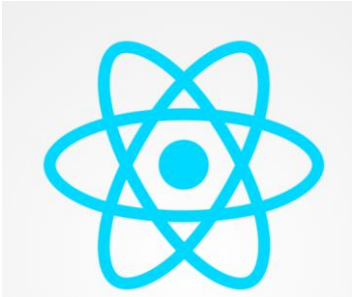
React.js



Banuprakash C

banuprakashc@yahoo.co.in

banu@lucidatechnologies.com

React.js Fundamentals

- What is React.js?
  - Developed by Facebook
  - React is a view layer library, not a framework like Backbone, Angular etc.
  - You can't use React to build a fully-functional web app
- Why was React developed?
  - Complexity of two-way data binding
  - Bad UX from using "cascading updates" of DOM tree
  - A lot of data on a page changing over time
  - Complexity of Facebook's UI architecture
  - Shift from MVC mentality

Who uses React?

https://github.com/facebook/react/wiki/Sites-Using-React

React: the good

- Easy to understand what a component will render
  - Declarative code → predictable code
  - You don't really need to study JS in the view file in order to understand what the file does
- Easy to mix HTML and JS
  - You do it already with template libraries (e.g. Handlebars, Mustache, Underscore etc.)
- Uses full power of JS
  - Decoupling templates from logic does not rely on the templates' primitive abstractions, but uses full power of JavaScript in displaying views
- No complex two-way data flow
  - Uses simple one-way reactive data flow
  - Easier to understand than two-way binding
  - Uses less code
- React dev tools
  - React Chrome extension makes debugging so much easier

React: the good

- React is fast!
  - Real DOM is slow
  - Using virtual DOM objects enables fast batch updates to real DOM, with great productivity gains over frequent cascading updates of DOM tree
- Server-side rendering
  - React.renderToString() returns pure HTML

React: the bad

- React is nothing but the view
  - No events
  - No XHR
  - No data / models
  - No promises / deferreds
  - No idea how to add all of the above
- Very little info in the docs
- Building JSX requires some extra work

Fundamentals

- Component
  - Components are self-contained reusable building blocks of web application.
  - React components are basically just idempotent functions (same input produces same output).
- **Function and Class Component**

```
function Welcome() {
  return <h1>Hello reader</h1>;
}
```

