# Web Development: Frontend in React

EE 461L Software Engineering & Design
Fall 2022

Evan King
e.king@utexas.edu

# Agenda

- Summary of last class
- Web development roadmap: where we are
- **Context:** Frontend frameworks
- **Deep dive:** React
  - Programming model
    - Components
      - Lifecycle
      - Props & State
    - Events
  - Tools

# Fundamentals

**HTML** – The structure of the page. Elements map to the Document Object Model (DOM), a tree structure that represents the hierarchical structure of the document.


**CSS** – The page design. It's art and a science. Tags, classes and IDs are used to select and style elements. The box model helps us define the flow of elements.


**JavaScript** – The behavior of the page. It enables computation in the browser; event handler functions define what the page does in response to different events.

# Refresher: JavaScript Event Handling

JavaScript *event handling* allows you to implement functions invoked in response to different events on the page.

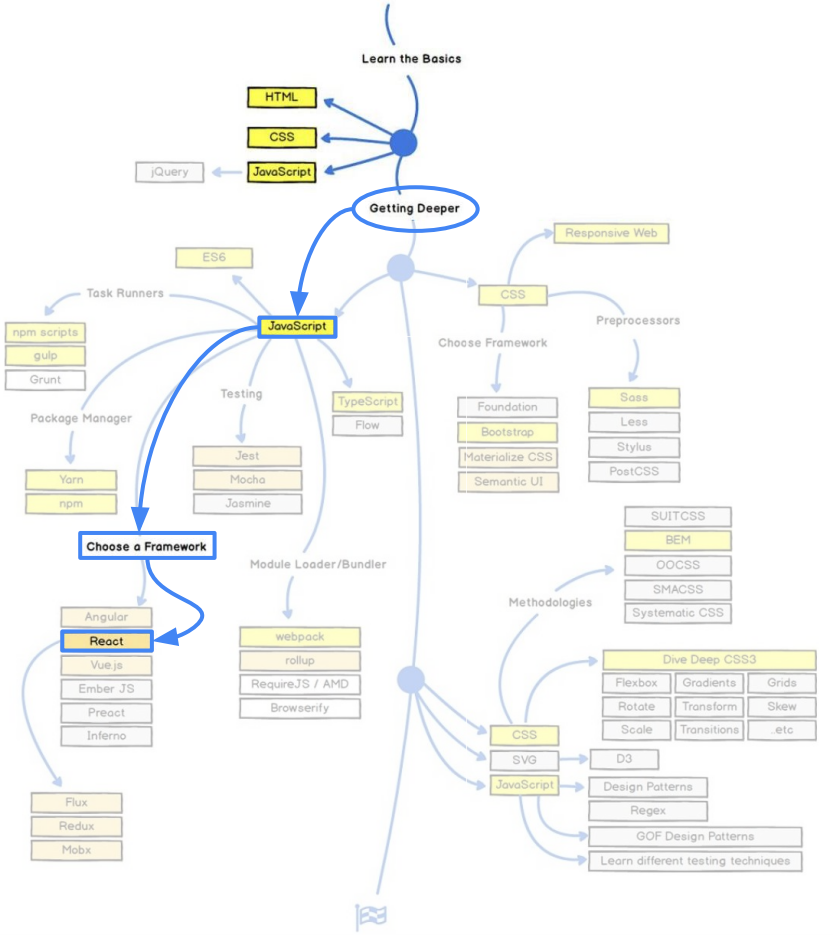
```
<script>
    function sayHello() {
        alert('Hello!')
    }
</script>

<button onclick="sayHello()">Receive greeting.</button>
```

# Web Dev Roadmap

HTML/CSS/JS are fundamentals – you should have a firm grasp on them.

- Knowing what HTML elements build a layout
- Knowing how to use CSS to style them
- Understanding how JS works with the DOM

Now we move on to frontend development using JS frameworks.



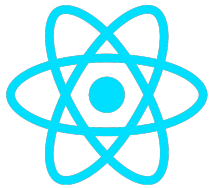
# Frontend Frameworks

JavaScript frameworks make frontend development easier by providing basic functionality out of the box. Each has its own programming model.




| Pros | Cons | Pros | Cons | Pros | Cons |
|---|---|---|---|---|---|
| Easy to get started | JSX docs are lacking | Intuitive MVC architecture | Steep learning curve | Easy to get started | Less enterprise support |
| Component structure is intuitive | UI development only | Robust state management | Verbose | Lots of flexibility | Flexibility leads to messy code |
| SEO-friendly | Unopinionated (lots of ways to design an app) | Lots of tooling for cross-platform development | Not SEO-friendly | Component structure is intuitive | Weaker cross-platform support |

# React



React's core principle: pages should *react* to changes in state by updating component views.

