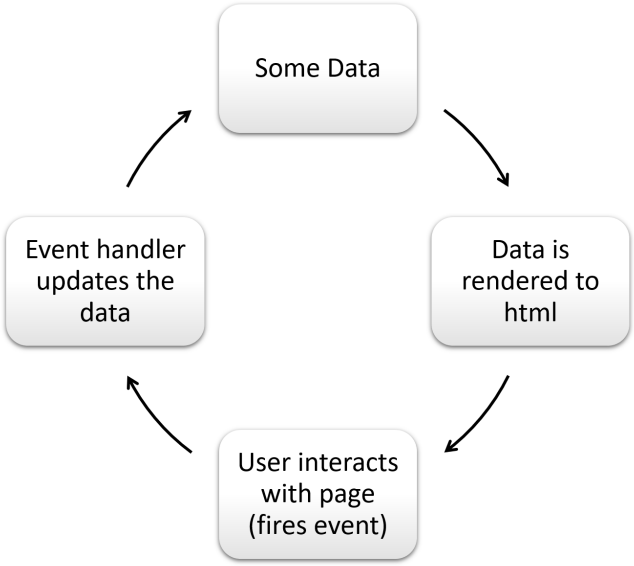
React Academy

React Cheatsheet

Components in React

- This is how components works in React
- No two-way binding. Only one-way binding



Describe the result

- React tries to be declarative instead of imperative
  - Don't try to modify the display (imperative) (like in jQuery)
  - Instead, just describe the result
    - React will figure out the changes that need to happen in DOM

```
<!-- just describe the following -->
<h3 title='alarm'>The alarm goes off at 10:04 AM</h3>
```

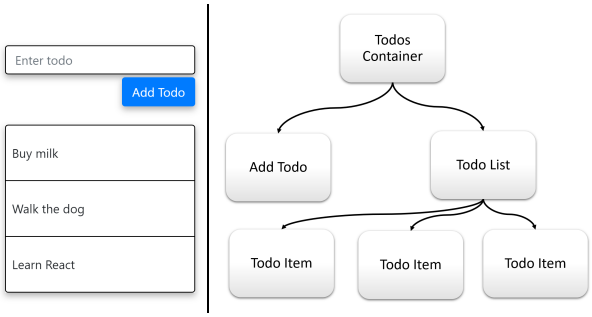
Create components

- Components are functions (or classes)
  - Always start with an uppercase letter
  - Take a single argument called props
- Components return (or render) a 'React element' (also called 'virtual element')

```
//Simple Component.
//Starts with uppercase
//takes a single argument
function SimpleComponent(props) {
  return null;
}
```

Think in React

- Decompose the UI into components
  - Each component will render itself, based on external and internal data
  - Build trees of the components to create app



JSX to the rescue

- React Elements should be created by JSX
- JSX is an easy syntax to create React elements
  - looks like HTML (gets transpiled to `createElement()`)
  - Inside of JSX, curly braces switches to JS

```
import React from 'react'; //we still need this
//because JSX transpiles to React.createElement();

// This is the code to display the time ⬇⬇.
function DisplayAlarm(props) {
  return (  // JSX is often enclosed in round brackets ()
    <h3 title='alarm'>
      The alarm goes off at {props.alarmTime}
    </h3>  // curly braces {} switches to JS ⬆⬆
  );
}
//Babel converts JSX to React.createElement() ⬆⬆
```

React elements can be created by createElement()

- Writing raw React elements is hard
- Easier with createElement(), but still too hard

```
//Need this ⇓⇓ to call React.createElement();
import React from 'react';

//No one writes React like this ⇓⇓.
//Instead, use JSX, it gets transpiled
//to the following JS.
function DisplayAlarm(props) {
  return React.createElement('h3', {title: 'alarm'},
            'The alarm goes off at ', props.alarmTime )
}
```

How React works behind the curtain

- React has a 'reconciliation' phase
  - Called after a component (and all its children) are 'rendered'
    - 'rendered to the virtual Dom' (or vDom): elements (js objects)
    - React compares current elements with the previous elements of the vDom
    - Any modification generates the smallest DOM change possible

```
//No one writes React like this.
//This is what the generated element looks like to React
function DisplayAlarm(props) {
  return {
    type: 'h3',
    props: {
      title: 'alarm',
      children: ['The alarm goes off at ',
                 props.alarmTime],
    },
    key: null, ref: null,
    $$typeof: Symbol.for('react.element')
  }
}
```

Component create a tree

- You can use components from other components.
- You build a "component tree" to describe your app

```
function DoubleAlarmClock(props) {
  //code could be found before the return
  return (
    <div>  {/* call other components */}
      <DisplayAlarm  />
      <div>  {/* multiple times */}
        <DisplayAlarm   />
      </div>
    </div>
  );
}
//all tags in jsx must be closed, or self-closed ⇓⇓
```

Use props to push data down

- Data coming from a parent component is called "props" (properties)
- Props are pushed from parent to children through JSX attributes
- Props are passed as the argument to the function.

```
function DoubleAlarmClock(props) {
  //otherAlarm is 10 minutes later (60000ms) ⇓⇓
  const otherAlarm = props.alarm + 60000;

  return (
    <div>
      <DisplayAlarm alarmTime={props.alarm}  />
      <div>
        <DisplayAlarm alarmTime={otherAlarm}  />
      </div>
    </div>
  );
}
```

