

Front-end



Development

# Lesson 10: React (Part 1)



University of Tehran



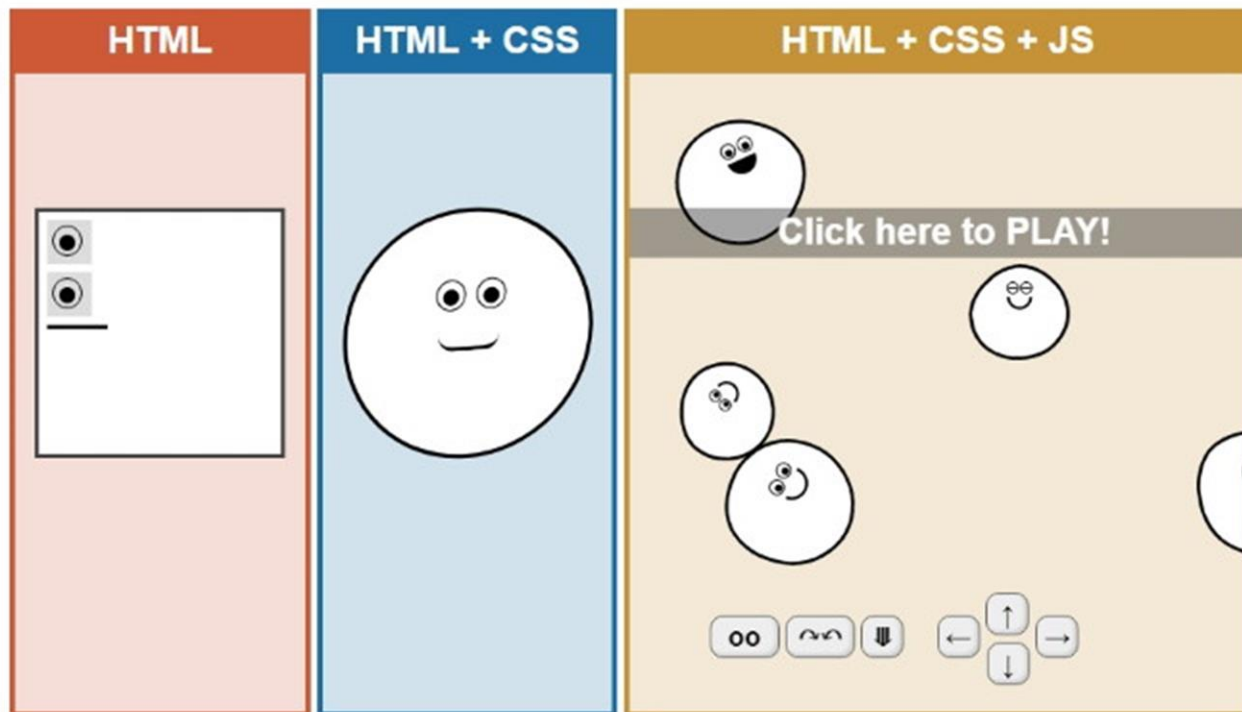
ACM  
Summer of Code 2024

Lecturer:  
Misagh Mohaghegh

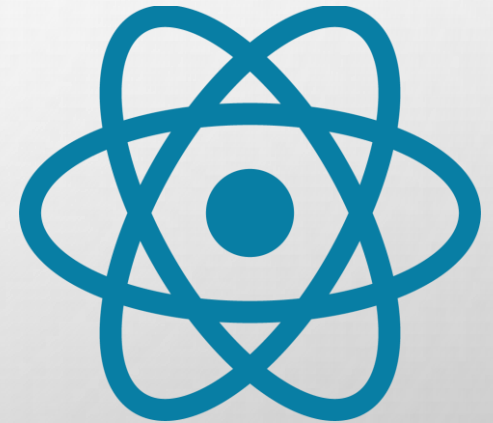
# A Short Review...

- Local Storage
- Object properties and methods
- Object accessors (getters and setters)
- Object constructor function
- Instanceof
- Object prototype chain and own properties
- Classes
- Class inheritance
- Static properties and methods
- "use strict";

Time for...



+



# JavaScript





# Modules



# Modules

Modules were introduced in ES6 which allowed us to break our JS code into separate files. This makes the codebase easier to maintain.

To use them, each JS file can **export** some variables, functions, or classes. Other JS files can **import** the exported things of other files.

There always exists a top-level JS file which imports other JS files (and those can also import other files) After the browser gets the top-level file, it makes requests to get the imported files.

