



React Fundamentals

Outline

1. Introduction
2. React Components
3. State
4. Components Communication
5. React Tools and Component Libraries

React Introduction



Used by Facebook, Instagram, Netflix, Dropbox, Outlook, Yahoo, Khan Academy,

<https://intellisoft.io/15-popular-sites-built-with-react-js/>

Web Dev Big Picture



Web Client

Request

Response



Web Server

Frontend development

HTML for page content and structure



CSS for styling



JavaScript for interaction



UI Components



Backend development

Web API



Data Management



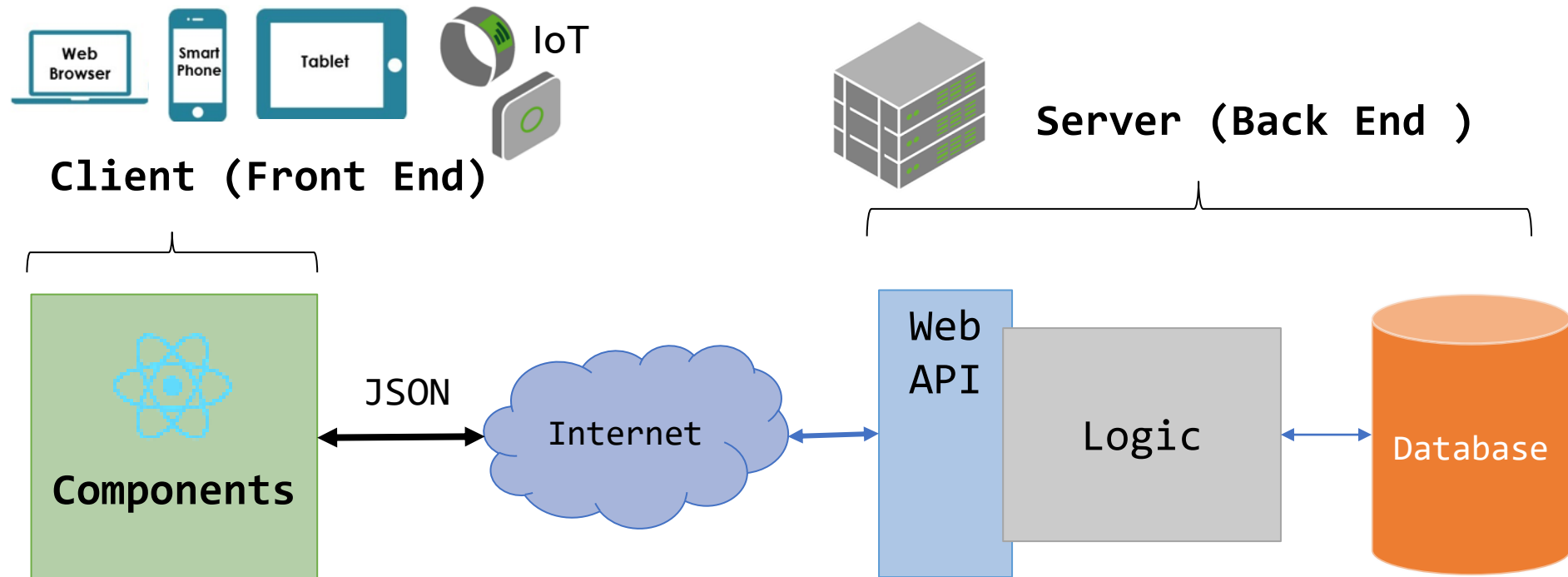
What is React?