Store data in State

- Modifiable data is called "state".
- State is defined by the 'useState' function.
  - It returns an array composed of two items
    - First item is the data (state)
    - Second is a function to change that data (setState).
- NEVER change the state directly.
  - Always use the set function
  - This will schedule a rerender of the component (and child components)

```
import React, {useState} from 'react';
    //named export ⇓⇓

function Counter(props) {
  const [count, setCount] = useState(5);
    //set to 5 the first time it's called ⇓⇓
    //use count to read the data
    //use setCount to write the data

  //a function inside a function
  //called by the click event ⇓⇓
  function increment(evt) {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Hello {props.name}, the count is: {count} </h1>
      <button onClick={increment} >Add 1</button>
    </div>
  );
 }
// events are wired this way in React: onClick={fn} ⇓⇓
// It is a synthetic event, different than
// a real DOM event: onclick="increment".
// (Notice the difference in casing)
// Use synthetic events whenever possible.
```

Change the state to rerender

- When state (or props) change, or the parent re-renders, your component updates AUTOMATICALLY.
- You never have to modify the DOM yourself
  - If you do, you are not using React properly

```
function Counter(props) {
  const [count, setCount] = React.useState(5);

  function increment(evt) {
    setCount(count + 1);
  }
  //setCount ⬆⬆ doesn't change the state right away
  //it schedules the change for right after the firing
  //of all events. Once all scheduled changes are done
  //changing the states, those component will now
  //"rerender" (along with the child components).

  return (
    <div>
      <h1>Hello {props.name}, the count is: {count} </h1>
      <button onClick={increment} >Add 1</button>
    </div>
  );
}
```

Looping over items

- Loop over an array of data with `.map()`
- It returns an element for each iteration
- For performance reasons, remember to provide an unique key for each element

```
const authors=[
  {id: 1, name: 'Jeff'},
  {id: 2, name: 'Bill'},
  {id: 3, name: 'Mary'},
];

function App() {
  return (<AuthorList authors={authors} />);
} ;

function AuthorList(props) {
  return (
    <ul>
      {
        props.authors.map( (author) => {
          return <li key={author.id}>{author.name}</li>;
        }) //keys ⬆⬆ are used by React for performance
           //in reconciliation phase, not part of the DOM
      }
    </ul>
  );
}
```

Different ways to style

- use CSS files and then set attribute with `className='danger'`
  - use 'className' ('class' is a reserved keyword in js)
- Set inline styles with `dangerStyle={{backgroundColor: 'red' }}`
  - you could refer to a global style variable
  - `dangerStyle={backgroundColor: red}`

```
<p className='danger'>Text</p>
<p style={dangerStyle}>Text</p>
<p style={{backgroundColor: 'red',
           width: '300px', height: 300}}>
<!-- {/*  numeric values will be converted to 'px' units
*/} -->
```

Use a third-party library for theming and styling

Here is a list of libraries to help you generate css from js.

- Aphrodite
- Emotion
- Glamor
- Fela
- Styletron
- Jss
- Radium
- React-Native-Web
- Styled-Component
- CSS-in-JS (Facebook 2020?)

Use effect

- First argument of `useEffect()` takes a function.
  - Runs right after reconciliation and painting the screen.
  - Inside of this callback, you perform async functions and side effects
    - Manipulating the DOM (read or write) is a side effect
    - Network calls are async functions
  -
- The second argument is a "change array" of dependencies
  - Effect gets called only when an item in the array is modified
    - Code should include in array any var that might change.
  - Put null or nothing to execute after all renders
  - Put an empty array [] for the effect to be called just once, right after the first render (when React "mounts" the component)

Code sample of useEffect

```
import React, {useState, useEffect} from 'react';
const url = 'https://randomuser.me/api/'

function DisplayUser(props){
  const [data, setData] = useState();

  useEffect(() => {
      fetch( `${url}?seed=${props.id}` )
        .then( (response) => response.json() )
        .then( (data) => setData(data.results[0]) );
  }, [props.id]);
  // useEffect ⬆⬆ is called after each render
  // where id was modified (by the parent component)

  //we return the render result ⬇⬇
  return (
    <div>
      {data ?  // if no data, show a "no data" msg
        <p>
          <img src={data.picture.thumbnail} />
          {data.name.first}
        </p>
      :
        <p>No data (yet)!</p>
      }
    </div>
    );
}
```

Returned effect: Cleanup

- The returned function is cleanup. it is called either:
  - Just before the next effect is called, but after React rendered
  - Just after the component is destroyed (when react "unmounts" the component)
- Makes it easy to subscribe and unsubscribe to stuff

```
useEffect( () => {
    chatSubscribe(friendId);
    return(function cleanup() {
      chatUnsubscribe(friendId);
    });
  }
, [friendId] );
//Each time the chat is re-rendered, if friendId was
//modified, then we unsubscribe to the previous chat,
//just before subscribing to the new friendId. The magic
//of "closures" will ensure the old friendID gets
//unsubscribed, not the current one.
```