Introduction to React and Web Bundlers

CIS 1962 (Winter 2026)
February 19th, 2026

Penn Engineering

Lesson Plan

React & Web Bundlers

5	JavaScript Frameworks
9	What is React?
15	Web Bundlers

React Syntax

22	JSX and Component Syntax
33	React Props
43	Conditional/List Rendering
54	Events & Inputs

Weekly Logistics

Homework 4: ChatJS Due Next Week (2/26)

- Make sure you received your API key for this homework (through Canvas), you will need it!

Review Activity

https://edstem.org/us/courses/91614/lessons/159893/slides/939137

Let's review some content from the previous lecture before we start!

JavaScript Frameworks

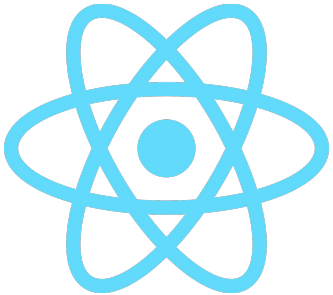
How does a JavaScript framework help developers build modern web applications quickly?

JavaScript Frameworks

JavaScript frameworks help build web applications efficiently by providing structure and reusable code.

They provide benefits like:

- Structure and conventions
- Built-in utilities for common tasks
- Faster development



The Rise of Frameworks

Vanilla JS uses DOM manipulation and works with scripts and ids. This often leads to problems with code maintenance and handling UI states, as things like automatically updating UI requires complex code.

Libraries like jQuery can improve the experience of working with DOM, but this still leads to complex code bases. This gets more complex as **Single-Page Applications (SPAs)** got more popular.

Using Frameworks

Each JavaScript framework is structured in a different way around JavaScript's features.

They each will provide certain **conventions** and **tools** to organize and enhance JavaScript, while allowing the use of JavaScript logic, including all of its syntax.

What is React?

React: Your View Layer

React is a declarative, component-based JavaScript library for building user interfaces.

It is not an all-in-one framework, as it only focuses on the view layer of the app, not the models or backend components.

It does contain a rich ecosystem of tailor-made libraries for it, to make up for things it lacks (routers, styling, forms, global state management)

How React Works

React has:

- **Component-Based UI**: UI is made up of reusable pieces
- **Declarative UI**: You describe what the UI looks like while React handles the updates
- **Virtual DOM**: React creates a virtual copy of the DOM, and only needs to update what is changed when changes occur.
- **Unidirectional Data Flow**: Data flows from parent to child within the DOM

Example: React Component

React components are written as JS functions that return JSX, or JavaScript XML, that is HTML-like code for React elements.

```
const Navbar = () => {
     const navLinks = [
          ...
     ];

     return (
          <nav>
               {navLinks.map((link) => (
                    <li key={link.href}>
                         ...
                    </li>
               ))}
          </nav>
     )
}

export default Navbar;
```

Example: Rendering Components

Once you have your component, you can insert it into other components. The starting point for rendering is made with the React API function `createRoot()`, which creates a top-level node that is now managed by React for rendering.

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import Navbar from '../Components/Navbar'

