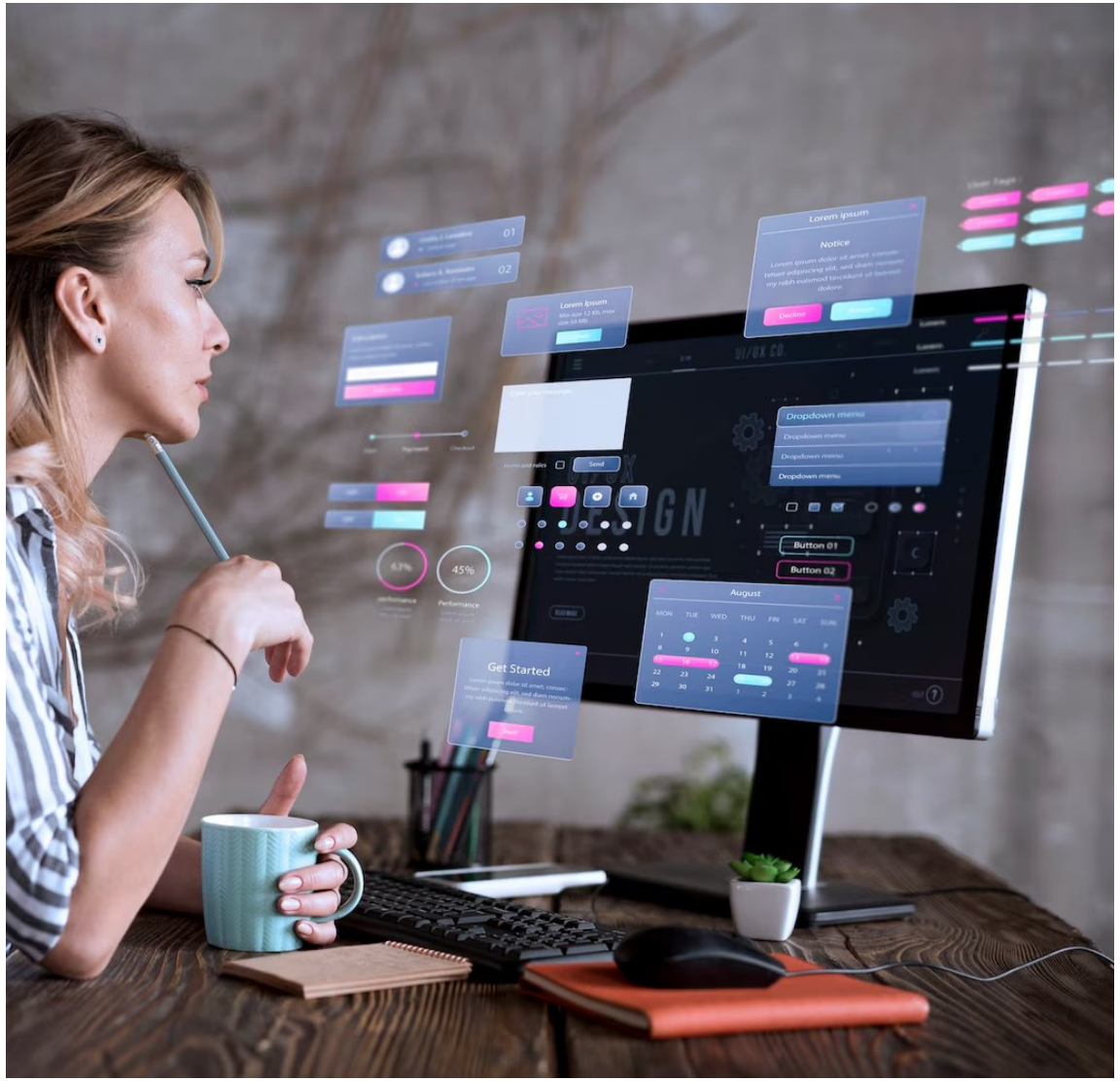
Frontend Development

Thusithanjana Thilakarathna

What is Frontend development?

Client-side development.

- Frontend development is the development of visual and interactive elements of a website that users interact with directly.
- Also known as Client-side development.



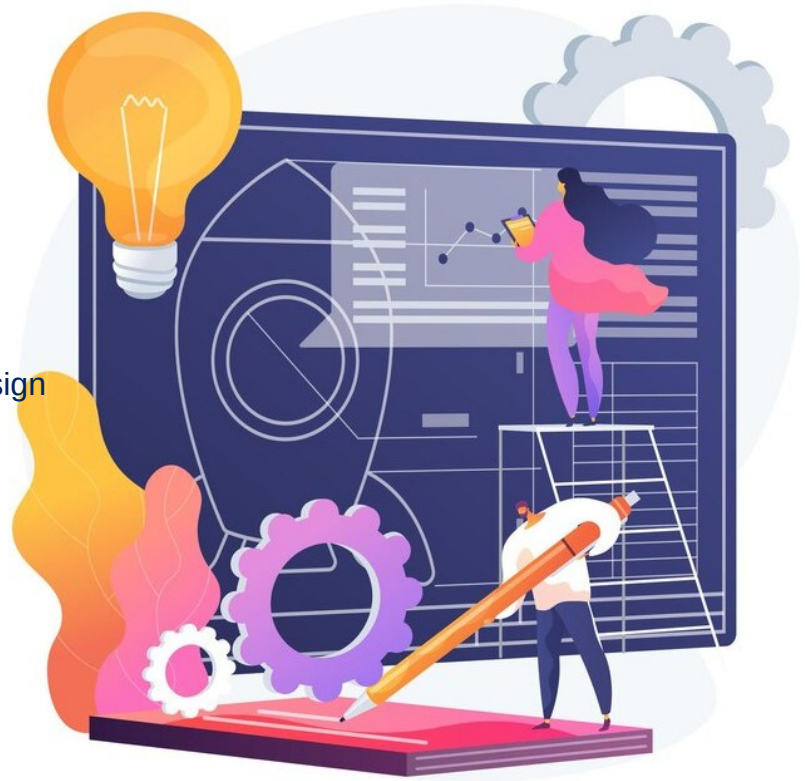
Why is Frontend development important?

