

# React.js

Regards: Hassan Bilal

# TABLE OF CONTENTS

**01** > **useState Hook**

**02** > **useEffect Hook**

**03** > **useRef Hook**

**04** > **useMemo Hook**

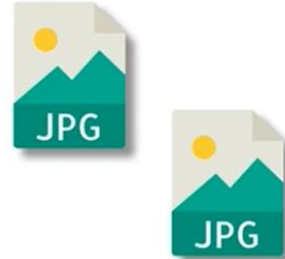
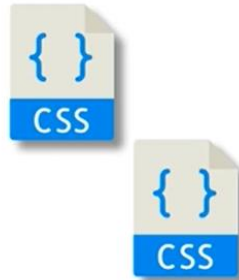
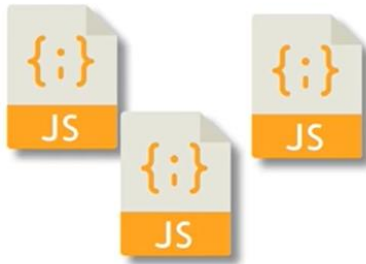
**05** > **useCallback Hook**

**01**

# Webpack

# What is Webpack

- Helps us with one important problem.
- When we build a website.
- We get all these different files.. JavaScript, CSS, JPG

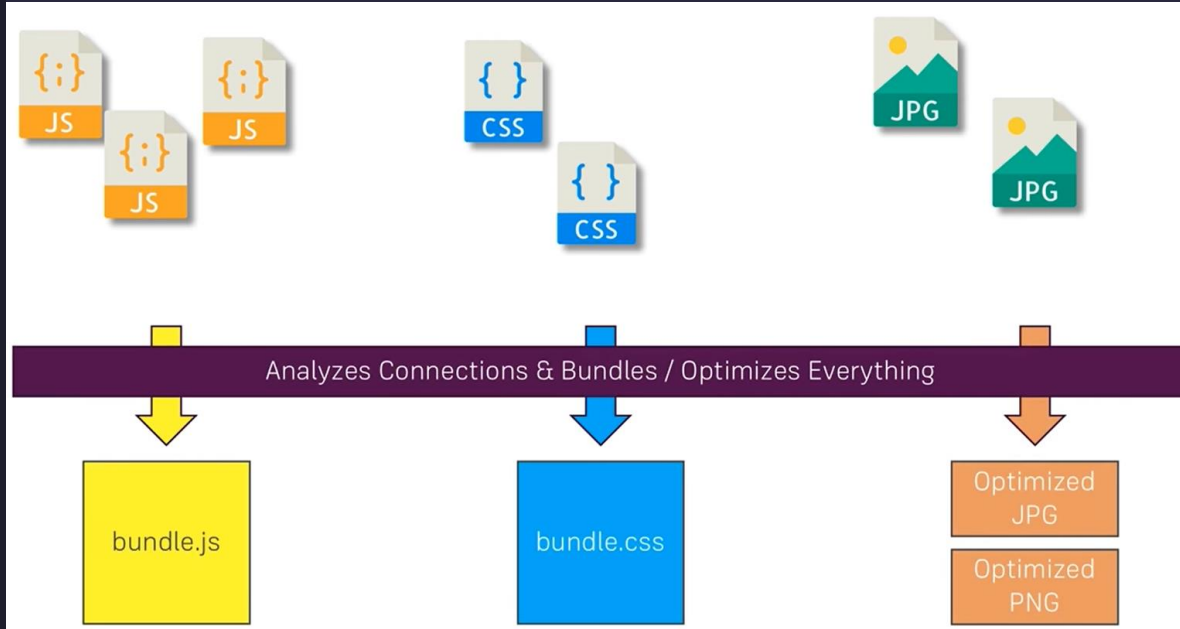


# What is Webpack

- For JavaScript we need to bundle all into one file for shipping them.
- Because most browser don't support multi file JavaScript apps.
- Where we try to import form other files.
- That is where webpack comes into play.

