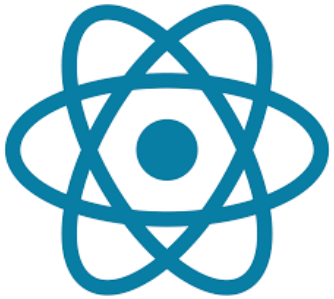
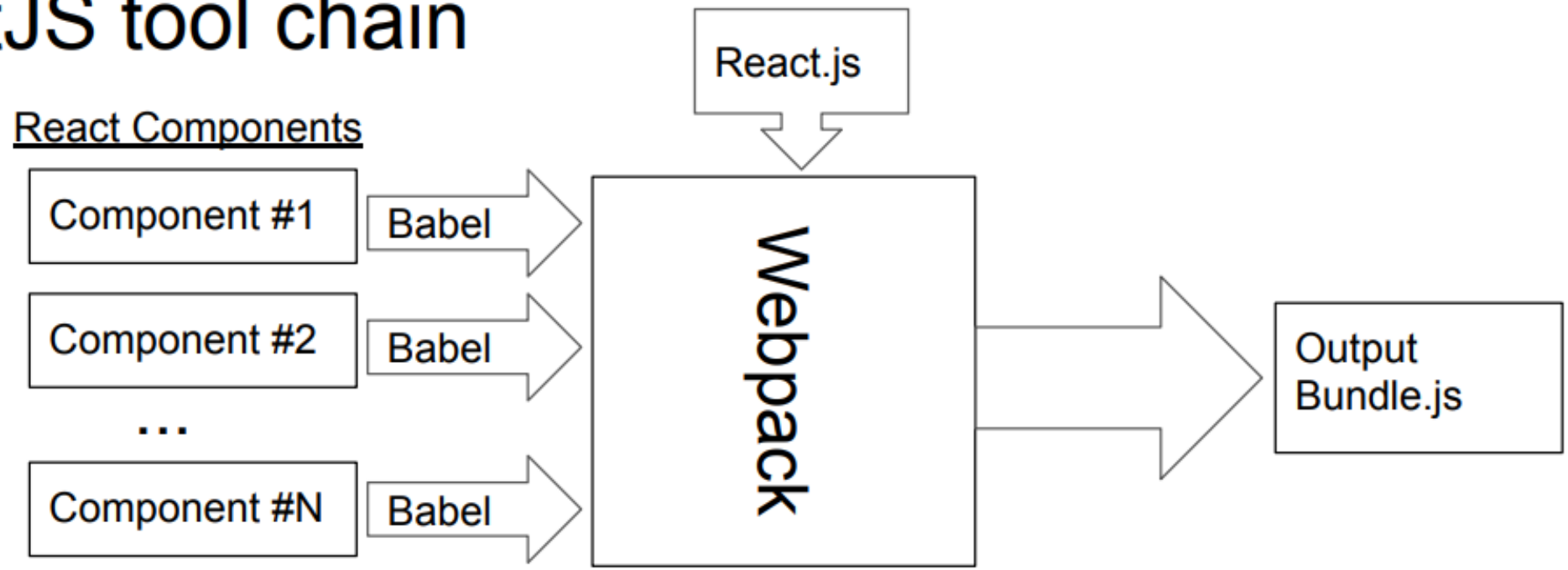
React Introduction



ReactJS

- JavaScript framework for writing the web applications
  - Like AngularJS - Snappy response from running in browser
  - Less **opinionated**: only specifies rendering view and handling user interactions
- Uses Model-View-Controller pattern
  - View constructed from Components using pattern
  - Optional, but commonly used HTML templating
- Minimal server-side support dictated
- Focus on supporting for programming in the large and single page applications
  - Modules, reusable components, testing, etc.

ReactJS tool chain



Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

Webpack - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

```
<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>

<script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

Components

```
class Example extends React.Component{

                render(){
                    return (<p>This is my first msg!</p>);
                }


            }
```

reactApp.js - Render element into browser DOM

```
import React from 'react';
import ReactDOM from 'react-dom';

import ReactAppView from './components/ReactAppView';

let viewTree = React.createElement(ReactAppView, null);
let where = document.getElementById('reactapp');

ReactDOM.render(viewTree, where);
```

ES6 Modules - Bring in React and web app React components.

Renders the tree of React elements (single component named **ReactAppView**) into the browser's DOM at the div with id=reactapp.

Components

```
 <p id="#p1"></p>
        <script type="text/babel">

class Example extends React.Component{
                render(){
                    return (<p>This is my first msg!</p>);
                }
            }

ReactDOM.render(<Example/>, document.getElementById('#p1'));

        </script>
```

Adding Style and Events:

```
<div style={{border: "1px solid black",width: 100, height:
100}}>This is for box</div>


<button onClick={this.handleClick}>Play the Game</button>


handleClick= () => {
                    //code for Handling the Event
                }
```

State