- Backend vs. Frontend
  - Frontend web development focuses on **creating a good look and feel for the user.**
  - Backend web development focuses on engineering the web application's structure, logic and data.

Why is Frontend development important?

- Frontend development can be considered under two aspects.
  - **Designing** a good look and feel
    - HCI Design, UI/ UX etc.
  - **Engineering** the designed good look and feel
    - Frontend technologies
      - E.g., HTML, CSS, JS, Different frameworks, libraries etc.

Human-Computer Interaction design



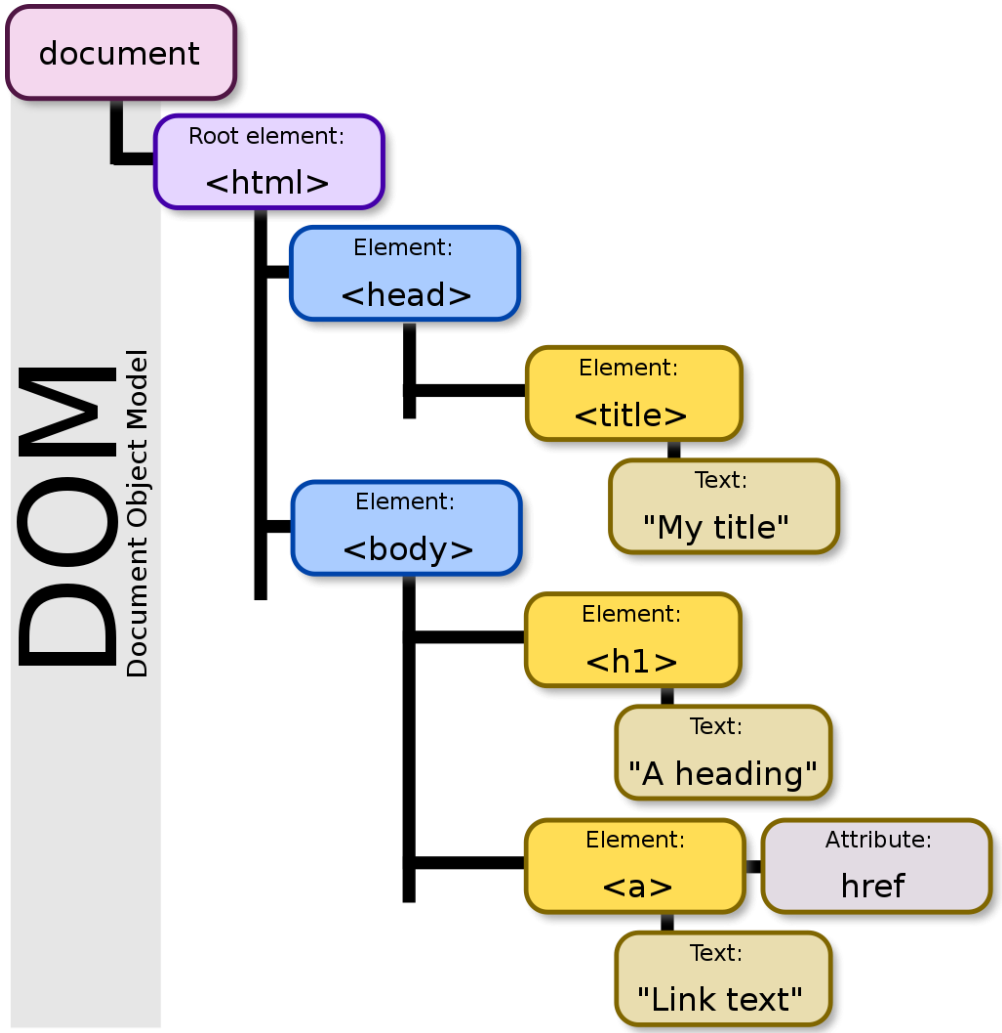
HTML, CSS, JS

- Frontend development is done with a combination of technologies in which,
  - **HTML** provides the structure
  - **CSS** provides the styling and layout
  - **JavaScript** provides the dynamic behavior and interactivity

Document object model (DOM)

- What do we actually manipulate with these technologies?
  - Document Object Model (DOM)!

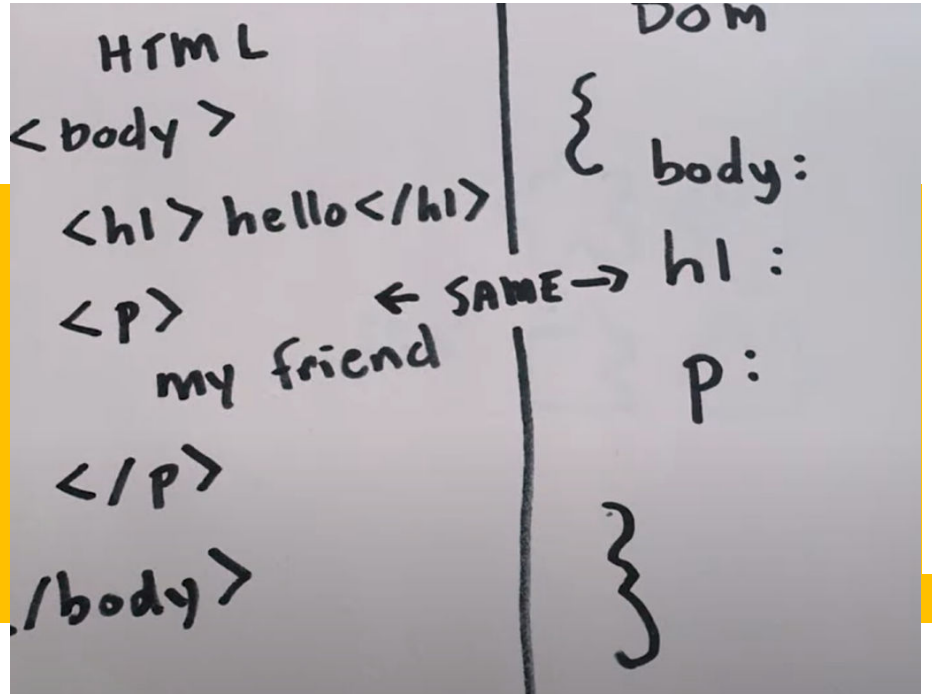
Document object model (DOM)



Source: https://en.wikipedia.org/wiki/Document_Object_Model

can all be accessed and manipulated using the DOM and a scripting language like JavaScript.
The DOM is not part of the JavaScript language, but is instead a Web API used to build websites.