- **ES6 class Component**

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello reader</h1>;
  }
}
```

```
<header className="App-header">
  < Welcome />
</header>
```

Fundamentals

- Props
  - Passed down to component from parent component and represents data for the component
  - accessed via this.props

State

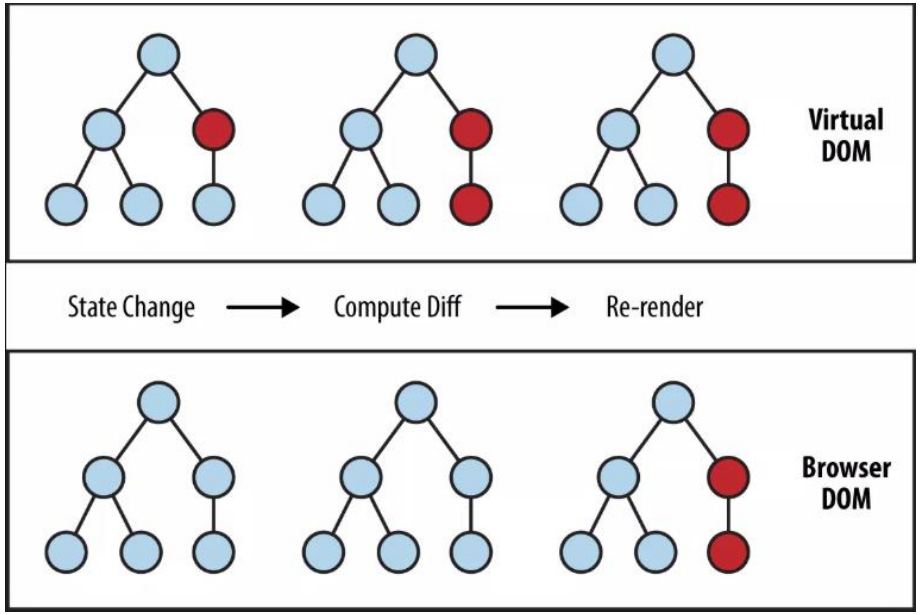
- Represents internal state of the component
- Accessed via this.state
- When a component's state data changes, the rendered markup will be updated by re-invoking render() method

JSX

- XML-like syntax for generating component's HTML
- Easier to read and understand large DOM trees

Virtual DOM

- The virtual DOM is used for efficient re-rendering of the DOM
- React aims to re-render the virtual tree only when the state changes
- Uses 2 virtual trees (new and previous) to find differences and batch update real DOM
- Observes data changes and does dirty-checking to know when to re-render component
- Does not update entire component in real DOM
  - only computes a patch operation that updates part of the DOM



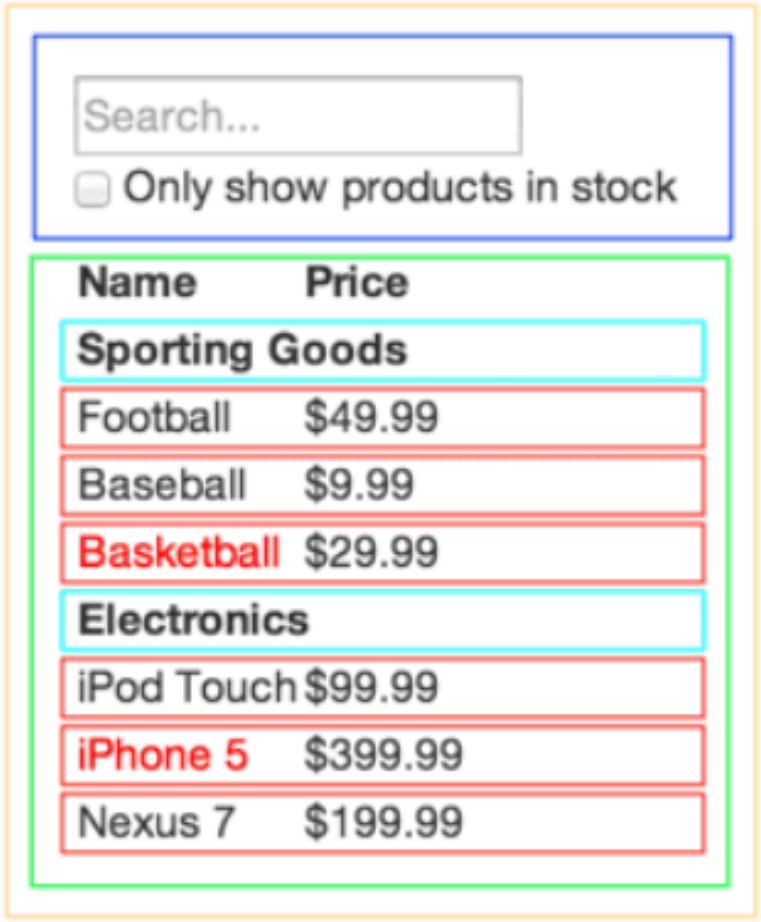
Thinking in React

- **Start With A Mock**
  - Our JSON API returns some data that looks like this:

```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Thinking in React

- **Break The UI Into A Component Hierarchy**
  - FilterableProductTable (orange): contains the entirety of the example
  - SearchBar (blue): receives all user input
  - ProductTable (green): displays and filters the data collection based on user input
  - ProductCategoryRow (turquoise): displays a heading for each category
  - ProductRow (red): displays a row for each product



Thinking in React

- Arrange them into a hierarchy
    - `FilterableProductTable`
        - `SearchBar`
        - `ProductTable`
            - `ProductCategoryRow`
            - `ProductRow`