```
class Example extends React.Component{
                    constructor(props){
                        super(props);
                        this.state={
                            //list of properties of the component
                            (key: value)

                        };
                    }
```

Use JSX to generate calls to createElement

app.js

```
render() {
    return (
        <div>
          <label>Name: </label>
          <input
                type="text"
                value={this.state.yourName}
                onChange={(event) => this.handleChange(event)}
            />
          <h1>Hello {this.state.yourName}!</h1>
       </div>
    );
}
```

- JSX makes building tree look like templated HTML embedded in JavaScript.

Component state and input handling

Greet.js

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  ....
```

Make `<h1>Hello {this.state.yourName}!</h1>` work

- Input calls to `setState` which causes React to call render() again

One way binding: Type 'D' Character in input box

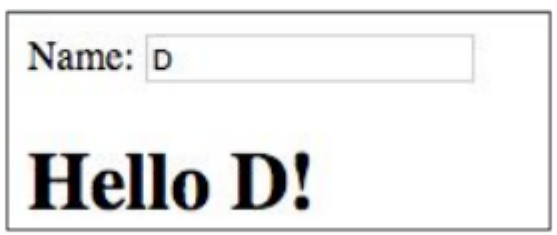
- JSX statement: `<input type="text" value={this.state.yourName}`
  `onChange={(event) => this.handleChange(event)} />`

  Triggers `handleChange` call with `event.target.value` == "D"

- handleChange - `this.setState({yourName: event.target.value});`

  `this.state.yourName` is changed to "D"

- React sees state change and calls render again:
- Feature of React - highly efficient re-rendering



Programming with JSX

- Need to remember: JSX maps to calls to `React.createElement`
  - Writing in JavaScript HTML-like syntax that is converted to JavaScript function calls
- `React.createElement(type, props, ...children);`
  - `type`: HTML tag (e.g. `h1, p`) or `React.Component`
  - `props`: attributes (e.g. `type="text"`) Uses camelCase!
  - `children`: Zero or more children which can be either:
    - String or numbers
    - A React element
    - An Array of the above

JSX templates must return a valid `children` param

- Templates can have JavaScript scope variables and expressions
  - `<div>{foo}</div>`
    - Valid if `foo` is in scope (i.e. if foo would have been a valid function call parameter)
  - `<div>{foo + 'S' + computeEndingString()}</div>`
    - Valid if `foo & computeEndString` in scope
- Template must evaluate to a value
  - `<div>{if (useSpanish) { ... } }</div>` - Doesn't work: `if` isn't an expression
  - Same problem with "for loops" and other JavaScript statements that don't return values
- Leads to contorted looking JSX: Example: Anonymous immediate functions
  - `<div>{ (function() { if ...; for ..; return val;})() }</div>`

Conditional render in JSX

- Use JavaScript Ternary operator (?:)

```
<div>{this.state.useSpanish ? <b>Hola</b> : "Hello"}</div>
```

- Use JavaScript variables

```
let greeting;
const en = "Hello";  const sp = <b>Hola</b>;
let {useSpanish} = this.prop;
if (useSpanish) {greeting = sp} else {greeting = en};
```

```
<div>{greeting}</div>
```

Iteration in JSX

- Use JavaScript array variables

```
let listItems = [];
for (let i = 0; i < data.length; i++) {
   listItems.push(<li key={data[i]}>Data Value {data[i]}</li>);
}
return <ul>{listItems}</ul>;
```

- Functional programming

```
<ul>{data.map((d) => <li key={d}>Data Value {d}</li>)}</ul>
```

`key=` attribute improves efficiency of rendering on data change

Stateless Components

- React Component can be function (not a class) if it only depends on props

```
function MyComponent(props) {
  return <div>My name is {props.name}</div>;
}
```

Or using destructuring...

```
function MyComponent({name}) {
  return <div>My name is {name}</div>;
}
```

- Much more concise than a class with `render` method
  - But what if you have one bit of state…

React Hooks - Add state to stateless components

- Inside of a "stateless" component add state: `useState(initialStateValue)`
  - `useState` parameter: `initialStateValue` - the initial value of the state
  - `useState` return value: An two element polymorphic array
    - 0th element - The current value of the state
    - 1st element - A set function to call (like `this.setState`)
- Example: a bit of state:

```
const [bit, setBit] = useState(0);
```

React Hooks Example - useState

```
import React, { useState} from 'react';
function Example() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
           Click me
      </button>
    </div>
  );
}
```

Communicating between React components

- Passing information from parent to child: Use props (attributes)

```
<ChildComponent param={infoForChildComponent} />
```

- Passing information from child to parent: Callbacks

```
this.parentCallback = (infoFromChild) =>
        { /* processInfoFromChild */};

<ChildComponent callback={this.parentCallback}> />
```

- React Context (https://reactjs.org/docs/context.html)
  - Global variables for subtree of components