Document Object Model (DOM)



- Document Object Model (DOM) – An object-oriented representation of the HTML structure
- DOM is not a JS functionality.
  - JS is the mostly used language to manipulate it, but other languages can also be used.
- It is just a Web API.
- [YouTube] The DOM in 4 minutes

HTML

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
```

- HTML stands for Hyper Text Markup Language
- HTML is the standard markup language for creating Web pages
- HTML describes the structure of a Web page
- HTML consists of a series of elements
- HTML elements tell the browser how to display the content
- HTML elements label pieces of content such as "this is a heading", "this is a paragraph", "this is a link", etc.
- HTML5 is the latest and most enhanced version of HTML.

CSS

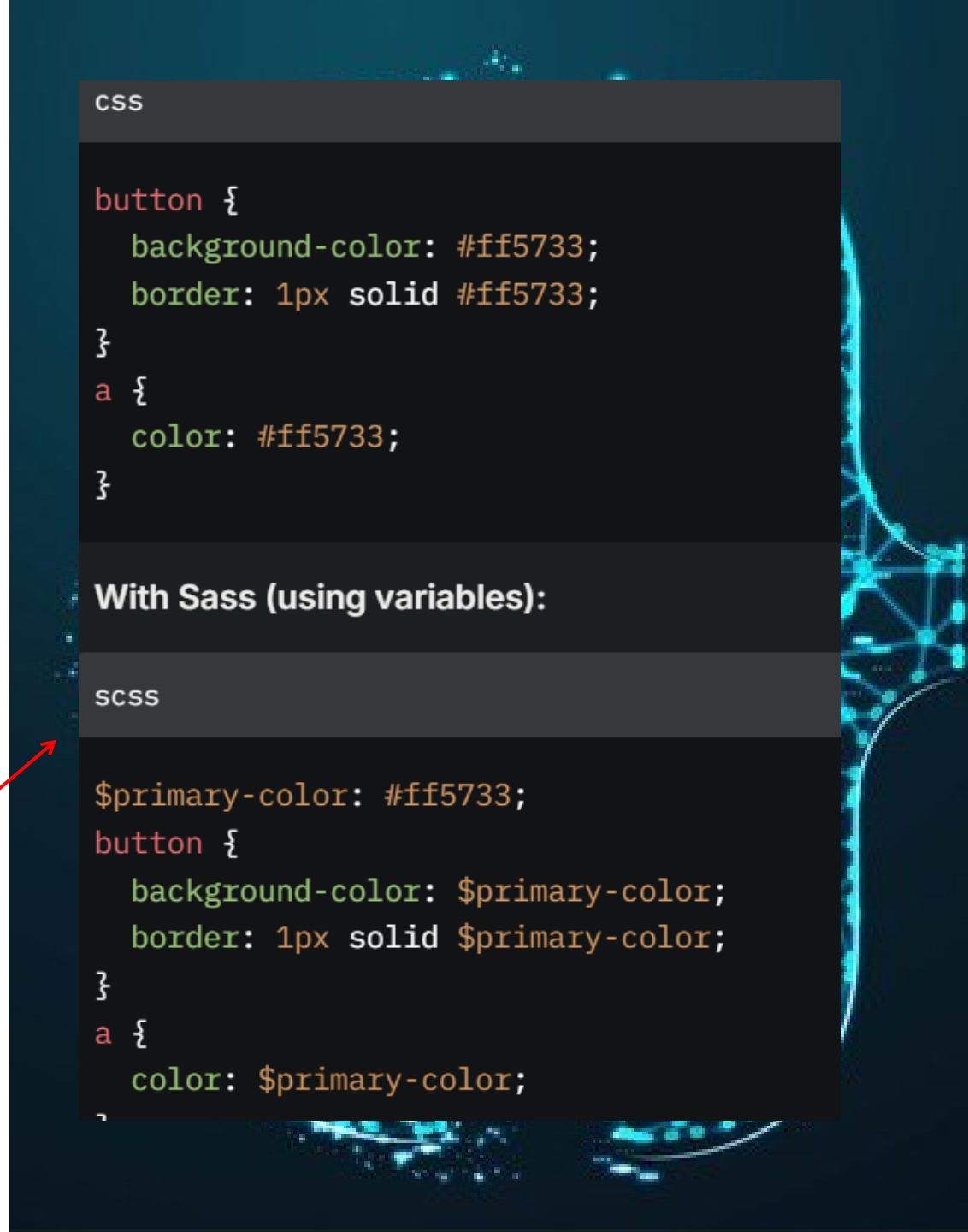
- CSS stands for Cascading Style Sheets
- CSS describes how HTML elements are to be displayed on screen, paper, or in other media
- CSS saves a lot of work. It can control the layout of multiple web pages all at once

allows you to centralize styling for an entire website.
By linking a single CSS file to multiple HTML pages

- External stylesheets are stored in CSS files

SASS

- Sass stands for Syntactically Awesome Stylesheet
- Sass is an extension to CSS
- Sass is a CSS pre-processor
- Sass is completely compatible with all versions of CSS
- Sass reduces repetition of CSS and therefore saves time
- Sass was designed by Hampton Catlin and developed by Natalie Weizenbaum in 2006
- Sass is free to download and use
- Stylesheets are getting larger, more complex, and harder to maintain. This is where a CSS pre-processor can help.
- Sass lets you use features that do not exist in CSS, like variables, nested rules, mixins, imports, inheritance, built-in functions, and other stuff.



This compilation can happen in two ways → Server-side, Client-side

LESS

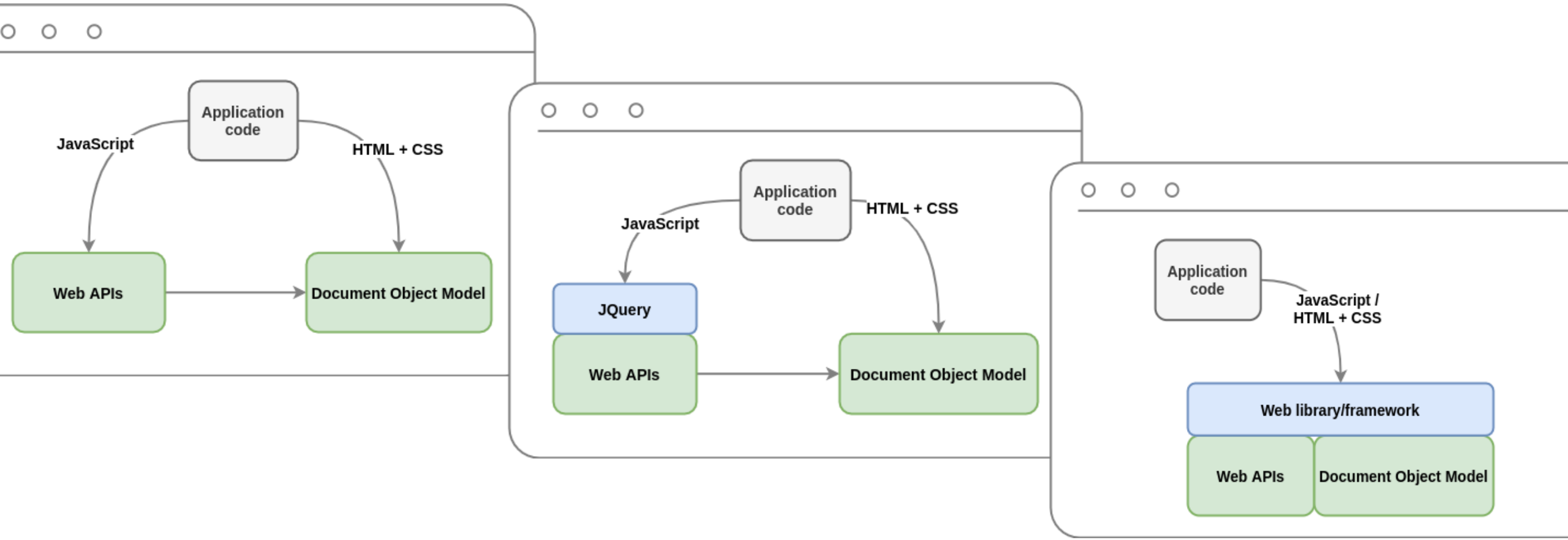
```less
@base-color: #333;
@link-color: blue;