Thinking in React

- **Build A Static Version in React**
  - **don’t use state at all** to build this static version.
  - State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don’t need it
  - You can build top-down or bottom-up.
    - That is, you can either start with building the components higher up in the hierarchy (i.e. starting with FilterableProductTable) or with the ones lower in it (ProductRow).
    - In simpler examples, it’s usually easier to go top-down, and on larger projects, it’s easier to go bottom-up and write tests as you build

Thinking in React

- Think of all of the pieces of data in our example application. We have:
  - The original list of products
  - The search text the user has entered
  - The value of the checkbox
  - The filtered list of products
- Ask three questions about each piece of data:
  - Is it passed in from a parent via props? If so, it probably isn't state.
  - Does it remain unchanged over time? If so, it probably isn't state.
  - Can you compute it based on any other state or props in your component? If so, it isn't state.
- The original list of products is passed in as props, so that's not state.
- The search text and the checkbox seem to be state since they change over time and can't be computed from anything.

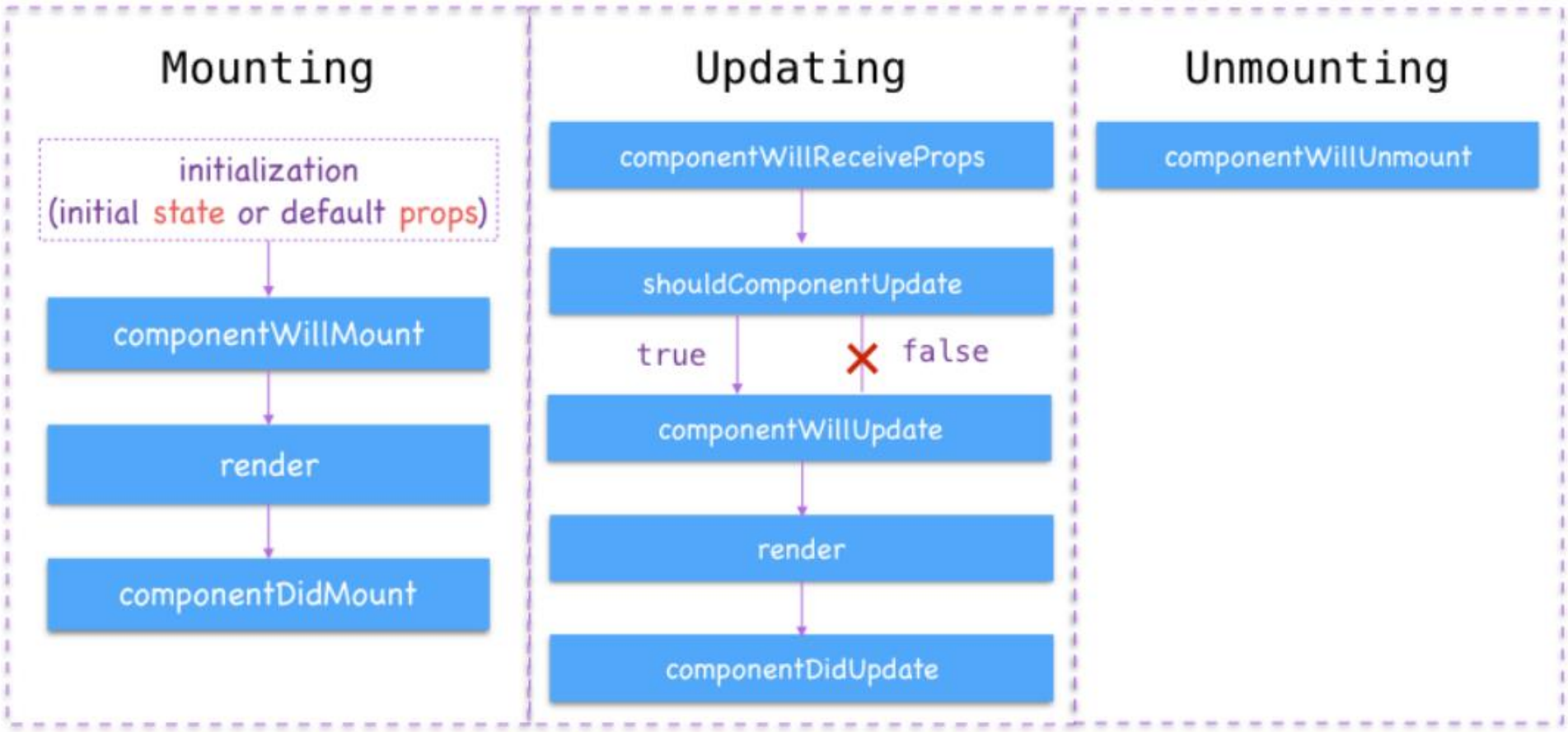
Thinking in React

- **Identify Where Your State Should Live**
  - Let’s run through this strategy for our application:
  - ProductTable needs to filter the product list based on state and SearchBar needs to display the search text and checked state.
  - The common owner component is FilterableProductTable.
  - It conceptually makes sense for the filter text and checked value to live in FilterableProductTable
  - First, add an instance property
  - this.state = {filterText: '', inStockOnly: false} to FilterableProductTable’s constructor to reflect the initial state of your application

Thinking in React

- **Add Inverse Data Flow**
  - props and state flowing down the hierarchy.
  - the form components deep in the hierarchy need to update the state in FilterableProductTable.

Lifecycle Methods



Mounting

- constructor: when a component is created
  - Do basic state initialization here
- componentDidMount: called after a component has finished mounting
  - AJAX calls that can cause component re-renders should go here
- componentWillMount: called during server rendering
  - Use constructor otherwise.

Updating

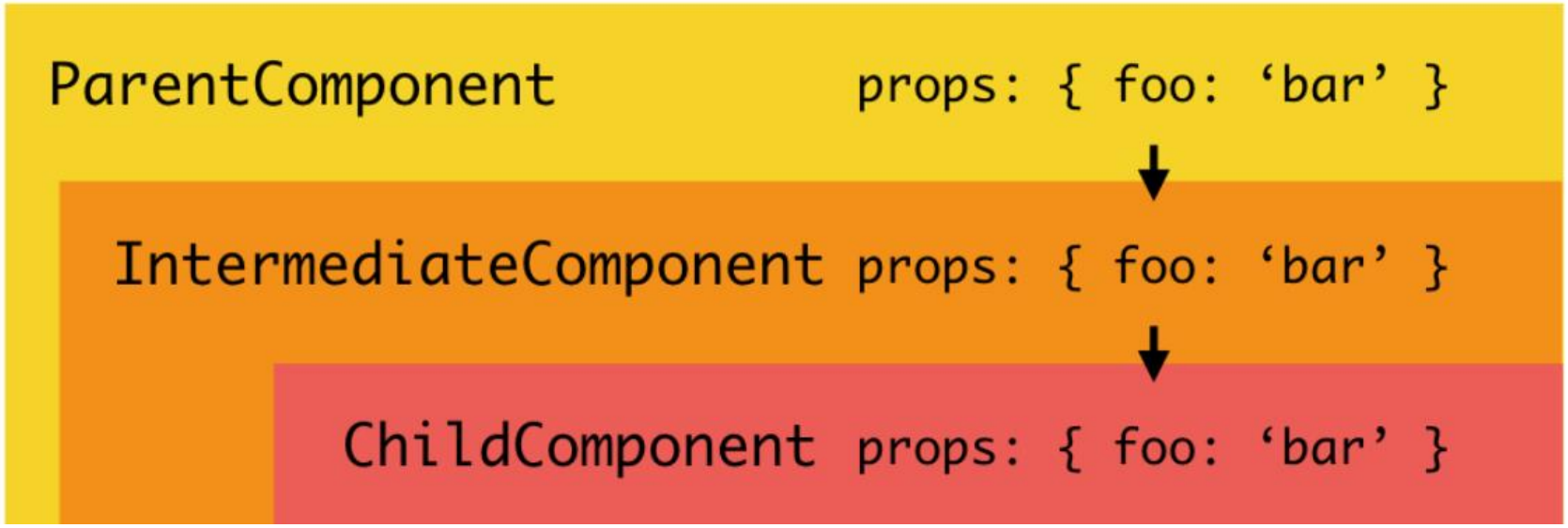
- shouldComponentUpdate: called after a component's props or state has changed.
  - Decides whether or not a component should re-render
  - Main use is performance optimization
- componentWillUpdate and componentDidUpdate: called before and after a component re-renders
  - Any manual work done outside of React when updates happen should be done here
  - E.g., encapsulation of 3rd party UI libraries within a component
- componentWillReceiveProps: called before a component has received props whose values have changed