# Export

To export a variable, function, or class, we use the export keyword either next to the declaration, or at the end of the file:

```
export const myVar = 10;  
export function myFunc() { return 10; }
```

```
const myVar = 10;  
function myFunc() { return 10; }  
  
export { myVar, myFunc };
```

This is called a **Named Export**.

# Default Export

We can have exactly one default export in our file:

```
export default function myFunc() {  
    return 10;  
}
```

```
function myFunc() {  
    return 10;  
}  
  
export default myFunc;
```



# Import

We can now import whatever was exported in other files.

```
// util.js
export function myFunc() {
    return 10;
}

// main.js
import { myFunc } from "./util.js";
console.log(myFunc()); // prints 10
```

```
// multiple imports
import { myFunc, myVar } from "./util.js";
```

# Default Import

To import a default exported variable, we omit the curly braces and can use any name:

```
// util.js
export default function myFunc() {
  return 10;
}

// main.js
import whatever from "./util.js";
console.log(whatever()); // prints 10
```

# Import Alias

We can also change the name of a normal named export when importing:

```
// util.js
export function myFunc() {
  return 10;
}

// main.js
import { myFunc as test } from "./util.js";
console.log(test()); // prints 10
```

The same can be done when exporting:

```
export { myVar as anotherName };
```

# HTML Script

To use modules, we need to specify that our script should be treated as one. Otherwise it will not run the import statements.

```
<script src="main.js" type="module"></script>
```



# Bundling



# Bundling

For every module file that we have, the browser makes a request to get it. This is not very efficient. This problem grows as our codebase grows and we have more files and more libraries.

JS bundlers turn all of our modules into a single JS file that the browser requests.

We do not change our code and still use modules like we used to.

The bundler will give us a code that has changed to work with a single compiled JS file.

Webpack is the most famous JS bundler.

There are other options such as Vite, Parcel, Esbuild, SWC, and Rollup.



Bundlers also support Sass (turn Sass files into a single CSS file)

# Minification

Bundlers usually perform code minification.

Minification removes all **comments** and unnecessary **whitespaces**.

This leads to a much smaller file size for our code.

A smaller code means that the HTTP request will get the response faster.

The browser will also execute the code faster.

HTML, CSS, and JS all get minified for **production** use.

Minification is not used for **development** because it reduces code readability.

# Minification



```
.entry-content p {  
  font-size: 14px !important;  
}  
  
.entry-content ul li {  
  font-size: 14px !important;  
}  
  
.product_item p a {  
  color: #000;  
  padding: 10px 0px 0px 0;  
  margin-bottom: 5px;  
}
```

Minify



```
.entry-content p,.entry-content ul li  
{font-size:14px!important}.product_item  
p a{color:#000;padding:10px 0 0;margin-  
bottom:5px;border-bottom:none}
```

```
// program to check if the number is even or odd  
// take input from the user  
const number = prompt("Enter a number: ");  
  
//check if the number is even  
if(number % 2 == 0) {  
  console.log("The number is even.");  
}  
  
// if the number is odd  
else {  
  console.log("The number is odd.");  
}
```

```
const number=prompt("Enter a number: ");number%2==0?console.log("The number is even."):console.log("The number is odd.");
```

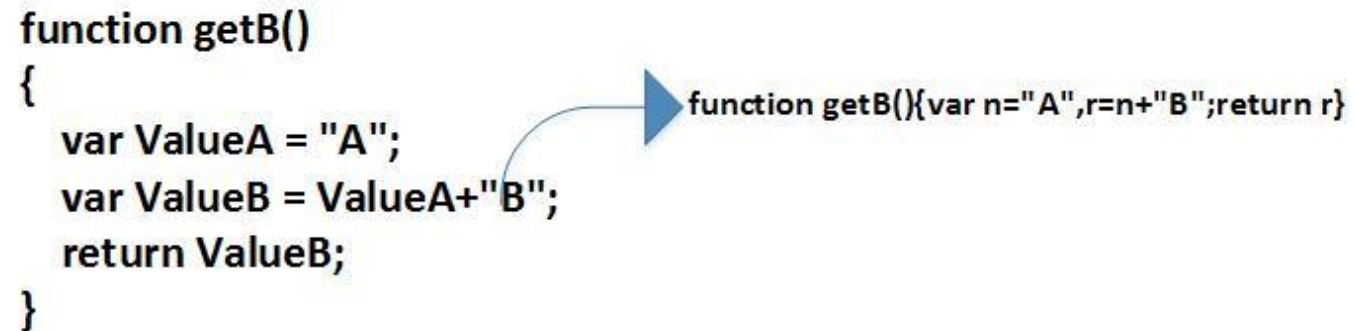


# Uglification

JS code can also be 'uglified' using tools such as UglifyJS.