body {
  font: 100% Helvetica, sans-serif;
  color: @base-color;
}

a {
  color: @link-color;
  &:hover { color: darken(@link-color, 10%); }
}
```

- LESS is a preprocessor that extends CSS with dynamic features. It is compiled into CSS either through a server-side script or in the browser.
- It simplifies managing CSS code with more powerful tools like variables, nesting, mixins, functions, and more.
- Use variables to store colors, fonts, or any CSS value you plan to reuse.
- LESS allows CSS rules to be nested within each other, which mimics visual hierarchy.
- Mixins allow you to group CSS declarations that you want to reuse
- Use functions to manipulate colors and other values

Feature/Aspect	Sass (Syntactically Awesome Stylesheet)	LESS
Full Name	Syntactically Awesome Stylesheet	Leaner Style Sheets
Year Introduced	2006 (by Hampton Catlin, developed by Natalie Weizenbaum)	2009 (by Alexis Sellier)
Syntax	Two syntaxes: - **SCSS** (Sassy CSS, CSS-like with `{}`) - **Sass** (indented, no braces/semicolons)	Single syntax, CSS-like with `@` for variables and `{}` for blocks
File Extension	`.scss` (SCSS) or `.sass` (indented Sass)	`.less`
Variables	Yes, uses `$` (e.g., `$color: #fff;`)	Yes, uses `@` (e.g., `@color: #fff;`)
Nesting	Supported (e.g., `nav { ul { list-style: none; } }`)	Supported (e.g., `nav { ul { list-style: none; } }`)
Mixins	Yes, with `@mixin` and `@include` (e.g., `@mixin border($r) { border-radius: $r; }`)	Yes, defined as classes or with parameters (e.g., `.border(@r) { border-radius: @r; }`)
Functions	Extensive built-in functions (e.g., `darken($color, 10%)`, `if()`, `map-`	Built-in functions (e.g., `lighten(@color, 10%)`, `ceil()`) but

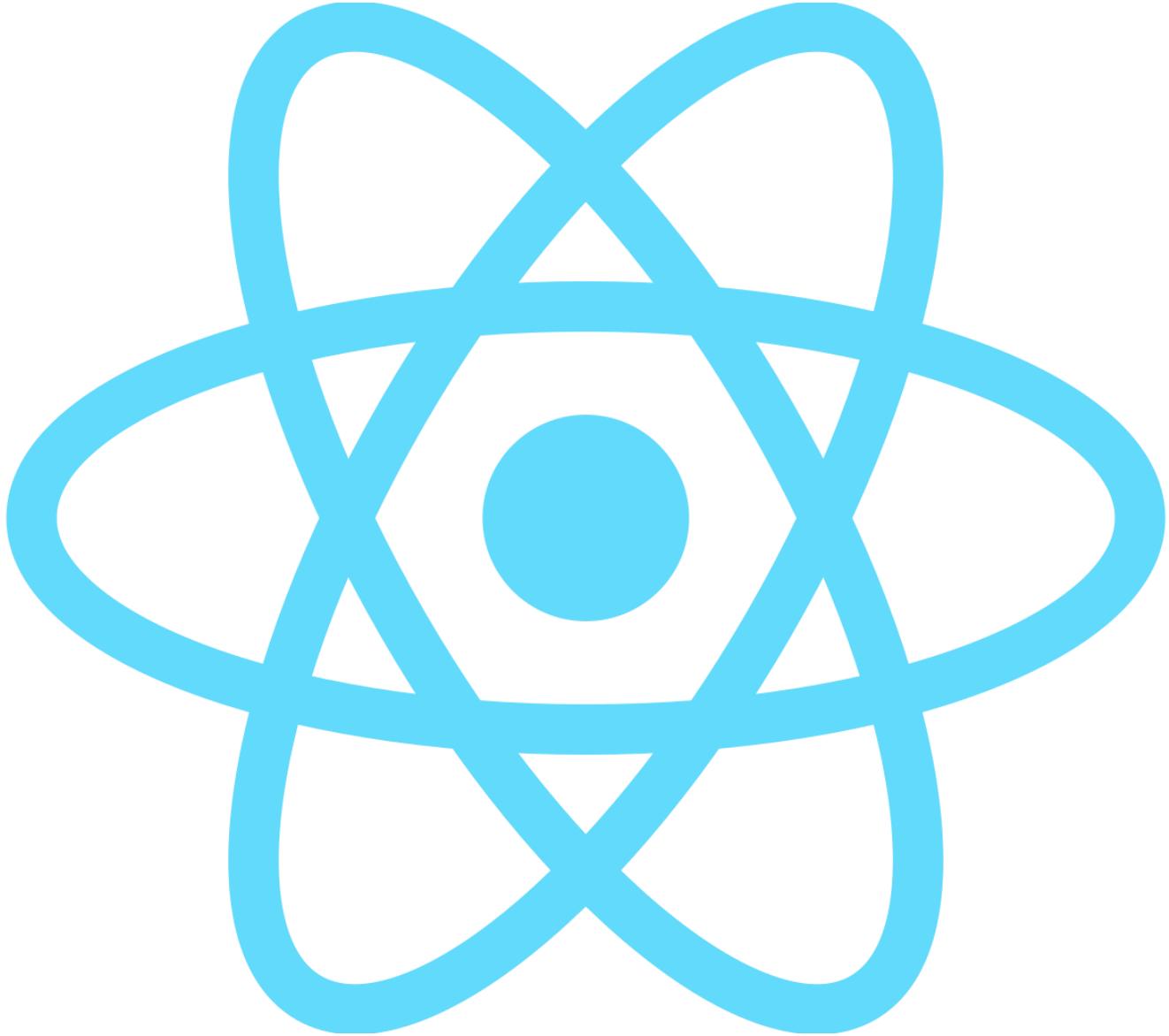
Similarities: Both Sass and LESS aim to make CSS more efficient with variables, nesting, and mixins, and they compile into standard CSS.
Differences: Sass offers more advanced features (e.g., loops, inheritance) and flexibility (two syntaxes), while LESS is simpler, lighter, and closer to CSS, with the unique ability to compile in-browser.

Evolution of Frontend Development



Source: https://www.epineda.net/the-evolution-of-front-end-development/amp/

