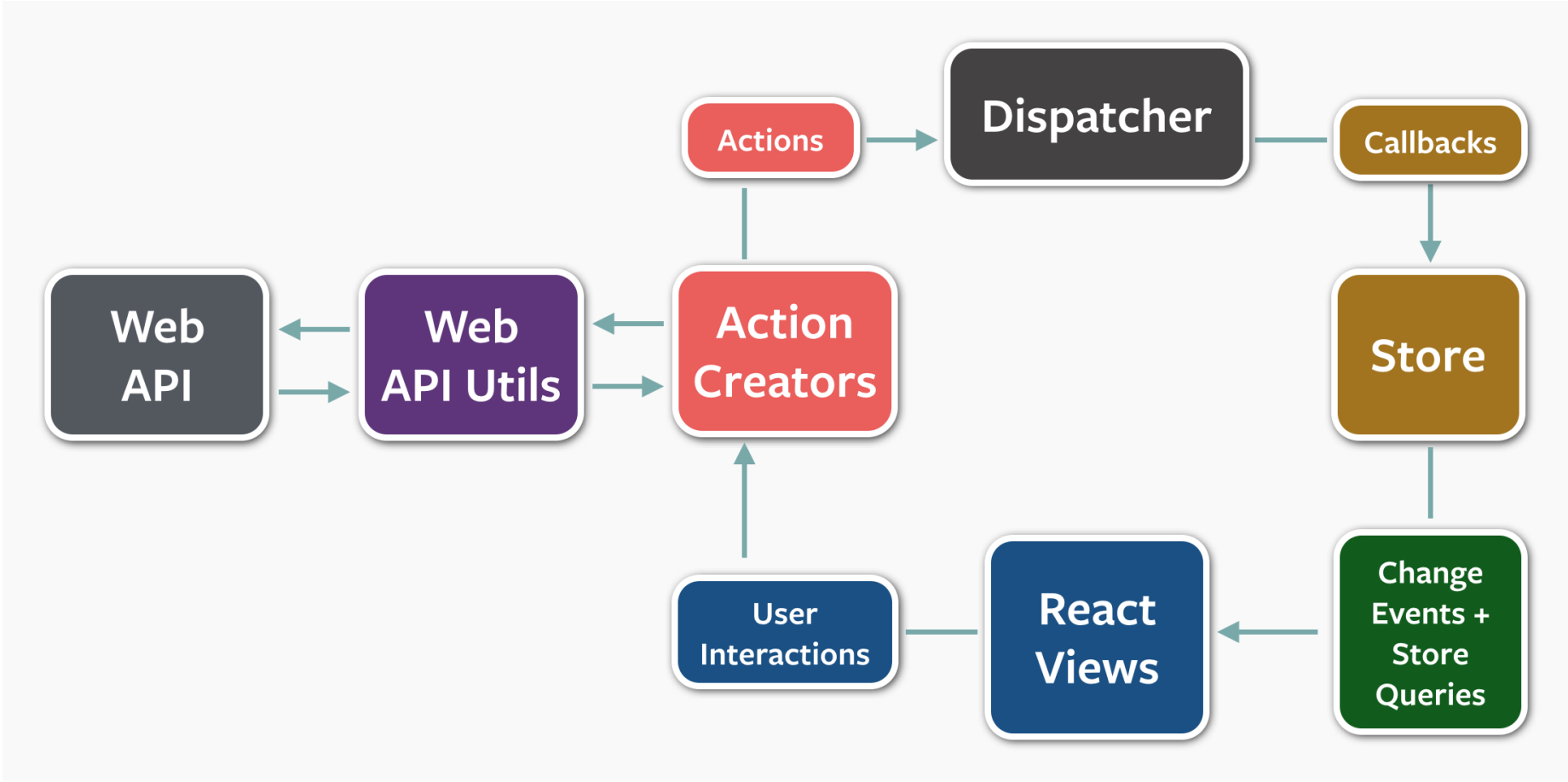
An Action is considered as an object carrying usually an identifier and payload data, that will be propagated via the Dispatcher towards the targeted Store.



FLUX ARCHITECTURE



FLUX ARCHITECTURE



FLUX IMPLEMENTATIONS