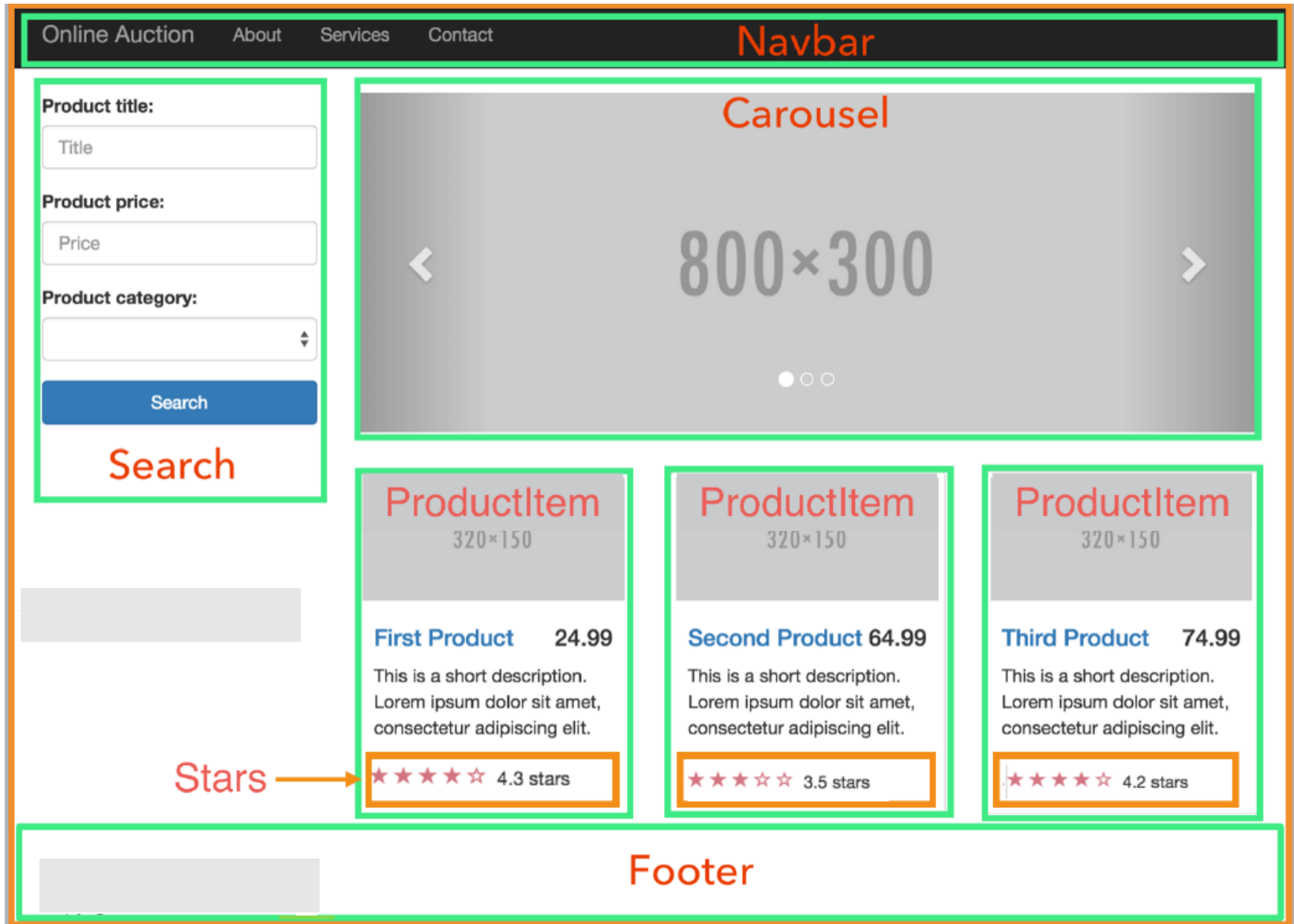
- React is an open-source JavaScript library for building **components-based user interfaces (UI)**
 - UI is **composed** of small reusable **components**
 - A component encapsulates **UI elements** and the **behavior** associated with them
- Ease creating a Single Page Application (SPA)
 - SPA is a Web app that load a single HTML page and **dynamically loads components** as the user interacts with the app
- Open-sourced by Facebook mid-2013 - <https://reactjs.org/>
- Competing with Angular <http://angular.io> and Vue.js <https://vuejs.org/>

Components of Single Page Application (SPA)

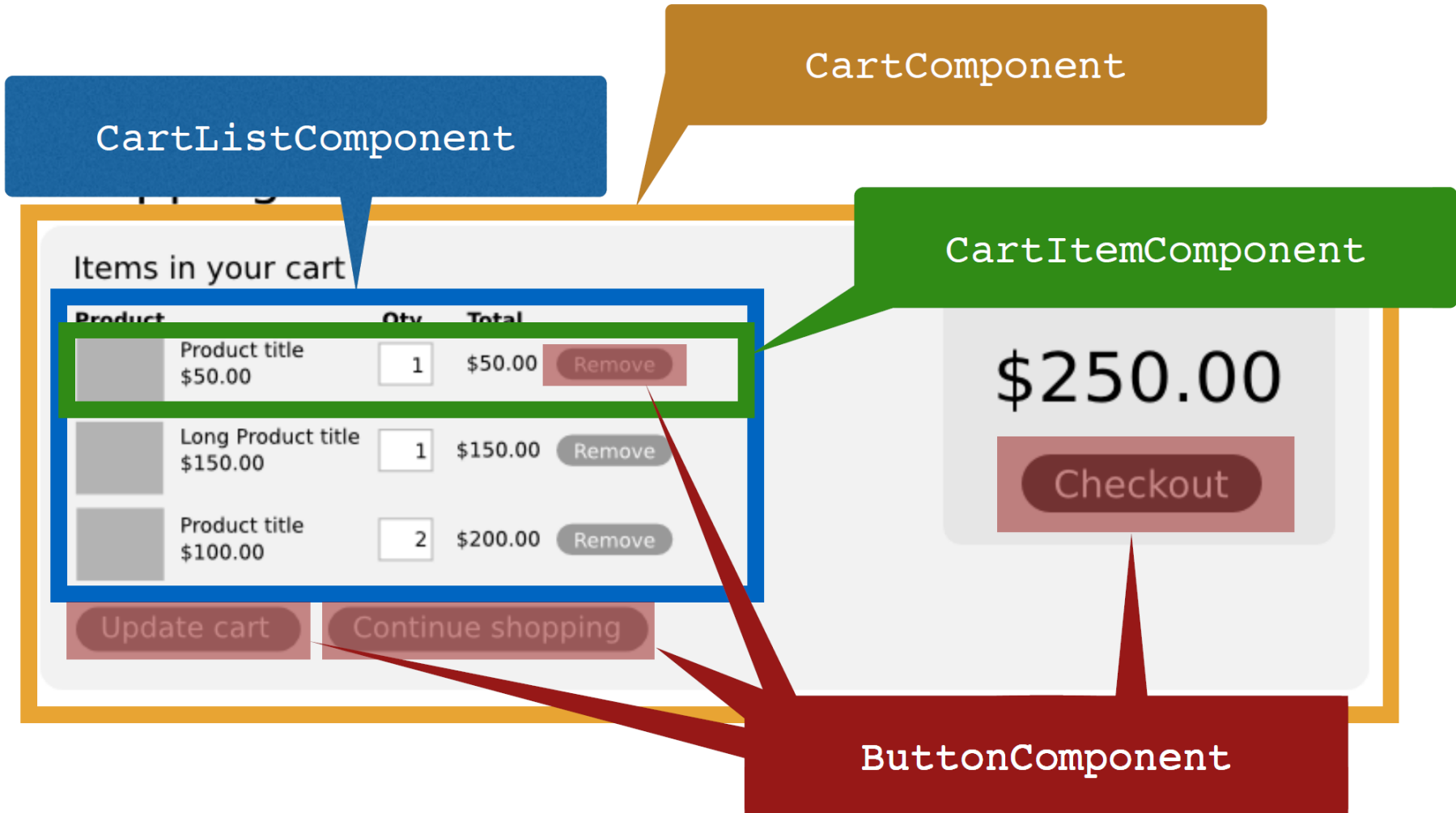
- A **Single-Page Application (SPA)** has **1 main shell page** and **multiple UI components loaded** in response to user actions