ReactJS



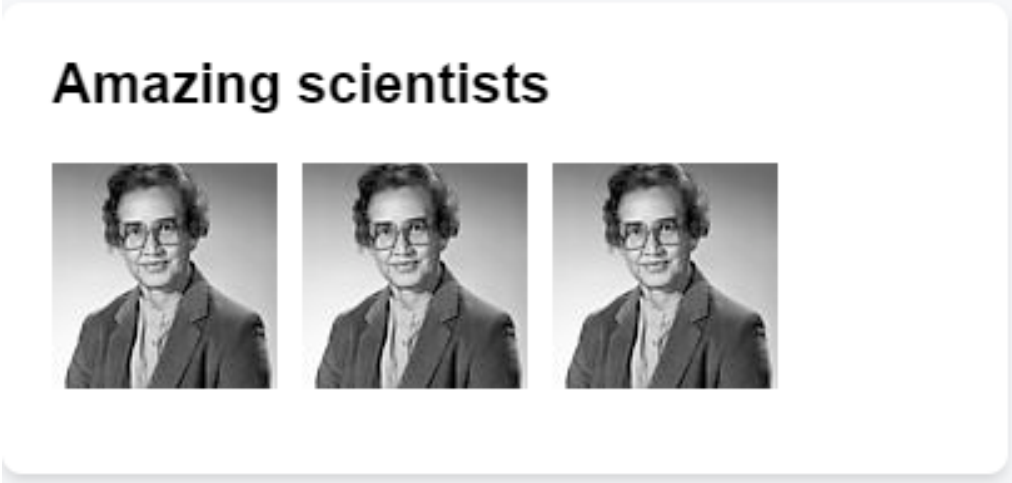
Introduction

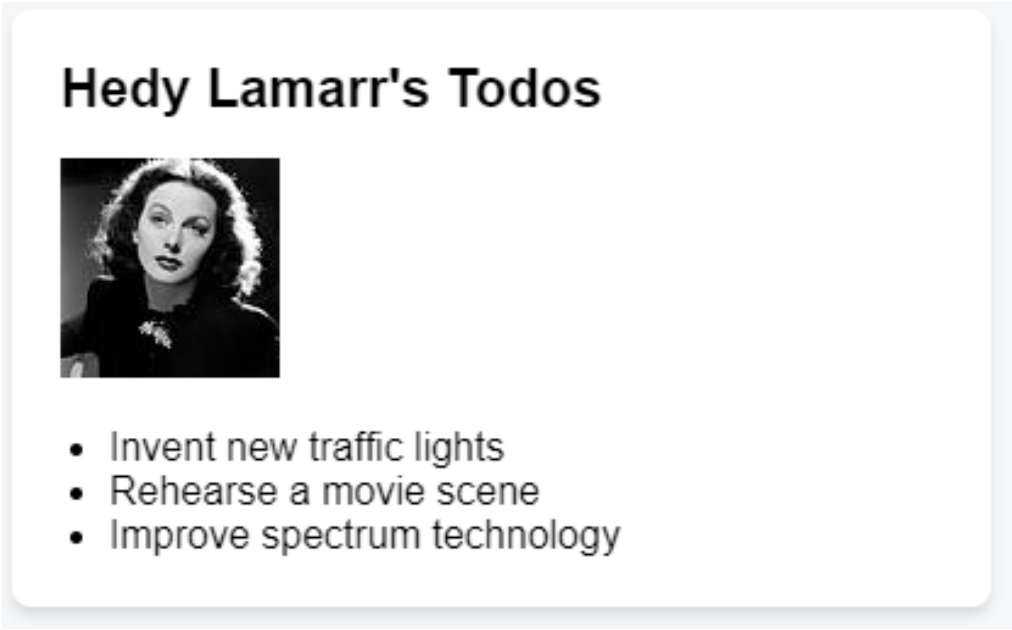
- Developed and maintained by Facebook and Instagram. meta
- A JavaScript library for creating user interfaces.
- Serves as the view of MVC architecture.
- Suitable for large web application which use data and change over the time without reloading the entire page.
- React Native caters developing mobile application for various platforms and React VR caters developing VR applications. Virtual Reality application
- Aims at Speed, Simplicity and Scalability.

Education: Virtual classrooms or simulations (exploring the human body in 3D).
Training: Flight simulators for pilots or medical procedure practice.

React : Components

- Components are one of the core concepts of React.
- They are the foundation upon which you build user interfaces (UI)
- UI is built from small units like buttons, text, and images.
- React lets you combine them into reusable, nestable components.
- From web sites to phone apps, everything on the screen can be broken down into components.





Notable features

One-Way data flow.

- Single source of truth - Data is originated and kept in one place, data is immutable.
- Data flow from the parent component to the child component.
- Action flow from child component to parent component.

Virtual DOM

- DOM manipulation is cost instead react create a virtual DOM tree and compare it with the rendered tree and update only what is changed.

The Problem: DOM Manipulation is Costly. React's Solution: The Virtual DOM

JSX

- React JS language for defining user interfaces.
- HTML/XML like syntax.
- Prevents cross-site scripting attacks by converting expressions to strings.

Props and State

read only

propertice

immutable - can't modify

- Props: Props are used to pass data from a parent component to a child component.

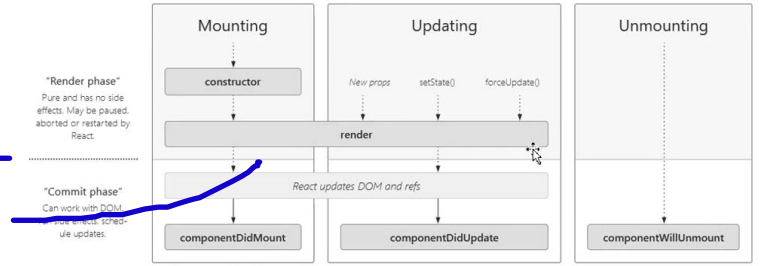
mutable - can modify

- State: State is used to manage data that changes over time in a component.

setState method is asynchronous so, if you want to use satet and props inside setState method use the second form of the method setState((state,props)=>{})