Lifecycle Methods

- Do basic state initialization in constructor.
- Use componentDidMount to run actions after all components are created.
- Use shouldComponentUpdate to control re-rendering.
- Use componentWillUpdate and componentDidUpdate for pre- and post-update actions.
- Use componentWillUnmount for end-of-life cleanup.

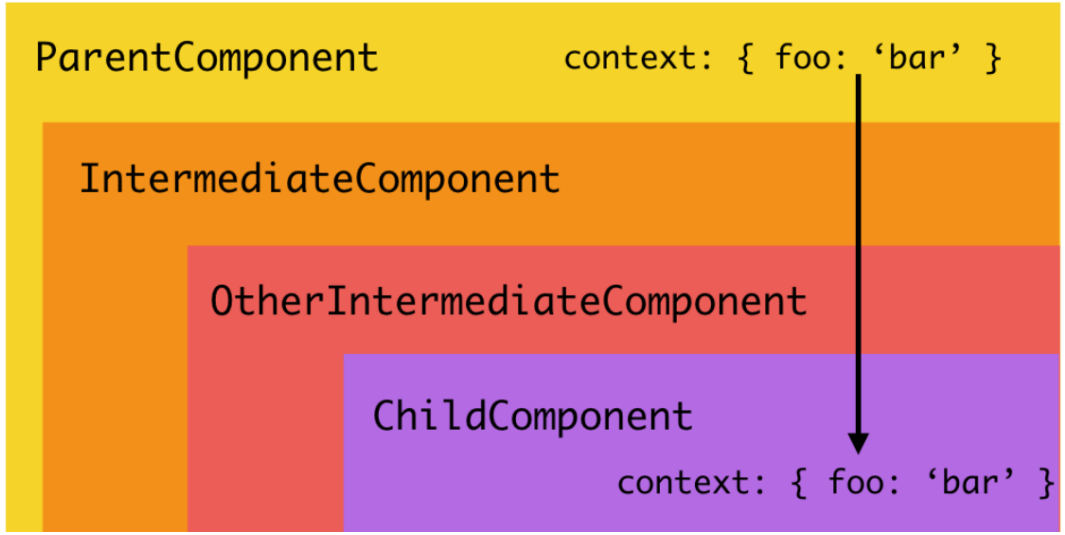
Props

- In React the primary mechanism for communication between your components is through properties, or props, for short



Context

- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language
- Context acts like a portal in your application in which components can make data available to other components further down the tree without being passed through explicitly as props



Context: Create provider and consumer components

```
import React, { createContext } from 'react';

const UserContext = createContext({
  username: '',
  updateUsername: () => {},
});

export class UserProvider extends React.Component {
  updateUsername = newUsername => {
    this.setState({ username: newUsername });
  };

  state = {
    username: 'user',
    updateUsername: this.updateUsername,
  };
```

```
  render() {
    return (
      <UserContext.Provider value={this.state}>
        {this.props.children}
      </UserContext.Provider>
    );
  }
}

export const UserConsumer = UserContext.Consumer;
```

Context: using the Provider

- The <UserProvider> component needs to wrap around all components that share state.

```
function App() {
  return (
    <UserProvider>
      <UserMessage />
      <SettingsForm />
    </UserProvider>
  );
}
```