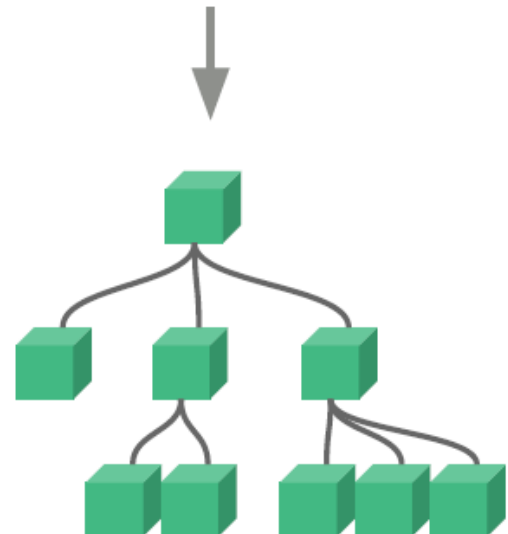
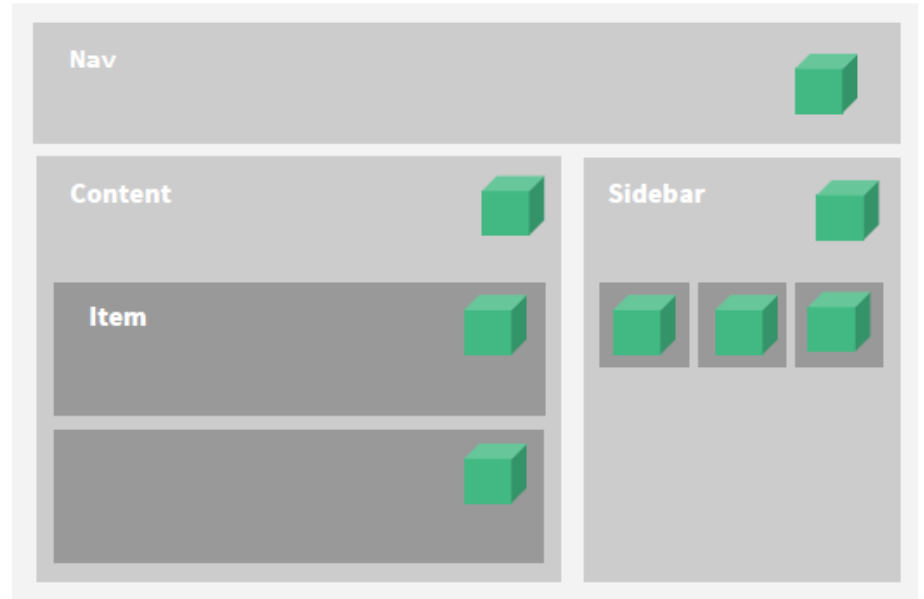
An app = a composition of components



An app = a tree of components



React Components



Getting started

- Install latest **Node.js** <https://nodejs.org/en/>
- Download **VS Code** <https://code.visualstudio.com/>
- Create an empty folder (with no space in the name use **dash** - instead)
- Create a react app

npx create-react-app .

- Run the app

npm start

React Component

- React App = composition of **components**
- A ***component***:
 - Return **HTML elements** to provide the UI
 - Encapsulate **state** (internal component data) and **functions** to ***handle events*** raised from the UI elements
- Component = UI + display logic
- Components allows creating new '**HTML tags**'

React = A declarative component-based programming model

- UI is built using JavaScript functions
 - Each function define a piece the app's UI programmatically
 - As **state** changes the UI automatically updates (Reactive UI)
 - without imperatively mutating DOM
- Declarative = you define the UI content and structure, combined with different states (e.g., "is a modal open or closed?")
 - Then you leave it up to React to figure out the appropriate DOM instructions



How to define a piece of UI?