```jsx
function App() {
  const[count, setCount] = useState(0);

  const incremnet = () => {
    //count +=1;
    setCount(count+1);
  }
  const decremnet = () => {
    if(count > 0){
      setCount(count-1);
    }
  }

  return (
    <div className="App">

      {/* -------------------- props ------------------*/}
      <Topic number="1" color="white"/>
      <Topic number="2" color="red"/>
      <Topic number="3" color="green">
        <span>children propertice</span>
      </Topic>
      <Topic number="3" color="green">
        <button>this is button</button>
      </Topic>
      <hr/>

      {/* -------------------- useState ------------------*/}
      <span>My counter</span>
      <p>the count is {count}</p>
      <button onClick={decremnet}>-</button>
      <button onClick={incremnet}>+</button>


    </div>
  );
}
```

```jsx
import React from 'react'

const Topic = (props) => {
  return (
    <div>
      Topic {props.number} the color is {props.color} this is {props.children}
    </div>
  )
}

export default Topic
```



React Component Lifecycle

In React, components go through a lifecycle:

Mounting:
When a component is created and inserted into the DOM.
component first render to the UI

Updating:
When a component's state or props change, triggering a re-render
.
Unmounting:
When a component is removed from the DOM.

componentDidMount():

- This method is called immediately after a component is mounted in the DOM. It is often used for initializing the component state or fetching data from an API.

componentWillUnmount():

- This method is called immediately before a component is unmounted from the DOM. It is often used for cleaning up any resources or event listeners associated with the component.

shouldComponentUpdate(nextProps, nextState):

- This method is called before a component is re-rendered. It returns a boolean value indicating whether the component should update or not, based on the next props and state values.

getDerivedStateFromProps(props, state):

- This is a static method that is called before rendering a component, both on the initial mount and on subsequent updates. It returns an object to update the component state based on the new props.

getSnapshotBeforeUpdate(prevProps, prevState):

- This method is called right before the most recent render output is committed to the DOM. It allows you to capture information about the DOM before it is changed, such as the scroll position, and return it as an object.

componentDidUpdate(prevProps, prevState, snapshot):

- This method is called immediately after a component is updated and re-rendered. It is often used for updating the component state or making API calls based on the new props or state.

All the above ones can be represented from useEffect() hook in React Functional Components

Event Handlers

2. Defined as Methods or Arrow Functions

- **Class Components**: Event handlers are typically defined as class methods.
  - You often need to bind `this` in the constructor to ensure the method has access to the component's context.
- **Functional Components**: Event handlers are usually defined as arrow functions or regular functions within the component.
  - No binding is needed since functional components don't use `this`.

- In React, event handlers are functions that are called in response to user actions, such as clicks, keystrokes, or form submissions.
- React event handlers are typically defined as methods of a component class or as arrow functions within a functional component.
- In JSX, event handlers are specified as attributes of elements, using a naming convention where the event name is prefixed with on, followed by the name of the event in camelCase.

In JSX, event handlers are attributes on elements, following the pattern onEventName (e.g., onClick, onKeyDown).

Event handlers can receive an event object as a parameter, which contains information about the event, such as the target element, the mouse position, or the key that was pressed.

event.target: The element that triggered the event.
event.type: The event type (e.g., "click").
event.clientX/clientY: Mouse coordinates (for mouse events).
event.key: The key pressed (for keyboard events).

Class Component Example

jsx Collapse