This process changes the variable names to smaller 1 character names.

This makes the code almost unreadable but reduces the file size.



The diagram illustrates the process of JavaScript uglification. On the left, a readable function is shown:

```
function getB()  
{  
  var ValueA = "A";  
  var ValueB = ValueA+"B";  
  return ValueB;  
}
```

A blue arrow points from this code to the minified version on the right:

```
function getB(){var n="A",r=n+"B";return r}
```

```
!function(a){"use strict";function b(a){var b=a.length,d=c.type(a);return"function"!==d&&!c.isWin  
;if(!(k&&k.isSVG||"tween"===l||A.Names.prefixCheck(1)[1]!==!1|A.Normalizations.registered[1]!==d
```

# Transpilation

We have already talked about transpilation.

Tools such as Babel convert our JS code into JS code that does not use the latest ECMAScript features.

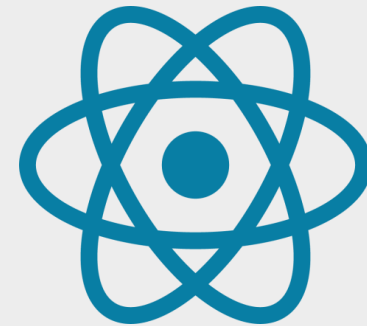
This helps us use the latest features while providing support for older browsers through polyfills.

TypeScript also compiles to JavaScript to run on the browser.

```
// using ECMAScript 2020  
// nullish coalescing operator  
const x = addr ?? "none";
```

```
// transpiled code  
const x = (addr !== null && addr !== undefined)  
  ? addr : "none";
```

React





# Imperative vs Declarative



# Imperative vs Declarative

Imperative code specifies how to do what we want. This is done through step-by-step codes. Declarative code specifies what we want. There is no focus on how it should be done.

```
const myList = [1, 2, 3];  
  
// Imperative  
let total = 0;  
for (let x of myList) { total += x; }  
  
// Declarative  
let total = sum(myList);
```

# Imperative UI

In an imperative design, we specify how to do things in a step-by-step manner.

What we have been doing to change our UI in JS was using an imperative paradigm.

For example, in our to-do list, we would add the item to a list and then to the DOM manually.

# Declarative UI

In a declarative approach, we specify what the final state should be.

For example, we specify that a heading should have the value of a variable called **h1text**.

If we change the variable, we should select the heading and change its text.

But in a declarative UI, after changing the variable, the heading text changes automatically.

So only the final state was described and changes to reach that state happen automatically.

# State

The variable we talked about (`h1text`) would then be called a **state** for our view.  
Whenever a state changes, the components that use the state are updated accordingly.