UI is **composed** of small reusable **components**

UI Component = a **function**:

- Takes some inputs and emits a piece of UI
- Function that converts the state (i.e., app data) into UI



- **UI = f(state) : UI is a visual representation of state**
(e.g., display a tweet and associated comments)
- **State changes trigger automatic update of the UI**



Component Example

- Create a **Welcome** component
 - Returns **JSX** : an HTML-like syntax to define the component UI
 - Can accept a parameter called **props**
 - to configure the component with different content / attributes - just like how HTML works (makes the component reusable)
 - **props** are read-only
 - Component name must start with a capital letter

```
import React from "react";  
function Welcome(props) {  
  return (<h1>Welcome to {props.appName}</h1>);  
}  
export default Welcome;
```

You can embed JavaScript expressions in JSX

- Use the **Welcome** component

```
<Welcome appName='React Demo App' />
```

What is JSX?

- React uses JSX (JavaScript XML) HTML-like markup to describe the component's UI
- Embraces the fact that rendering logic is inherently coupled with other UI logic
- JSX allows us to write HTML like syntax which gets transformed to JavaScript objects

JSX

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
);
```

JavaScript

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hello, world!'  
);
```

It's just JavaScript!!

Props destructuring

- In a **react** component you can destructure **props into variables**

```
function UserInfo(props) {  
  return (  
    <div>  
      First Name: {props.firstName}  
      Last Name: {props.lastName}  
    </div>  
  );  
}
```

Becomes



```
function UserInfo({ firstName, lastName }) {  
  return (  
    <div>  
      First Name: {firstName}  
      Last Name: {lastName}  
    </div>  
  );  
}
```


Special "children" Prop

- The children property holds the content you might have provided between the component's opening and closing tags
 - A special children property auto-added by react

```
<Welcome name="Ali Faleh">  
  <h2>Welcome to QU</h2>  
    
</Welcome>
```

```
function Welcome({name, children}) {  
  return (  
    <>  
      <h1>Welcome {name}</h1>  
      {children}  
    </>  
  );  
}
```

Rendering a List of items (with .map())

Lists are handled using **.map** array function

```
function FriendsList({friends}) {  
  return <ul>  
    {friends.map( (friend, i) =>  
      <li key={i}>{friend}</li>  
    )}  
  </ul>  
}
```

- Fatima
- Mouza
- Sarah

```
<FriendsList>  
  <ul>  
    <li key="0">Fatima</li>  
    <li key="1">Mouza</li>  
    <li key="2">Sarah</li>  
  </ul>  
</FriendsList>
```

Key helps identify which items have changed, added or removed

- Use the **FriendsList** component

```
<FriendsList friends={['Fatima', 'Mouza', 'Sarah']}/>
```

State

$$f(\text{State } \begin{matrix} \text{name: John} \\ \text{surname: Dough} \end{matrix}) =$$

View

John

Dough

OK

Component State

- A component can store its own local data (**state**)
 - Private and fully controlled by the component
 - Can be passed as **props** to children
- Use **useState** hook to create a *state variable* and an *associated function* to update the state

```
const [count, setCount] = useState(0);
```

useState returns a state variable *count* initialized with 0 and a function *setCount* to be used to update it

- Calling *setCount* causes React to **re-render the app components** and **update the DOM** to reflect the state changes



Never change the state directly by assigning a value to the state variable => otherwise React will NOT re-render the UI