Context: Using the consumer to read state

```
export default function UserMessage() {
  return (
    <UserConsumer>
      {({ username }) => <h1>Welcome {username}!</h1>}
    </UserConsumer>
  );
}
```

Context: Using the consumer to update the state

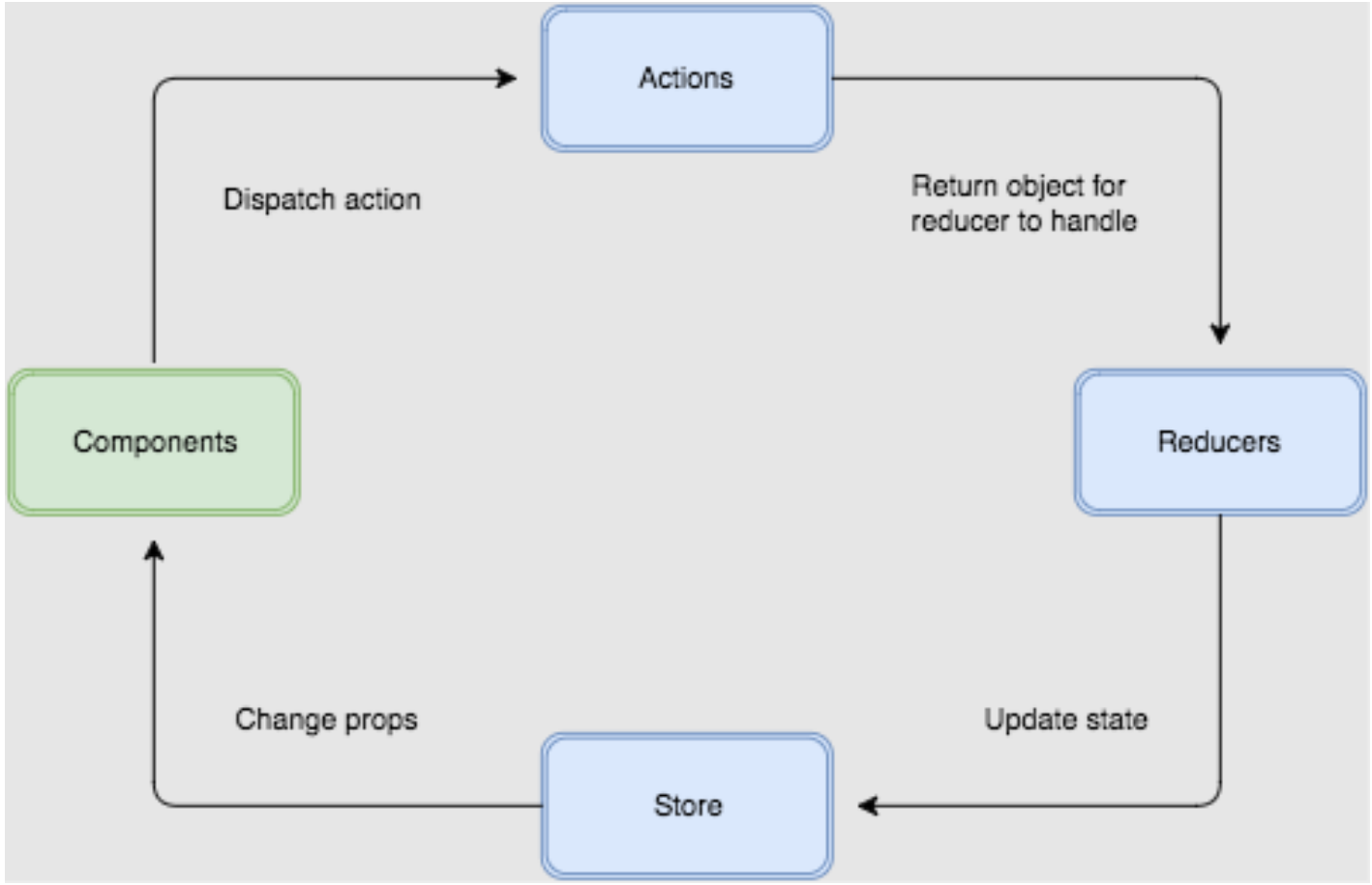
```
export default function UserSettings() {
  return (
    <UserConsumer>
      {({ updateUsername }) => (
        <div>
          <h2>Settings</h2>
          <label htmlFor="username">Username: </label>
          <input
            id="username"
            type="text"
            onChange={event => {
              updateUsername(event.target.value);
            }}
          />
        </div>
      )}
    </UserConsumer>
  );
}
```