createRoot(document.getElementById('root')).render(
     <NavBar/>
)
```

React Ecosystem

Due to the popularity of React, there are many libraries made for it to provide extra features to round it out.

We will cover many React libraries, including:

- Routing libraries like **React Router**
- **Redux** for state management normal React states

Additionally we'll be covering **Next.js**, an entire framework built on React that provides its own routing, server-side rendering, and more.

Web Bundlers

How do web bundlers help
create, configure, and manage a React project?

What is a Web Bundler?

Modern JS projects, especially those that include frameworks, have a lot of JS modules and scripts, CSS, and image files.

Large projects can easily become an unoptimized mess with browser compatibility issues and module incompatibilities!

Web Bundlers solve this issue by optimizing and packaging everything for proper deployment, including module support in browsers.

Web Bundlers

There are many bundlers out there that have different performance features, ease of setup, and capabilities.

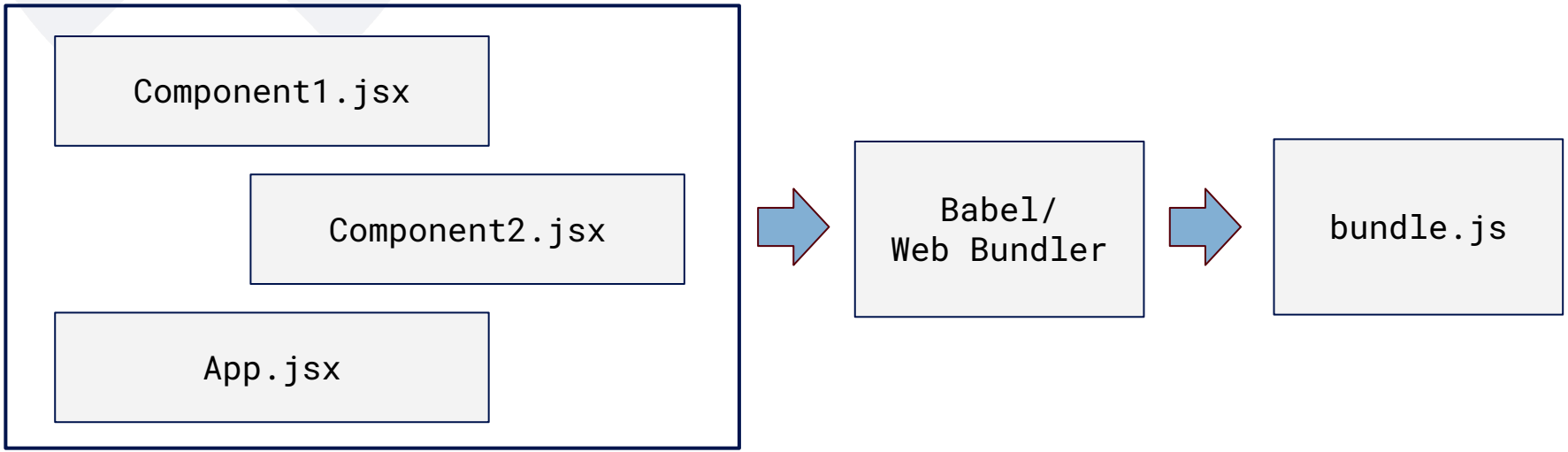
Webpack

Parcel

Vite (build tool + dev server that uses Rollup)

Bundlers and React

Browsers don't natively understand JSX, so bundlers will provide transpilation tools such as Babel to turn JSX into JS, among other optimizations.



Deprecated: create react app

You may find many resources that suggest create react app (CRA) as a way to create a React project. This method uses WebPack and Babel in the background.

However, as of early 2025, this method is now deprecated, with a warning if you do still use it!

Modern Build Tools: Vite

We'll suggest you use **Vite** (`npm create vite@latest`) to create your React projects.

It includes:

- **esbuild**, a code transformer for quickly transpiling JSX to JS
- **Rollup** as a bundling tool
- A very fast dev server for fast updates and startup

Activity: Create Vite Projects

https://edstem.org/us/courses/91614/lessons/159893/slides/939136

We'll be using Vite to make a TS project and a React project and explore the different components.

JSX and Component Syntax

What syntax do we use to create components in React?

Review: The "root" Container

Within a project's `index.html`, there is a container with the id "root" that React uses to render content to the page.

```
<div id="root"></div>
```

Your entry point to React will be some file that calls the `createRoot()` function from this HTML element, and a `render` function to define what will show up in that container:

```
import React from 'react';
import { createRoot } from 'react-dom/client';

createRoot(document.getElementById('root')).render(
     <p> Hello World! <p/>
)
```

JSX: JavaScript XML

JSX makes it easier for us to write React apps by converting HTML into React elements, allowing our scripts to create HTML.

```
const myElement =
document.createElement('p');
myElement.textContent = 'Hello World!';
document.getElementById('root').appendChild
(myElement);
```

Plain JS

```
const myElement = <p> Hello World! <p/>;

createRoot(document.getElementById('root'))
.render(
     myElement
)
```

React

JSX Syntax: Attributes & Styles

Common attributes like `class` are reserved by JS, so you have to use JSX versions like `className` instead.