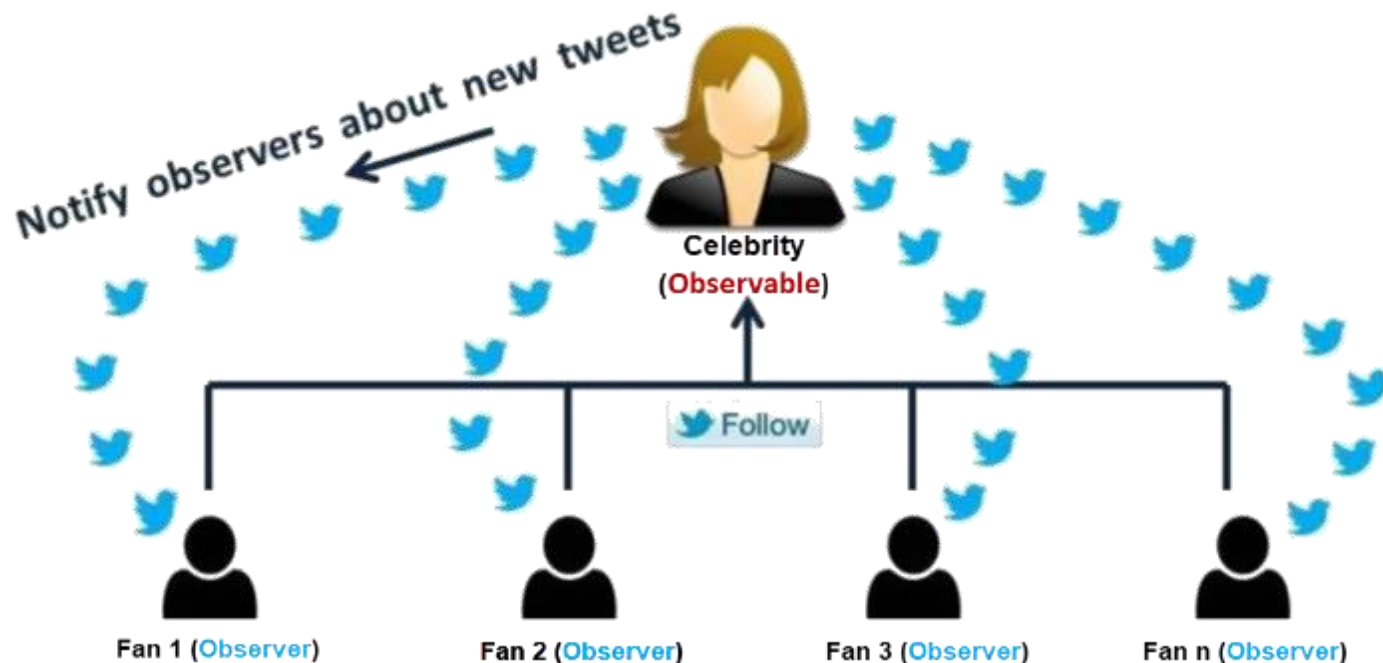
State

- State = any value that can change overtime
 - State variable must be declared using **useState** hook to act as **Change Notifiers**
 - 👍 • They **are observed** by the React runtime
 - Any change of a state variable will trigger the **re-rendering** of any functions that **reads** the state variable
 - Both props and state changes trigger a render update
- => UI is **auto-updated** to reflect the updated app state

Observer Pattern at the heart of Jetpack Compose

Observer Pattern Real-Life Example: A celebrity who has many fans on Tweeter

- Fans want to get all the latest updates (posts and photos)
- Here fans are **Observers** and celebrity is an **Observable** (analogous **state variable** in React)
- A **State variable** is an **observable data holder**: React runtime **observes its changes** and updates the UI accordingly



Imperative UI vs. Declarative UI

- Imperative UI – manipulate DOM to change its internal state / UI

```
document.querySelector('#bulbImage').src = 'images/bulb-on.png';  
document.querySelector('#switchBtn').value = "Turn off";
```



UI in React is immutable

- In react you should NOT access/update UI elements directly (as done in the imperative approach)
- Instead update the UI is by updating the state variable(s) used by the UI elements – this triggers automatic UI update
 - E.g., change the bulb image by updating the ***isBulbOn*** state variable

```
<input type="button"  
  value= {isBulbOn ? "Turn off" : "Turn on"}  
  onClick={() => setIsBulbOn(!isBulbOn)} />
```

useState Hook

State Variable

Setter Function

Initial Value



```
// State with Hooks  
const [count, setCount] = useState(0);
```

The diagram shows three white arrows pointing from the labels above to the code below. The first arrow points from 'State Variable' to the `count` variable in the array. The second arrow points from 'Setter Function' to the `setCount` function in the array. The third arrow points from 'Initial Value' to the `0` argument of the `useState` function.

Component with State + Events Handling

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

Count: 4



```
function Counter(props) {  
  const [count, setCount] = useState(props.startValue);  
  const increment = () => { setCount(count + 1); };  
  
  const decrement = () => { setCount(count - 1); };  
  
  return <div>  
    Count: {count}  
    <button type="button" onClick={increment}>+</button>  
    <button type="button" onClick={decrement}>-</button>  
  </div>  
}  
export default Counter;
```



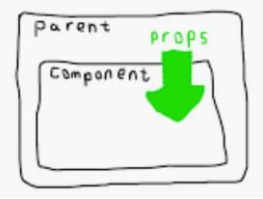
Handling events is done the way events are handled on DOM elements

- Use the **Counter** component

```
<Counter startValue={3}/>
```

Uni-directional Data Flow:

Props vs. State

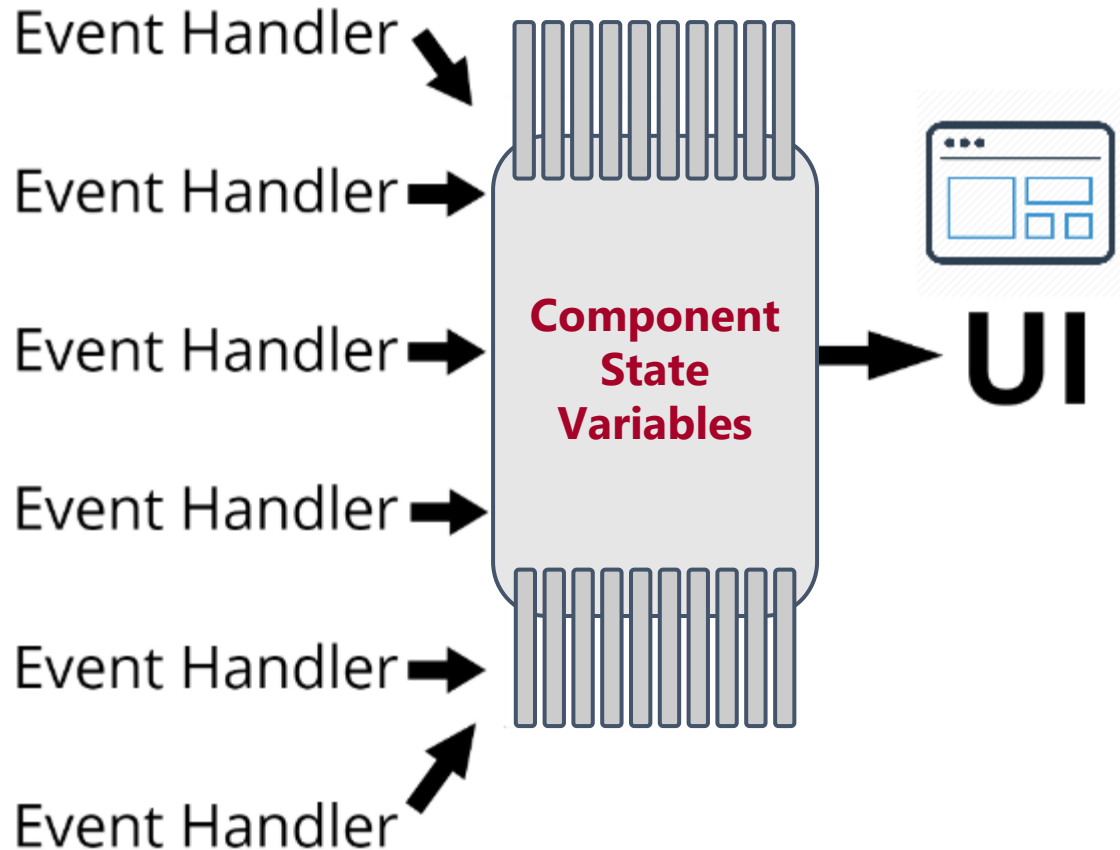


- **Props** = data passed to the child component from the parent component
- **Props** parameters are **read only**

- **State** = internal data managed by the component (cannot be accessed and modified outside of the component)
- **State** variables are **Private** and **Modifiable** inside the component only (through **set** functions returned by `useState`)

👍 React **automatically re-render the UI** whenever **state** or **props** are updated

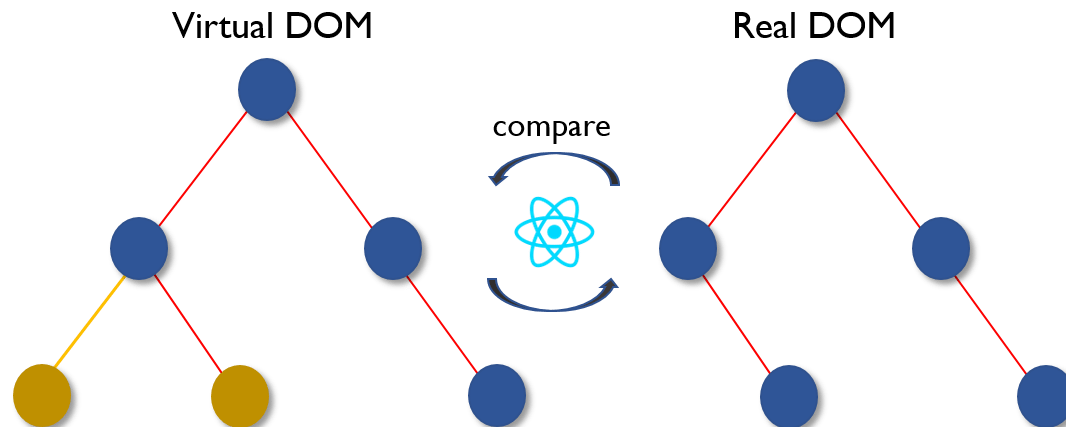
Event Handlers update the State and React updates the UI



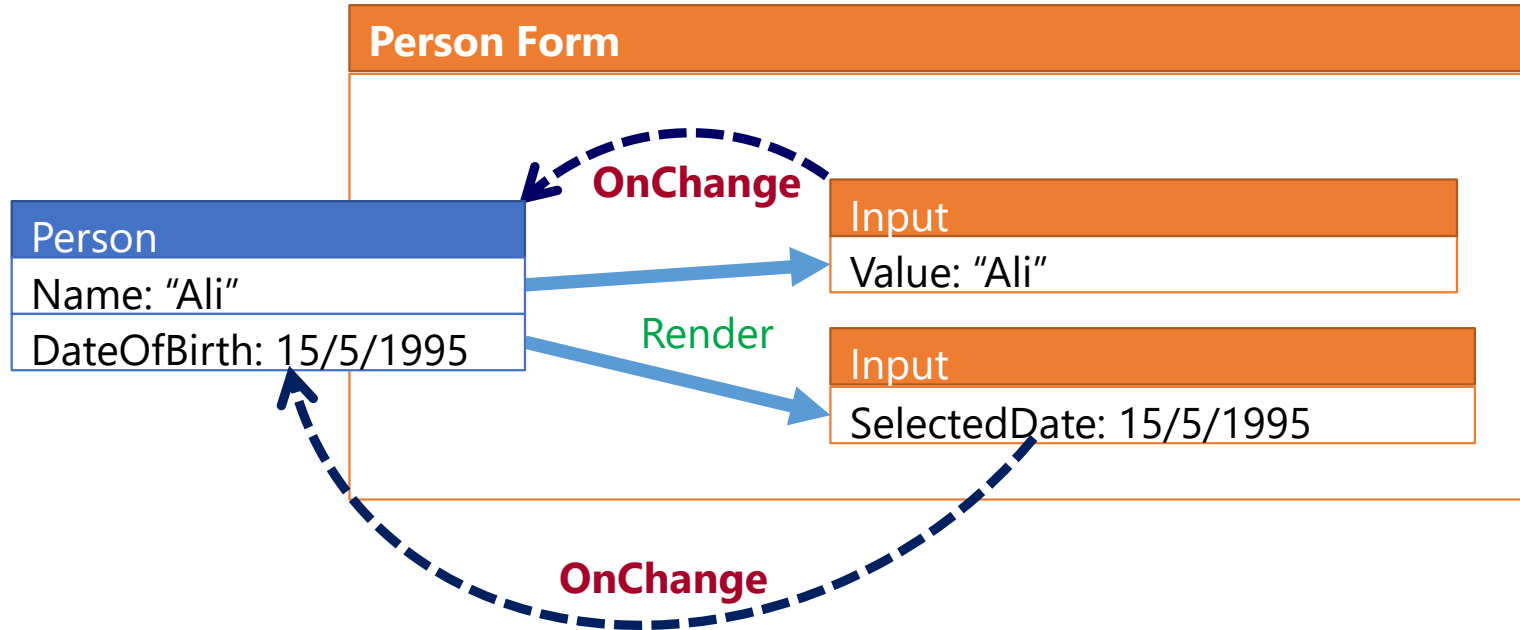
Every place a state variable is displayed is guaranteed to be auto-updated