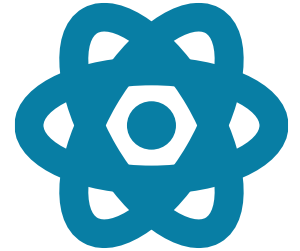
React



# React

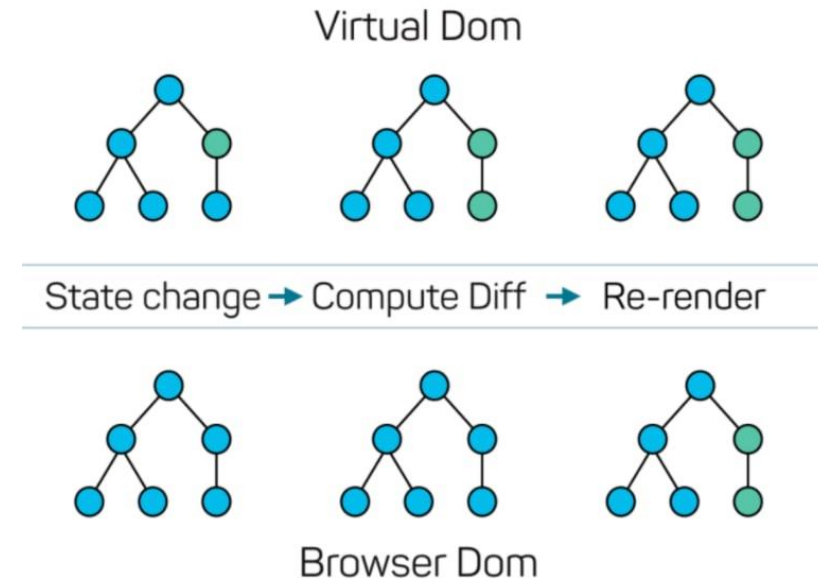
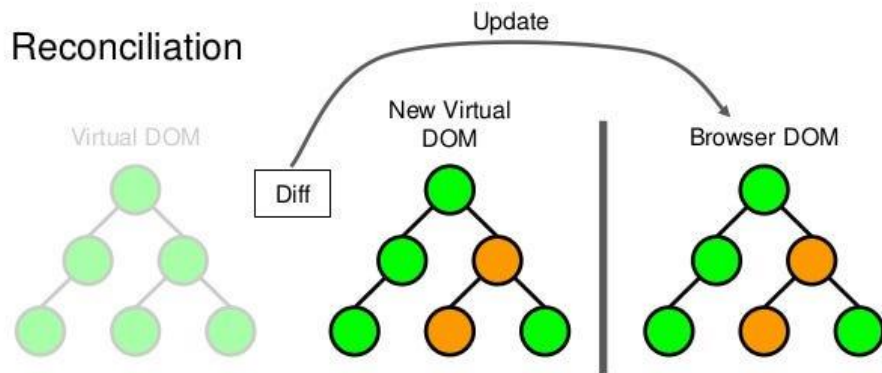
React is a JS framework for building user interfaces using components.  
It is an open-source project created by Meta (Facebook) in 2013.

It uses a declarative UI approach and components.



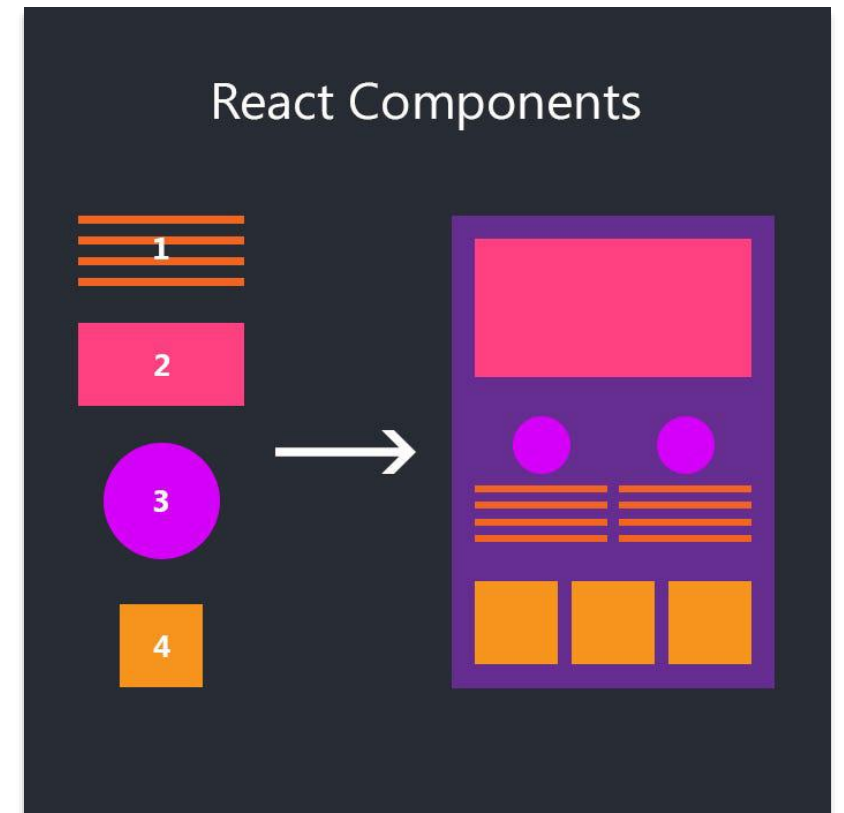
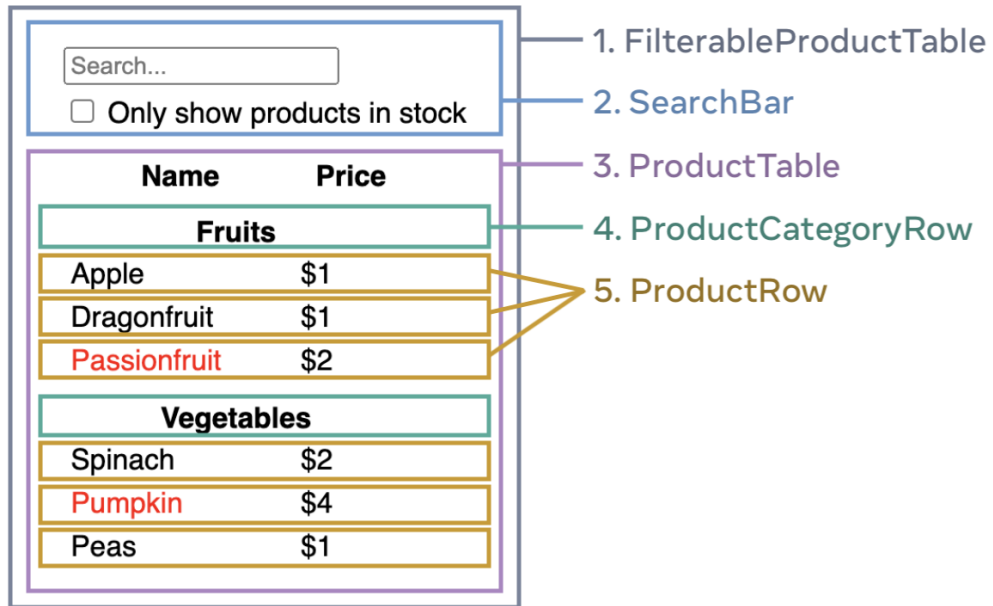
# Virtual DOM