```
<p class="highlight">I am made for plain JS, I use class!</p>
```

```
<p className="highlight">I am made for JSX, I use className!</p>
```

Additionally, any inline styles will be an object, and attributes will use camelCase, rather than kebab-case, like their JS counterparts:

```
<p style={{ color: 'red', fontSize: '12px' }}>I'm some red text, size 12!</p>
```

JSX Syntax: Multiple Lines

A single JSX expression can represent multiple lines of HTML if wrapped in parentheses ().

However, there must be only **one** parent/top level element for each expression. Often, you'll use an fragment, a set of empty tags <></> to wrap items together.

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);

createRoot(document.getElementById
('root')).render(
  myElement
);
```

JSX Syntax: Expressions

You can insert JavaScript expressions within JSX to execute JS code.

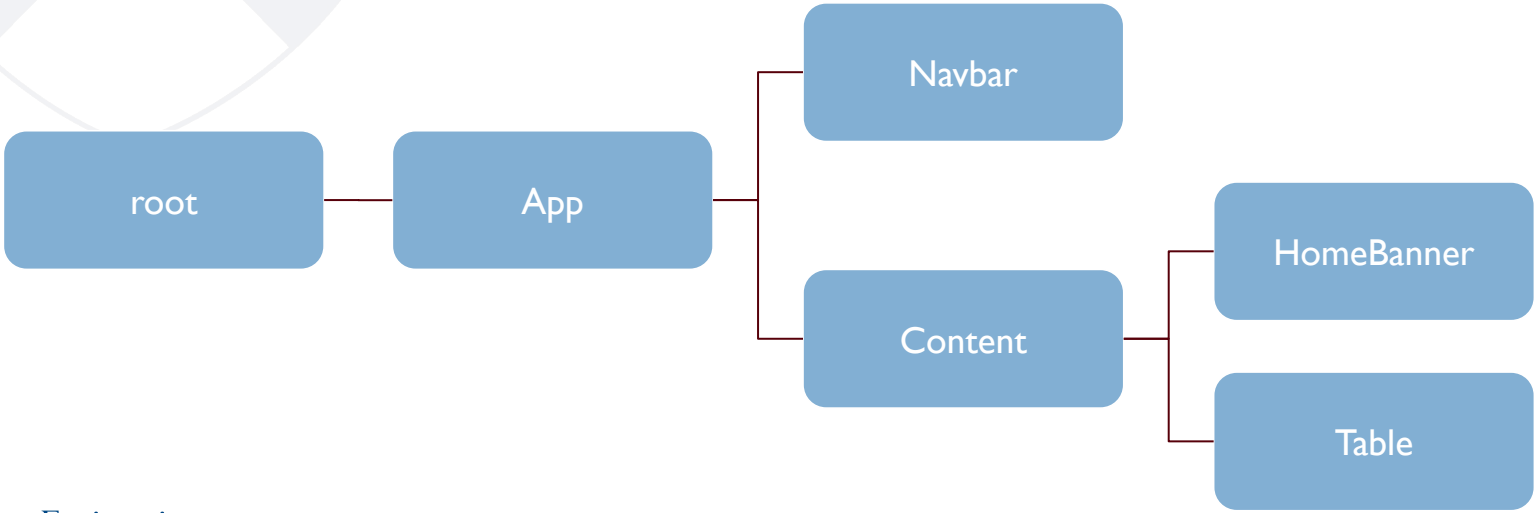
You define an expression by using the curly brackets {}

This allows you to use the breadth of JS syntax in the expression: variables, ternaries, string methods, and more!

```
<a className="button" href={loggedIn ? '/logout' : '/login'}>
    {loggedIn ? 'Log Out' : 'Log In'}
</a>
```

React Components

Components are the building blocks of React apps. We can represent a like a tree, that is made up of parent and child components.



Functional Components

Most React components use functional components as they are less complex and more top-to-bottom readable.