Virtual DOM

- Virtual DOM = Pure JavaScript lightweight DOM, totally separate from the browser's slow JavaScript/C++ DOM API
- Every time the component **updates its state** or **receives new data via props**
 - A new virtual DOM tree is generated
 - New tree is **diffed** against old...
 - ...producing a minimum set of changes to be performed on real DOM to bring it up to date



Unidirectional Data Flow in Forms



Common Events: `onClick` - `onSubmit` - `onChange`

Forms with React

Form UI

```
<form onSubmit={handleSubmit}>
  <input
    name="email"
    type="email" required
    value={state.user}
    onChange={handleChange} />
  <input
    name="password"
    type="password" required
    value={state.password}
    onChange={handleChange} />
  <input type="submit" />
</form>
```

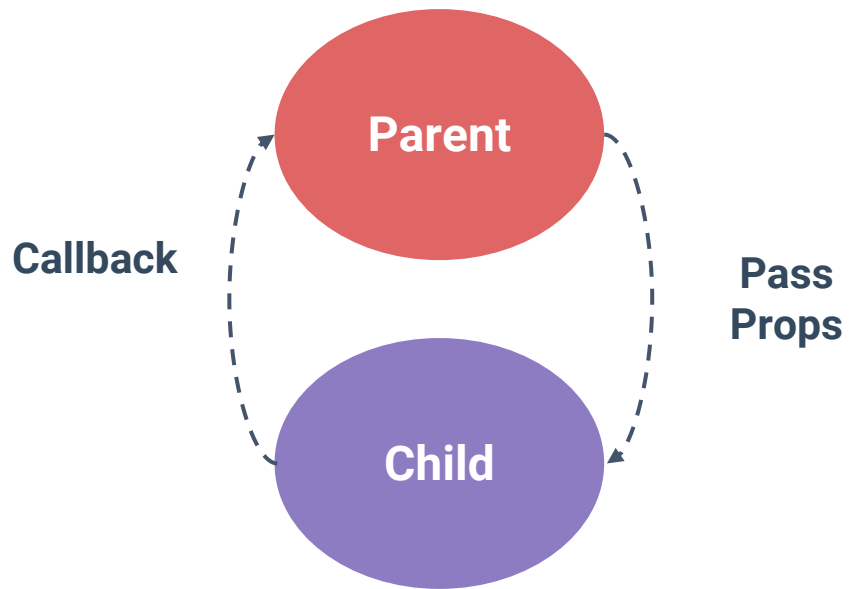
```
const [state, setState] = useState({ email: "", password: "" });
```

```
const handleChange = e => {
  const name = e.target.name;
  const value = e.target.value;
  //Merge the object before change with the updated property
  setState({ ...state, [name]: value });
};
```

```
const handleSubmit = e => {
  e.preventDefault();
  alert(JSON.stringify(state));
};
```