React uses a virtual DOM (which is a lightweight version of DOM in an object) to apply your changes. The virtual DOM does not change the page. After the virtual DOM changes, React checks the differences and updates only the changed parts in the real browser DOM (Reconciliation). This method is used to make updates faster.



# Thinking in Components

In React, we break down our code into many components.



# JSX

JSX (JavaScript XML) is a syntax extension to JS that React uses to describe the UI.

Using JSX, we basically include HTML inside of JS code.

We can use curly braces {} to put JS code inside the JSX HTML.

```
const element = <h1>Hello, world!</h1>;

function func(link) {
  let name = "me";
  return <a href={link} title="test">Hi {name}!</a>
}
```

# JSX Void Elements

We should close void elements like XHTML:

```
 // invalid  
 // correct  
  
<br> // invalid  
<br /> // correct  
  
<input type="text"> // invalid  
<input type="text" /> // correct
```

# Components

In React, components are basically JS functions that start with a capital letter and return HTML code. They can be used just like HTML elements. HTML elements are lowercase while React components are capitalized.

```
function ListItem() {  
  return (  
    <li>  
      <img src={photo} alt="sample photo" />  
      <p>Example text</p>  
    </li>  
  );  
}
```

```
function MyList() {  
  return (  
    <div>  
      <h1>This is my list</h1>  
      <ListItem />  
      <ListItem />  
    </div>  
  );  
}
```

# Fragment

The React.Fragment element (<>) is used to group same-level elements when returning:

```
function Hero() {  
  return (  
    <>  
      <h1>Hi!</h1>  
      <p>Welcome!</p>  
    </>  
  );  
}
```



# Props

Props or properties are arguments passed to components:

```
function ListItem({ imgSrc, text }) {  
  return (  
    <li>  
      <img src={imgSrc} alt="sample photo" />  
      <p>{text}</p>  
    </li>  
  );  
}
```

```
function MyList() {  
  return (  
    <div>  
      <h1>This is my list</h1>  
      <ListItem imgSrc="https://a"  
        text="The Sea!" />  
    </div>  
  );  
}
```

# Prop Default Value

Props can have default values.

If a prop is not passed and does not have a default value, undefined is used.

```
function Sample({ text, size = 10 }) {  
  return (  
    ...  
  );  
}
```

# Children Prop

The children prop contains all child elements of a component.

```
function Container({ children }) {  
  return (  
    <div className="container">  
      {children}  
    </div>  
  );  
}
```

```
function Sample() {  
  return (  
    <Container>  
      <h1>Child element</h1>  
    </Container>  
  );  
}
```

# Class Components

Before React 16.8 (2019) introduced Hooks, class components were mostly used. Function components should be preferred as of now.

```
class Greeting extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}!</h1>;  
  }  
}  
  
// the same in function components:  
function Greeting({ name }) {  
  return <h1>Hello, {name}!</h1>;  
}
```