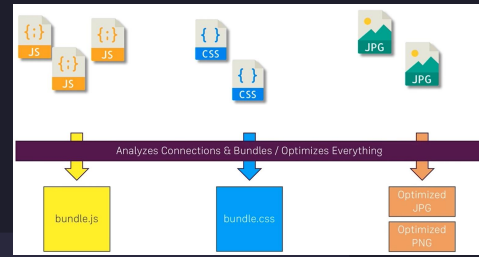
# What is Webpack

- Bundle and optimize all these things.

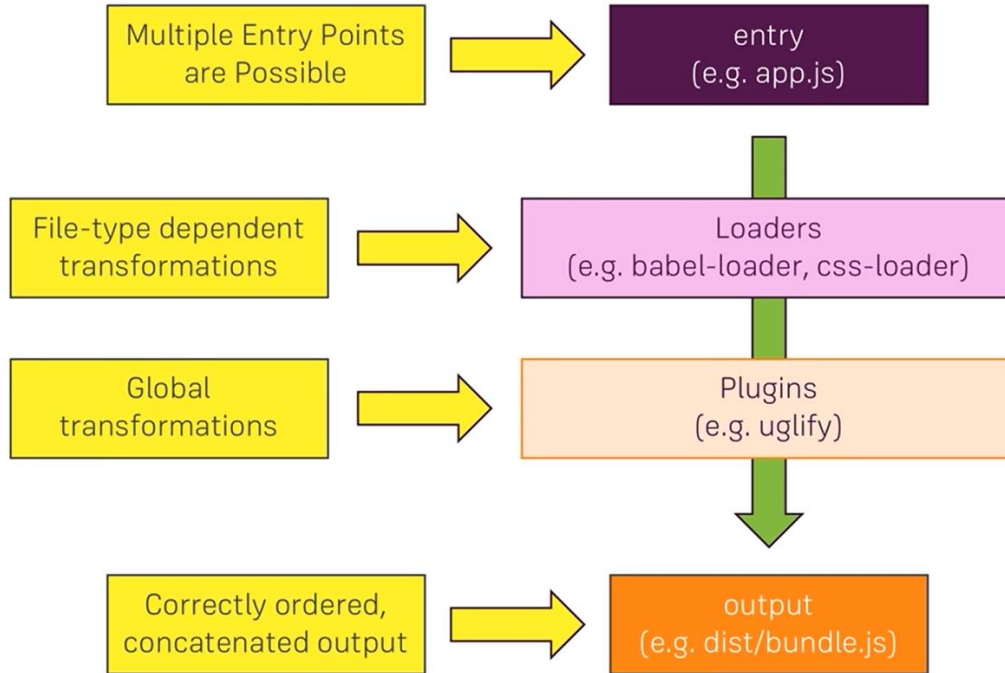


# What is Webpack

- It helps us setup a development work flow.
- Where we have some input files.
- Which attach bundles together.
- So, we get some optimized and bundled files.
- Which we can then deploy.
- So we can ship the code as small as possible.



# What is Webpack – How it works





**02**

# Code-Splitting

# Code-Splitting – Bundling

- Most React apps will have their files “bundled”.
- Using tools like Webpack, Rollup or Browserify.
- Bundling is the process of following imported files and merging them into a single file: a “bundle”.
- This bundle can then be included on a webpage.
- To load an entire app at once.

# Code-Splitting – Bundling – Example

- Your bundles will end up looking a lot different than this.

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

**Bundle:**

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

# Code-Splitting

- But as your app grows, your bundle will grow too.
- Especially if you are including large third-party libraries.
- You need to keep an eye on the code you are including in your bundle.
- So that you don't accidentally make it so large.
- That your app takes a long time to load.