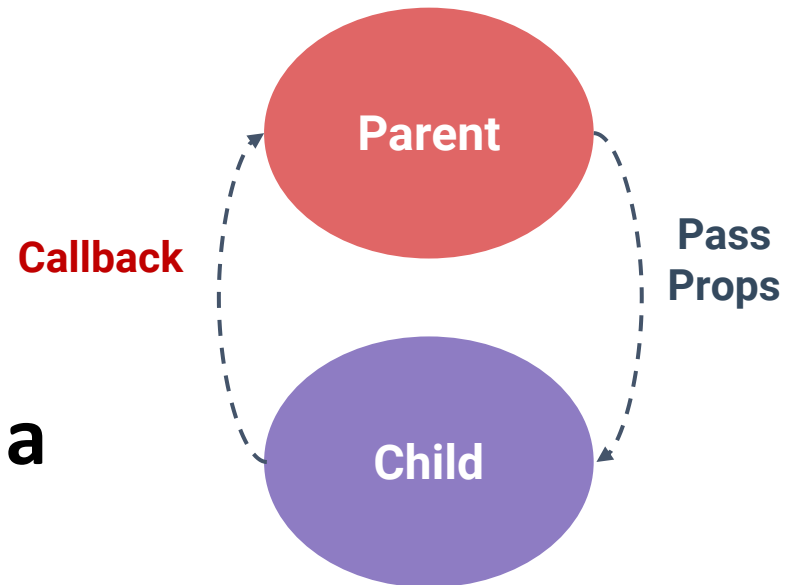
Form State and Event Handlers

Components Communication

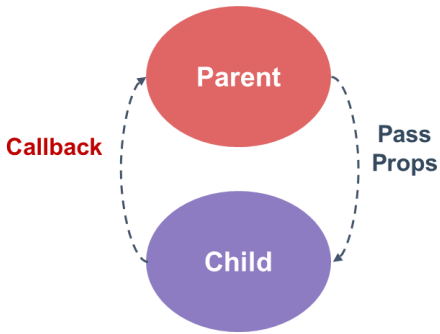


Composing Components

- Components are meant to be used together, most commonly in parent-child relationships
- Parent passes data down to the child via **props**
- The child notify its parent of a **state change via callbacks** (a parent must pass the child a callback as a parameter)



Parent-Child Communication



Parent

```
<Counter startValue={3}  
  onChange= {count => console.log(`Count from the child component: ${count}`)} />
```

Child

```
function Counter(props) {  
  const [count, setCount] = useState(props.startValue);  
  
  const increment = () => {  
    const updatedCount = count + 1;  
    setCount(updatedCount);  
    props.onChange(updatedCount);  
  };  
  
  return <div>  
    Count: {count}  
    <button type="button" onClick= {increment}>+</button>  
  </div>  
}
```

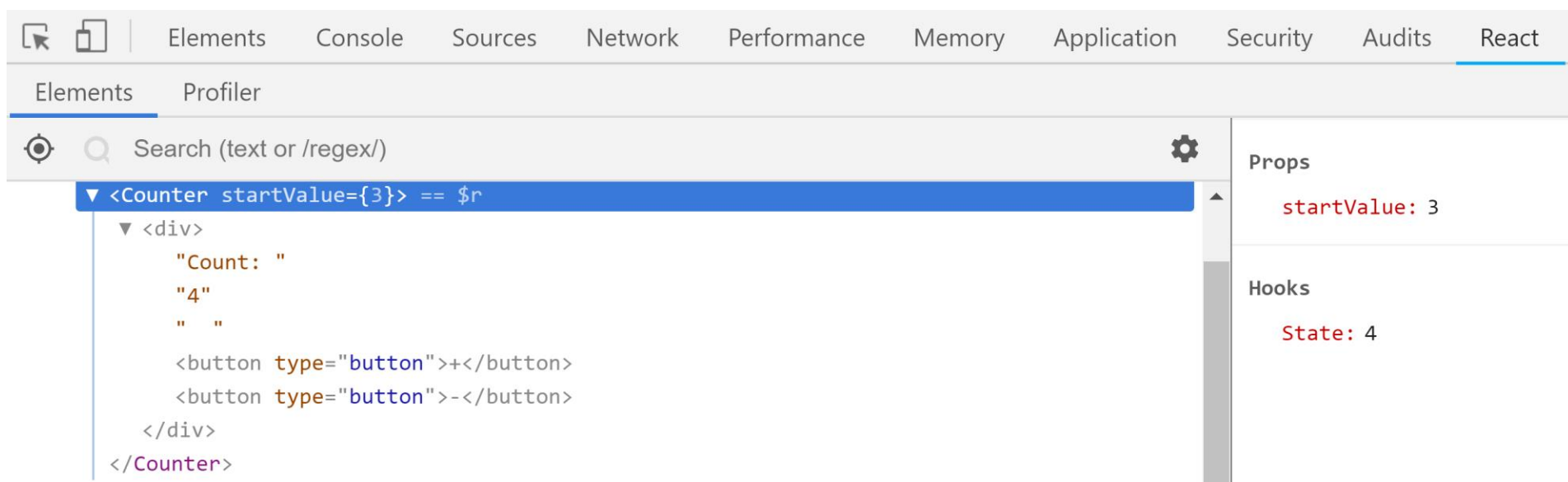
React Tools and Component Libraries

- **React Dev Tools**
- **React Components Libraries**

React Dev Tools

- React Dev Tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>



React Component Libraries

- **Material-UI:** React components with Material Design
<https://mui.com/>
- **Grommet Components**
<https://v2.grommet.io/components>
- **Blueprint:** React-based UI toolkit
<https://blueprintjs.com/>
- **Fluent UI React Components**
<https://react.fluentui.dev/>

Summary

- React = a declarative way to define the UI
- Decompose UI into self-contained and often reusable components
- Why React:
 - Component-based
 - Virtual DOM
 - Declarative
- React uses JSX syntax to define component's UI

Resources

- Thinking in React

<https://reactjs.org/docs/thinking-in-react.html>

- React Router

<https://reactrouter.com/>

- Useful list of resources

<https://github.com/enaqx/awesome-react>