# Rendering Lists

We need to use loops to render lists. It is common to use `.filter` and `.map` methods.

```
const items = [
  { id: 1, name: 'A' },
  { id: 2, name: 'B' }
];

function MyList() {
  const listItems = items.map(x =>
    <li key={x.id}>{x.name}</li>
  );
  // we could also use a (for x of items) loop and
  // push HTML elements to the array
  return <ul>{listItems}</ul>;
}
```

```
function MyList() {
  // putting the list directly in JSX
  return (
    <ul>
      {items.map(x => (
        <li key={x.id}>{x.name}</li>
      ))}
    </ul>
  );
}
```

# Looping

To repeat an element/component n times, we can loop over an array of n elements:

```
function MyRepeat() {  
  return (  
    <div>  
      {[...Array(4)].map((_, i) => (  
        <SomeComponent />  
      ))}  
    </div>  
  );  
}
```

# Conditionals

We can write conditional code inside JSX:

```
function Home({ isLoggedIn }) {  
  const content = isLoggedIn ? <UserPanel /> : <AboutPage />;  
  return (  
    <div>  
      {content}  
    </div>  
  );  
}
```

```
function Home({ isLoggedIn }) {  
  return (  
    <div>  
      {isLoggedIn ? (  
        <UserPanel />  
      ) : (  
        <AboutPage />  
      )}  
    </div>  
  );  
}
```

# Conditionals

There is a short form when we don't have an else branch:

```
function Home({ isLoggedIn }) {  
  return (  
    <div>  
      {isLoggedIn && <UserPanel />}  
    </div>  
  );  
}
```



# HTML Attributes

Some HTML attributes are handled differently in React:

```
<p class="align-right">Test</p> // invalid
<p className="align-right">Test</p>

<label for="input-id">Name</label> // invalid
<label htmlFor="input-id">Name</label>

<span style="color: red; height: 20px;">Test</span> // invalid
<span style={{ color: 'red', height: 20 }}>Test</span>

// the double {} is just passing a JS object to it:
let myStyle = { color: 'red', height: 20 }
<span style={myStyle}>Test</span>
```

# HTML Attributes

Some HTML attributes are handled differently in React:

```
<select name="test">
  <option value="A">a</option>
  <option value="B" selected>b</option> // invalid
</select>
```

```
<select name="test" defaultValue="B">
  <option value="A">a</option>
  <option value="B">b</option>
</select>
```

```
// we will learn that 'value' should be used for controlled forms
// instead of defaultValue.
```

# HTML Attributes

Some HTML attributes are handled differently in React:

```
<button onclick="myFunc('arg');">Click</button> // invalid
```

```
<button onClick={() => myFunc('arg')}>Click</button>
```

```
// Notice the capital C letter.
```

```
// The HTML event handlers (such as onclick) would not be used in normal JS code
```

```
// and we would use addEventListener.
```

```
// In React however, the event handlers in camelCase are used for listening.
```

```
// For example, the 'mouseenter' event would be onPointerEnter.
```

```
// or onChange for the input 'change' event.
```

```
// The function can also take the event object: onClick={event => ...}
```

# Installation

Node.js should be installed first.

We have talked about Node.js, it is a way to run JS code outside of the browser (a JS runtime environment). The Node Package Manager (NPM) is used on the command-line to install libraries and more.



```
C:\Users\Lenoidea\Desktop>node
Welcome to Node.js v20.16.0.
Type ".help" for more information.
> console.log("Hi");
Hi
undefined
> |
```

# Installation

To setup a React project, there are many methods to choose from.

## Basic React:

- CRA (Create React App): This was the official way to setup projects made by the React team. However, it was removed from the documentations a year ago.
- Vite: This is another way to setup a React project. It is much faster and also supports TypeScript.

## Full-stack React:

- Next.js: This is a full-stack React framework. It supports all features and also server-side rendering.  
The React documentations seem to recommend using this for all projects.
- Remix: Another full-stack React framework.

# CRA

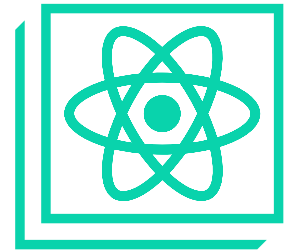
We will stick to the CRA method because it is easier to focus on only React. In options such as Next.js we also have to mind the server features which may not be the best for learning.

To setup a CRA project, use the following command:

```
npx create-react-app my-app
```

npx is Node Package Execute which runs a template (create-react-app here).

CRA includes Webpack, Babel, and ESLint.



# Structure

The CRA project structure and files are explained in code.

```
<CODE />
```

Thank you for your attention.