- **Modular:** apps are built with components.
- **Stateful:** components have state. When state changes, views are updated (one-way data binding)

JSX

- A syntax extension to JS used by React
- JS expressions can be embedded in HTML

```
<h1> Hello, {formatName(user)} </h1>
```

# Today’s example: a user login flow

Components → Props & State → Events

# Components

Modularize the UI by splitting it into independent and reusable *components*.

**As a function:**

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

**As a class:**

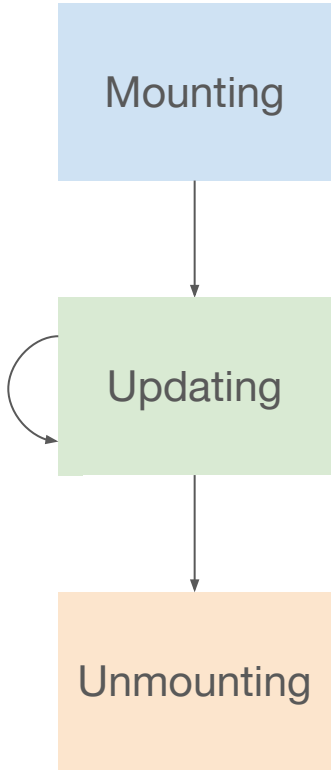
```
class Welcome extends React.Component {
    render() {
        return ( <h1>Hello</h1> )
    }
}
```

this is the style I'll be using in class today

# Lifecycle

Components have a three-phase *lifecycle*. During each phase, different default functions are invoked. Overriding these functions defines behavior at each phase.

- **Mounting** – component added to DOM
  - `constructor()` and `render()`
- **Updating** – component living in DOM
  - `render()`
- **Unmounting** – component removed from DOM
  - `componentWillUnmount()`

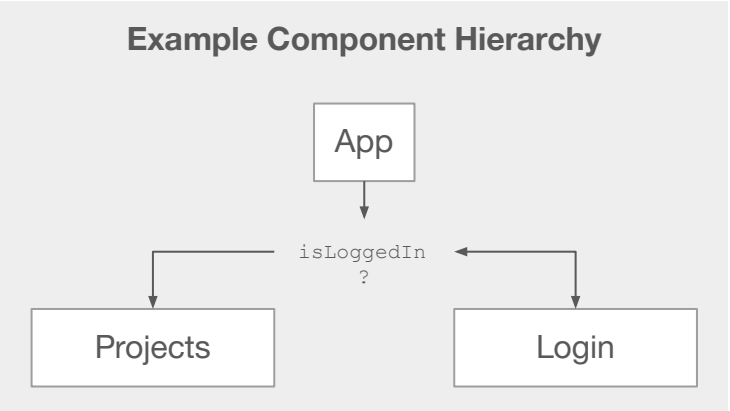



Component Lifecycle Reference

# Hierarchy

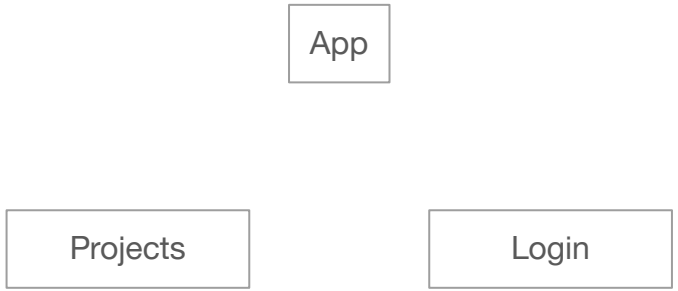
Approach React UI design by thinking about the *component hierarchy.*

- A few top-level components compose many smaller, low-level components.
- “Single responsibility” – components should be responsible for one thing.




*an App with a Projects component which is displayed when the Login component says the user is logged in.*

# Start with components…



# Props

Props or “properties” are *immutable* values that are internal to components.