```jsx
import React from "react";

class ButtonComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    // Bind the handler in the constructor
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick(event) {
    console.log("Clicked at:", event.clientX, event.clientY);
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  }
}

export default ButtonComponent;
```

Functional Component Example

jsx Collapse

```jsx
import React, { useState } from "react";

function InputComponent() {
  const [text, setText] = useState("");

  const handleChange = (event) => {
    console.log("Target:", event.target.value);
    setText(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault(); // Prevents page refresh
    console.log("Submitted:", text);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={text} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default InputComponent;
```

Hooks

UseState

```jsx
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0); // Initial state is 0

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

UseEffect

```jsx
import React, { useState, useEffect } from "react";

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("https://api.example.com/data")
      .then(res => res.json())
      .then(result => setData(result));

    return () => console.log("Cleanup"); // Runs on unmount or before next effect
  }, []); // Empty array = runs once on mount

  return <div>{data ? data.message : "Loading..."}</div>;
}
```

UseContext

```jsx
import React, { createContext, useState } from 'react';

// Create a Context
const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(theme === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

```jsx
import React from 'react';
import { ThemeContext } from './ThemeProvider';

const ThemedButton = () => {
  return (
    <ThemeContext.Consumer>
      {({ theme, toggleTheme }) => (
        <button
          onClick={toggleTheme}
          style={{
            backgroundColor: theme === 'light' ? '#fff' : '#333',
            color: theme === 'light' ? '#000' : '#fff',
          }}
        >
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
```

- React Hooks are a way to use state and other React features in functional components, without the need for class components.
- The most common Hooks are useState, useEffect, useContext, and useReducer. Each Hook has a specific purpose and usage.
- The useState Hook allows you to add state to a functional component. It returns an array with two values: the current state value and a function to update the state.

run every time component will update. if not, not run, avoid every time running problem

- The useEffect Hook allows you to perform side effects in a functional component, such as fetching data from an API or updating the DOM. It takes a function as its argument and runs it after every render.

replacing lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

- The useContext Hook allows you to access data from a React context in a functional component. It takes a context object as its argument and returns the current context value.

avoid passing props via multiple levels

inspired by Redux.

- The useReducer Hook is an alternative to useState that allows you to manage more complex state updates. It takes a reducer function and an initial state value as its arguments and returns the current state value and a dispatch function to update the state.
- Hooks should only be used at the top level of a functional component or another custom Hook. They should not be used inside loops or conditionals.

Managing state | Reacting to inputs

- As your application grows, it helps to be more intentional about how your state is organized and how the data flows between your components.
- Redundant or duplicate state is a common source of bugs.
- React uses a declarative way to manipulate the UI.
- Instead of manipulating individual pieces of the UI directly, you describe the different states that your component can be in, and switch between them in response to the user input.
- This is similar to how designers think about the UI.

Redundant State: Storing the same data in multiple places, causing inconsistencies.
Out-of-Sync Updates: When related pieces of state don't update together.

Managing state | State Structure

A well-structured state reduces bugs, simplifies debugging, and makes components easier to modify.
Poor structure can lead to confusion, contradictions, or unnecessary complexity.

- Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs.

Best Practices for Structuring State

- **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.

```
const [name, setName] = useState({ first: "", last: "" });
const updateName = (f, l) => setName({ first: f, last: l });
```

- **Avoid contradictions in state.** When the state is structured in a way that several pieces of state may contradict and "disagree" with each other, you leave room for mistakes. Try to avoid this.

Contradictory state (e.g., isLoading: true and data: "loaded")

- **Avoid redundant state.** If you can calculate some information from the component's props or its existing state variables during rendering, you should not put that information into that component's state.

```
const [first, setFirst] = useState("John");
const [last, setLast] = useState("Doe");
const [fullName, setFullName] = useState("John Doe");
// Redundant
```

- **Avoid duplication in state.** When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.

```
const [userId, setUserId] = useState(1);
const [userData, setUserData] = useState({ id: 1, name: "John" }); // Duplicates userId
```

- **Avoid deeply nested state.** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

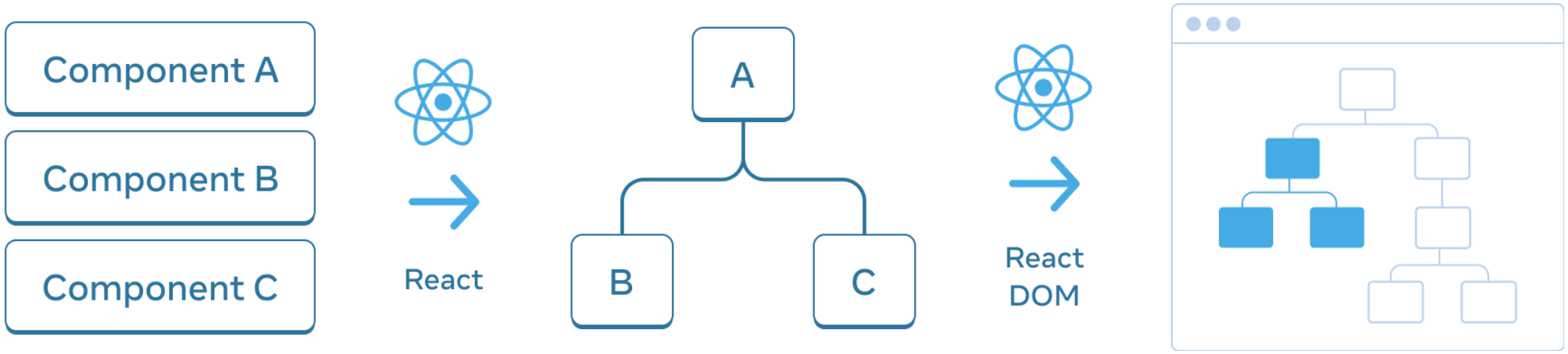
```
const [data, setData] = useState({
  user: { profile: { name: "John" } },
});