# Code-Splitting

- To avoid winding up with a large bundle.
- It's good to get ahead of the problem and start “splitting” your bundle.
- Code-Splitting is a feature supported by bundlers like Webpack.
- Which can create multiple bundles that can be dynamically loaded at runtime.

# Code-Splitting


- Code-splitting your app can help you “lazy-load”.
- Just the things that are currently needed by the user.
- Which can dramatically improve the performance of your app.
- While you haven't reduced the overall amount of code in your app.
- You've avoided loading code that the user may never need.
- And reduced the amount of code needed during the initial load.

# Code-Splitting – Import( )

- The best way to introduce code-splitting into your app is through the dynamic import() syntax.

```
import { add } from './math';  
  
console.log(add(16, 26));
```

```
import("./math").then(math => {  
  console.log(math.add(16, 26));  
});
```




# Code-Splitting – React.Lazy

- lets you render a dynamic import as a regular component.

```
import OtherComponent from './OtherComponent';
```

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```






# Code-Splitting – React.Lazy

- This will automatically load the bundle containing the OtherComponent when this component is first rendered.

```
import OtherComponent from './OtherComponent';
```

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```




# Code-Splitting – React.Lazy

- React.lazy takes a function that must call a dynamic import().
- This must return a Promise.
- Which resolves to a module with a default export containing a React component.

```
import OtherComponent from './OtherComponent';
```

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```



# Code-Splitting – React.Lazy – Suspense

- The lazy component should then be rendered inside a Suspense component.
- Which allows us to show some fallback content.
- Such as a loading indicator.
- While we're waiting for the lazy component to load.

# Code-Splitting – React.Lazy – Suspense

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

# Code-Splitting – React.Lazy – Suspense

- You can even wrap multiple lazy components with a single Suspense component.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

**03**

# Intro to React Hooks

# What are Hooks

- Hooks allow function components to have access to state and other React features.
- Because of this, class components are generally no longer needed.
- Although Hooks generally replace class components, there are no plans to remove classes from React.
- Hooks allow us to "hook" into React features such as state and lifecycle methods.

# What are Hooks – Rules of Hooks

- Only call Hooks at the top level.
- Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components.



# What are Hooks – Types of Hooks

- `useState`
- `useEffect`
- `useRef`
- `useContext`
- `useReducer`
- `useCallback`
- `useMemo`
- There are more...

**03**

# **useState hook**

# useState hook

- State generally means data or properties.
- The React useState Hook allows us to track state in a component.

## useState hook – Usage

- Import it into our component.
- Notice that we are destructuring useState from react as it is a named export.

```
import { useState } from "react";
```

# useState hook – Initialize useState

- We initialize our state by calling useState in our function component.
- useState accepts an initial state and returns two values:
  - The current state.
  - A function that updates the state.

```
import { useState } from "react";

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

# useState hook – Initialize useState

- The first value, color, is our current state.
- The second value, setColor, is the function that is used to update our state.

```
import { useState } from "react";

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

## useState hook – Read state

```
function FavoriteColor() {  
  const [color, setColor] = useState("red");  
  
  return <h1>My favorite color is {color}!</h1>  
}
```

# useState hook – Update state

```
function FavoriteColor() {  
  const [color, setColor] = useState("red");  
  
  return (  
    <>  
      <h1>My favorite color is {color}!</h1>  
      <button  
        type="button"  
        onClick={() => setColor("blue")}  
      >Blue</button>  
    </>  
  )  
}
```



# useState hook – Multiple useState

```
function Car() {  
  const [brand, setBrand] = useState("Ford");  
  const [model, setModel] = useState("Mustang");  
  const [year, setYear] = useState("1964");  
  const [color, setColor] = useState("red");  
  
  return (  
    <>  
      <h1>My {brand}</h1>  
      <p>  
        It is a {color} {model} from {year}.  
      </p>  
    </>  
  )  
}
```

# <QnA>

>

Thanks!

<

Regards: Hassan Bilal