- Props are passed to child components by parent components

```
class App extends React.Component {
 render() {
   return ( <Welcome name='Evan' /> )
 }
}
```

*parent component*

```
class Welcome extends React.Component {
 render () {
   return ( <h1>Hello, {this.props.name}</h1> )
 }
}
```

*child component*

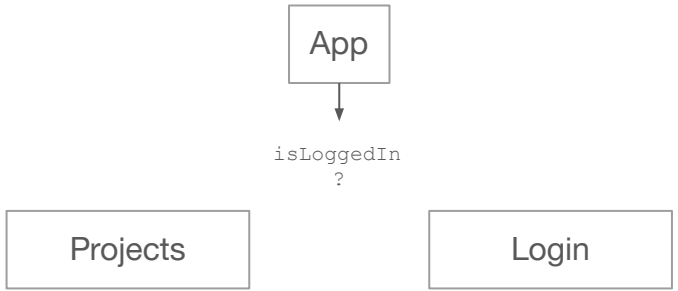
# State

Components have state which is *mutable.* When state changes, views of the state are updated automatically.

```
class Clock extends React.Component {
 constructor(props) {
   super(props)
   this.state = { date: new Date()}; }

 render() {
   return (
     <div>
       <h1>Hello, world!</h1>
       <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
     </div>
   );
 }
}
```

Use `this.state = { key: value }` to initialize state
and `this.setState({ key: value })` to change it.

# Add some state…

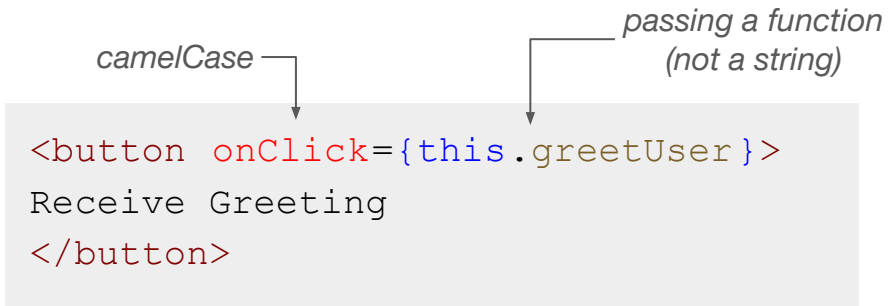


# Events

1. Define a function inside the component

```
greetUser() {
   console.log('Hello user')
}
```

2. Bind it to an element's event handler (like plain JS, but using JSX syntax)



# Note: Event Handler Syntax(es)

There are several different ways to set event handlers.

### As a class method:

```
constructor() {
 this.handleClick = this.handleClick.bind(this)
}

handleClick() { … }

render() {
  return (
    <button onClick={this.handleClick}>OK</button>
  )
}
```

### As a “public class field”:

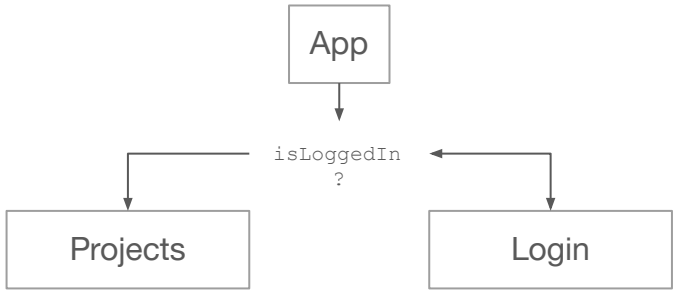
```
handleClick = () => { … }

render() {
  return (
    <button onClick={this.handleClick}>OK</button>
  )
}
```

### As a class method bound with an arrow function:

```
handleClick() { … }

render() {
  return (
    <button onClick={() => this.handleClick()}>OK</button>
  )
}
```

JavaScript is a multi-paradigm language 😉

# Add some event handling…



# Getting Started With a Toolchain

A toolchain is a set of tools used to initialize projects, download and manage libraries, run a project while developing, and build it for deployment.

- There are a few different React toolchains.
- The one you should use for this class uses Node.js (download it)

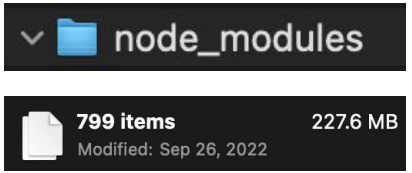
Node.js

- Recall: JavaScript can be run *in the browser* or *standalone*
- Node.js is a *standalone* JavaScript runtime
- Includes a package manager (like `pip`) for downloading and managing libraries (`npm`)
- The React toolchain we use is built on Node.js

# npm and npx

## `npm` – Node.js package manager

React projects rely on *a lot of modules* to work. `npm` is what we use to manage all of those dependencies.