They often use hoisted **function declaration** (classic) or **arrow functions** (preferred in modern React).

```
function Hello(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Hoisted functions

```
const Hello = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

Arrow functions (preferred)

Component Logic

Within functional components, before you return JSX, you will often perform rendering logic, and even embed other functions inside the component for later use.

```
const Fibonacci = ({ n }) => {
  function fib(num) {
    if (num <= 1) return num;
    return fib(num - 1) + fib(num - 2);
  }

  const result = fib(n);

  return <p>Fibonacci of {n} is {result}</p>;
}
```

Using Components

```
const Card = () => {
  return (
     <div className='card'>
        ...
     </div>
  );
};

export default Card;
```

```
import Card from './Card'

const CardGrid = () => {
  return (
     <div className='grid-squares'>
          <Card/>
          <Card/>
          <Card/>
          <Card/>
     </div>
  );
};
```

Once declared, components provide reusable ways to insert JSX elements into other JSX elements.

Often, you'll have components declared in their own file and imported into other components where they are needed.

Class Activity

https://edstem.org/us/courses/91614/lessons/159893/slides/939297

React Props

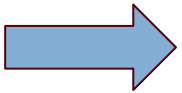
How do we pass data from parent to child components within React?

What are Props?

Props, short for "properties", are the arguments passed into React components. These take the form of the actual arguments of the functional components you write.

These props are read-only inside the component that receives them, representing a unidirectional data flow from parent to child.

```jsx
// App.jsx
<Hello name="Voravich" />
```



```jsx
const Hello = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};
```

Passing and Using Props

In some parent component, you can pass props as **attributes** on JSX elements.

Then in the child component, you access the props via the `props` object parameter, or **destructuring** the `props` object.

```
import Button from "./Button"

const ButtonExample = () => {
  return (
     <>
          <Button label="Click me"
                  color="red" />
     </>
  );
};
```

```
const Button = ({ label, color }) => {
  return <button style={{ background:
color }}>{label}</button>;
}
```

Functions as Props

Since prop data is immutable in the child, we need alternative ways for the child to communicate with with the parent.

Callback functions as props provide ways to children to communicate changes with the parent component.

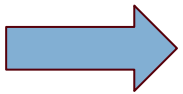
```
const Parent = () => {
  const handleClick = () => alert('Button clicked!');
  return <Child onClick={handleClick} />;
}

const Child = ({ onClick }) => {
  return <button onClick={onClick}>Click Me</button>;
}
```

Wrapper Components

You can access nested JSX content between the opening and closing tags of one component to another with `props.children` property.

This is useful for dynamic rendering and layout of certain components.

```
<Card>
  <h2>Title</h2>
  <p>Description</p>
</Card>
```



```
const Card = ({ children }) => {
  return <div
className="card">{children}</div>;
}
```

Prop Drilling

```
const Grandparent = () => {
  return <Parent user="voravich" />;
}

const Parent = (props) => {
  return <Child user={props.user} />;
}

const Child = (props) => {
  return <div>User: {props.user}</div>;
}
```

In some cases you may want to pass data across several nested layers of parent/child, leading to **prop drilling**.

This may become unwieldy, and could be solved using the hook useContext instead.

Props and TypeScript

With props, it's important to maintain type safety, since it is one of the ways data is passed through a React app.

With TypeScript you can used typed props in components:

```tsx
interface ButtonProps {
  label: string;
  color?: string;
  onClick: () => void;
}

const Button: React.FC<ButtonProps> = ({ label, color = "blue", onClick }) => (
  <button style={{ background: color }} onClick={onClick}>
    {label}
  </button>
);
```

TS Types for React

React.FC<P>: Functional component with typed props

React.ReactNode: Any valid renderable content (elements, strings, numbers, fragments, etc.)

React.ReactElement: The object returned from JSX

React.ChangeEvent<T>: Form/input event type

React.MouseEvent<T>: Mouse event type

Props.children and Generics

This pattern below can provide more flexibility when you are using `props.children` in your components with TypeScript:

```tsx
interface CardProps {
  children: React.ReactNode;
}

const Card: React.FC<CardProps> = ({ children }) =>
<div>{children}</div>;
```

Optional/Required and Default Props

By default, props are optional: if a prop is not provided in the parent, it will be undefined for its value in the child.

We can provide a default prop value in the case no values are provided.

TS can also allow us to make prop values required/optional.

```
interface MessageProps { text?: string; }

const Message: React.FC<MessageProps> = ({ text = "Hello!" }) => <div>{text}</div>;
```

Conditional & List Rendering

How we we leverage JSX to easily render items under certain conditions?