Before You Use Context

- Context is primarily used when some data needs to be accessible by many components at different nesting levels.
- Apply it sparingly because it makes component reuse more difficult.

- If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

Redux


Actions
Dispatch action
Return object for
reducer to handle
Components
Reducers
Change props
Update state
Store

Redux Basics

- **State and** Store
- App state is stored in plain objects.
- Store
  - Where the entire application state exists
  - The store is an observable
- An application should only have one store
  - In larger applications, the store will have several parts

```
{
    todos: [
        {text: "Eat food", completed: true},
        {text: "Exercise", completed: false}
    ],
    visibilityFilter : "SHOW_COMPLETED"
}
```

Redux Basics

- Actions
- To change something in the state, you need to dispatch an action.
- An action is a plain JS object with a type field and payload

```
// Actions: plain objects with a "type" field
{ type: "ADD_TODO", text: "Go to swimming pool" }
{ type: "TOGGLE_TODO", index: 1 }
{ type: "SET_VISIBILITY_FILTER", filter: "SHOW_ALL" }
```

Redux Basics

- **Action Creators**
- It is common to use *action creator* functions to encapsulate the process of creating action objects. This may seem like overkill for simple use cases, but consistent use of action creators leads to cleaner code and better reusability. Action creators are not required, but are a good practice.

```
// Action creators: functions that return an action
function addTodo(text) {
    return {
        type : "ADD_TODO",
        text
    };
}
```

Redux:**Reducers**

- All state update logic lives in functions called **reducers**
- **Reducers need to update data immutably, by making copies of state and modifying the copies before returning them, rather than directly modifying inputs.**

```
function todosReducer(state = [], action) {
    switch (action.type) {
        case "ADD_TODO":
            // Instead of calling `state.push()`, which modifies the
            // existing array, use `state.concat()` to return a new array
            return state.concat([{
                text: action.text, completed: false
            }]);
        case "TOGGLE_TODO":
            // `map()` returns a new array, and
            // `{...someObject}` returns a new object
            return state.map((todo, index) => {
                if(index !== action.index) return todo;
                return {...todo, completed: !todo.completed }
            })
        case "REMOVE_TODO":
            return state.filter( (todo, index) => {
                return index !== action.index
            });
        default: return state;
    }
}
```

Example

- State
  - const INITIAL_STATE = 0
- Actions

```
const INCREASE = 'INCREASE'
const DECREASE = 'DECREASE'

function increase() {
  return { type: INCREASE }
}

function decrease() {
  return { type: DECREASE }
}
```

Example

- Reducer

```
function counterReducer(state = INITIAL_STATE, action = {}) {
  switch (action.type) {
    case INCREASE:
      return state + 1

    case DECREASE:
      return state - 1

    default:
      return state
  }
}
```

Example

- Connect to Redux

```
let { createStore, combineReducers } = Redux
const rootReducer = combineReducers({ counter: counterReducer })
const store = createStore(rootReducer)
```

Example

```
store.subscribe(() => {
  ReactDOM.render(
    <div>

      <pre>
        { JSON.stringify(store.getState(), null, 2) }
      </pre>

      <button
        onClick={ () => store.dispatch(increase()) }>
        Increase
      </button>

      <button
        onClick={ () => store.dispatch(decrease()) }>
        Decrease
      </button>

    </div>,
      document.getElementById('root')
  )
})
```