- `npm install packagename` is used to install new packages
- dependencies are managed in a file called `package.json`

## `npx` – Node.js package runner (or eXecutor)

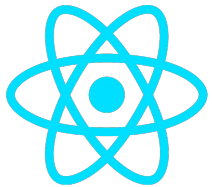
Some node packages are *executable*. `npx` runs them.

- The one we care about is `create-react-app`. It creates all the starter code for a new React app.
- `npx create-react-app yourappname`

# Creating a starter app

Once you've downloaded Node.js, run:

```
npx create-react-app myappname
```

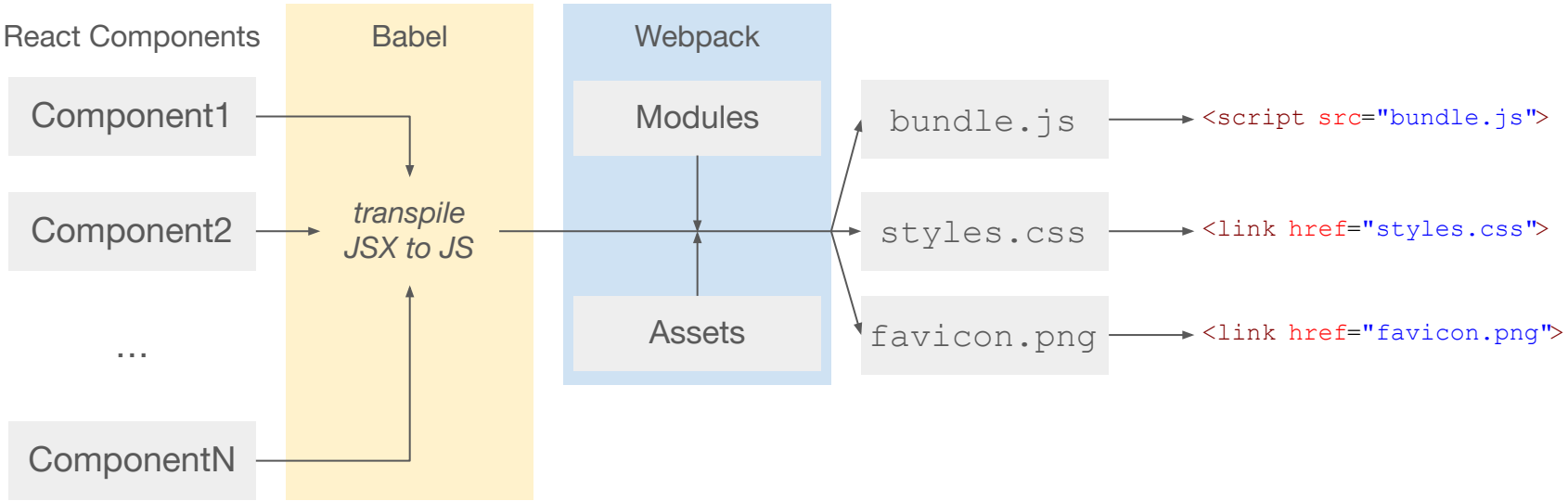


This will create a starter React app in your current directory. Inside that directory, run:

```
npm start
```

This starts a development server, which will "run" your frontend in the browser.

# Anatomy of a Build




**Babel** – transpiles JSX to basic JavaScript

**Webpack** – bundles components with modules (including React itself) into a single JavaScript file, which is loaded in HTML using the `<script>` tag. Other assets like stylesheets and images are packaged and placed in their respective static directories for deployment.

# Code Snippets

Refer to the code snippets on Canvas for self-contained examples of other React concepts.

| | |
|---|---|
| Rendering Elements<br>• Rendering elements in the DOM<br>• Updating elements | `Rendering_Elements.js` |
| Components & Props<br>• Rendering a component<br>• Composing components | `Rendering_Components.js`<br>`Composing_Components.js` |
| Conditional Rendering | `Conditional_Rendering.js` |
| Lists and Forms | `Forms.js` |

# Reference Materials

The official documentation is a great resource on React. A few useful pages:

- Using the toolchain to start a new project
- Components and props
- State and lifecycle
- Event handling
- Lifting state up (sharing state between components)