Conditional Rendering

A common pattern in React is using conditional logic to only render items under certain conditions.

This often gets used with the useStates hook to have automatic updates to components.

Plain HTML/JS Conditionals

With plain HTML/JS, you would need to manually update the DOM or trigger updates with eventListeners.

Messy, prone to bugs, hard to scale!

```
const user = { isLoggedIn: true, name: "Alice" };
const appDiv = document.getElementById("app");

function render() {
  if (user.isLoggedIn) {
    appDiv.textContent = `Welcome, ${user.name}!`;
  } else {
    appDiv.textContent = "Please log in";
  }
}

render()
```

React Conditionals

With React, we can automatically re-render items on state changes.

Cleaner and more scalable!

```
const Welcome = ({ user }) => {
  return (
    <div>
      {user.isLoggedIn ? `Welcome,
${user.name}!` : "Please log in"}
    </div>
  );
}
```

```
const App = () => {
  const [user, setUser] = useState({ isLoggedIn:
true, name: "Alice" });

  const toggleLogin = () => {
    setUser((prevUser) => prevUser.isLoggedIn
      ? { isLoggedIn: false, name: "" }
      : { isLoggedIn: true, name: "Alice" }
    );
  };

  return (
    <div>
      <button onClick={toggleLogin}>
        {user.isLoggedIn ? "Log out" : "Log in"}
      </button>
      <Welcome user={user} />
    </div>
  );
}
```

Conditionals with if/ternary

You will often use if statements (outside of JSX) or ternary operators (within JSX) to do conditional rendering.

```
const Greeting = ({ isMember, name }) => {
  return (
    <div>
      {
        isMember
          ? <h2>Welcome back, {name}!</h2>
          : <h2>Hello, guest. Please sign up.</h2>
      }
    </div>
  );
}
```

Conditionals with &&

Leveraging truthy/falsy values, we can allow something to be rendered only when some value exists or is truthy.

```
const Notification = ({ message }) => {
  return (
    <div>
      {message && <div className="notification">🔔 {message}</div>}
    </div>
  );
}
```

Conditionals with Switch

Switch cases can allow you to handle more than 2 cases.

```
const StatusMessage = ({ status }) => {
  switch (status) {
    case "loading": return <Spinner />;
    case "error": return <Error />;
    case "success": return <Success />;
    default: return null;
  }
}
```

List Rendering

JSX makes it easy to dynamically render an entire list of items as multiple JSX elements.

Each list element needs a **unique key** so that React can identify elements for updates and optimization.

```
function UserList() {
  const users = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
    { id: 3, name: "Charlie" },
    { id: 4, name: "Diana" }
  ];

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Class Activity

https://edstem.org/us/courses/91614/lessons/159893/slides/939322

Events and Inputs

How do we handle user actions and inputs in React?

React Event Handling

React can handle events similar to HTML DOM events.

There's a few notable differences:

- Events are written in camelCase (onClick, not onclick)
- Handlers are often written as functions called inside of JSX.

```
const MyButton = () => {
  function handleClick() {
    alert("You clicked me!");
  }

  return <button onClick={handleClick}>Click Me</button>;
}
```

Common React Event Listeners

onClick: User clicks an element (button, div, etc.)

onChange: User input changes (input, textarea, select)

onSubmit: Form submission

onMouseEnter / onMouseLeave: Mouse moves in/out of an element

onKeyDown / onKeyUp: User presses keyboard keys

onFocus / onBlur: Input focus handling

Forms and Inputs

React forms are **controlled components**, meaning input values are synced with the component's state, not the DOM.

Forms often use `onSubmit`, while inputs often use `onChange` to manage updates.

Don't forget `preventDefault`!

```
const MyForm = () => {
  const [name, setName] = useState("");

  function handleChange(e) {
    setName(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    alert(`Hello, ${name}!`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={handleChange}
placeholder="Your name" />
      <button type="submit">Say hello</button>
    </form>
  );
}
```

Class Activity

https://edstem.org/us/courses/91614/lessons/159893/slides/939391