// Updating name requires spreading multiple levels
setData({ ...data, user: { ...data.user, profile: { ...data.user.profile, name: "Jane" } } });
```

Managing state | State Sharing

(e.g., a toggle affecting multiple UI parts).

- Sometimes, you want the state of two components to always change together.
- Remove state from both and move it to the closest parent component
- Then pass it down to them via props
- This is known as ***lifting state up***, and it's one of the most common things you will do writing React code.

Move the state to the closest common parent component and pass it down via props.



React associates state with a component based on its position in the UI tree (the component hierarchy).

This is tied to React's reconciliation process, where it matches components across renders using their place in the tree and their keys (if provided).

Why It Matters: Isolation ensures components are independent and reusable, but it also means you need to manage when state persists or resets.

Managing state | Preserving & Resetting State

- State is isolated between components.
- React keeps track of which state belongs to which component based on their place in the UI tree.
- You can control when to preserve state and when to reset it between re-renders.

Problem: Components with many state updates across multiple event handlers (e.g., onClick, onChange) can become cluttered and hard to maintain.

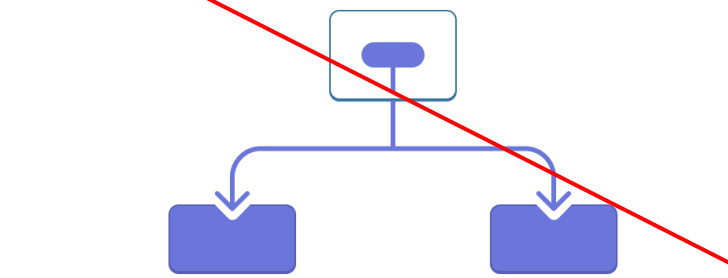
Solution: Move state update logic into a single reducer function outside the component, consolidating all state transitions in one place.

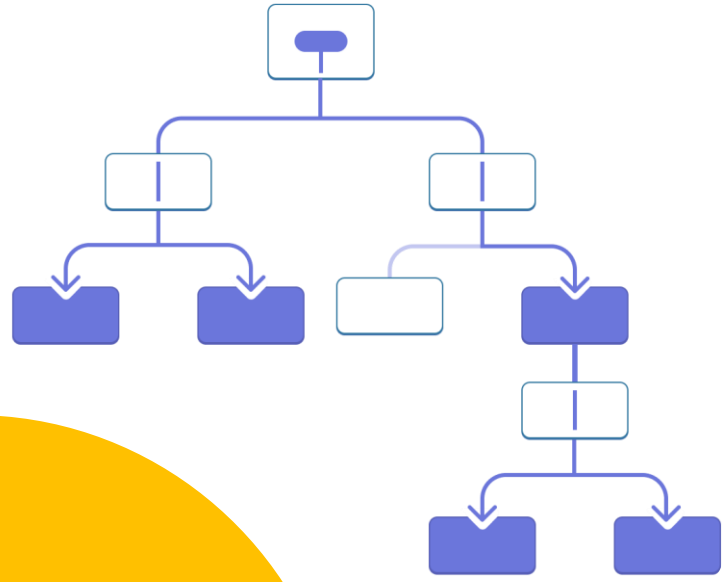
Managing state | Using Reducer

- Components with many state updates spread across many event handlers can get overwhelming
- For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer.
- Although reducers can "reduce" the amount of code inside your component, they are named after the reduce() operation that you can perform on arrays.
- We will discuss about this in the tutorial.

```
const initialState = { name: "", age: 0, isValid: false };
function reducer(state, action) {
  switch (action.type) {
    case "setName":
      return { ...state, name: action.value, isValid: action.value.length > 2 && state.age > 0 };
    case "setAge":
      return { ...state, age: action.value, isValid: state.name.length > 2 && action.value > 0 };
    default:
      return state;
  }
```

Problem with Props
Prop Drilling: Passing state through many components (e.g., Parent > Child > Grandchild) is verbose and inconvenient.





Managing state | with Context

- Usually, you will pass information from a parent component to a child component via props.
- But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information.
- Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

Context lets a parent component provide data to any component in its subtree, bypassing prop drilling.

Babel

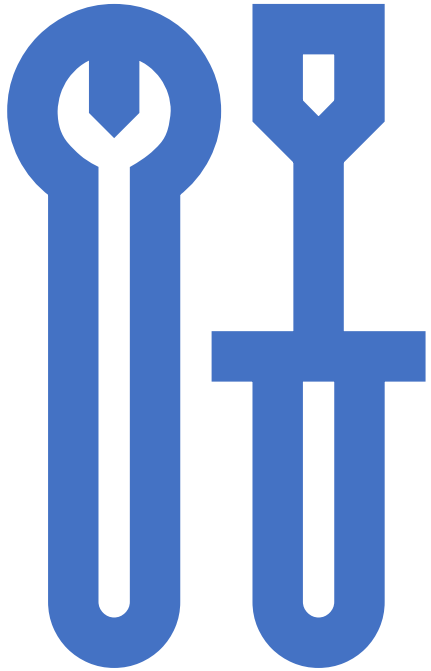
Babel is a JavaScript compiler that transforms modern JavaScript (and related languages like JSX or TypeScript) into code compatible with older browsers or environments.
Purpose: Allows developers to use cutting-edge JavaScript features without worrying about browser support.

js transpiler

- Babel is a JavaScript compiler that allows you to write modern JavaScript code and transform it into code that can run in older browsers or environments.
- Babel can transform modern JavaScript features like arrow functions, template literals, and de-structuring into equivalent code that is compatible with older browsers.
- Babel uses plugins to transform specific features of JavaScript. There are many plugins available for Babel, and you can choose which ones to use based on your needs.
- Babel can also transform code written in other languages that compile to JavaScript, like TypeScript and JSX.
- Babel is open-source software and is maintained by a community of developers. It is a widely used tool in the JavaScript ecosystem and is supported by many popular libraries and frameworks.
- Babel supports many advanced features of JavaScript, such as async/await, class properties, and decorators, allowing you to use the latest language features even in older environments.

Bundling

- In web development, bundling refers to combining multiple files into a single file.
- Bundling is typically done with JavaScript or CSS files.
- Bundling reduces the number of HTTP requests needed to fetch all the required resources for a web page.
- This can improve the page's performance by reducing network latency.
- Bundling tools like Webpack or Parcel are used to create the bundle file.
- These tools analyze dependencies between files to create the bundle.
- Optimization techniques like minification and tree shaking may also be used as part of the bundling process.



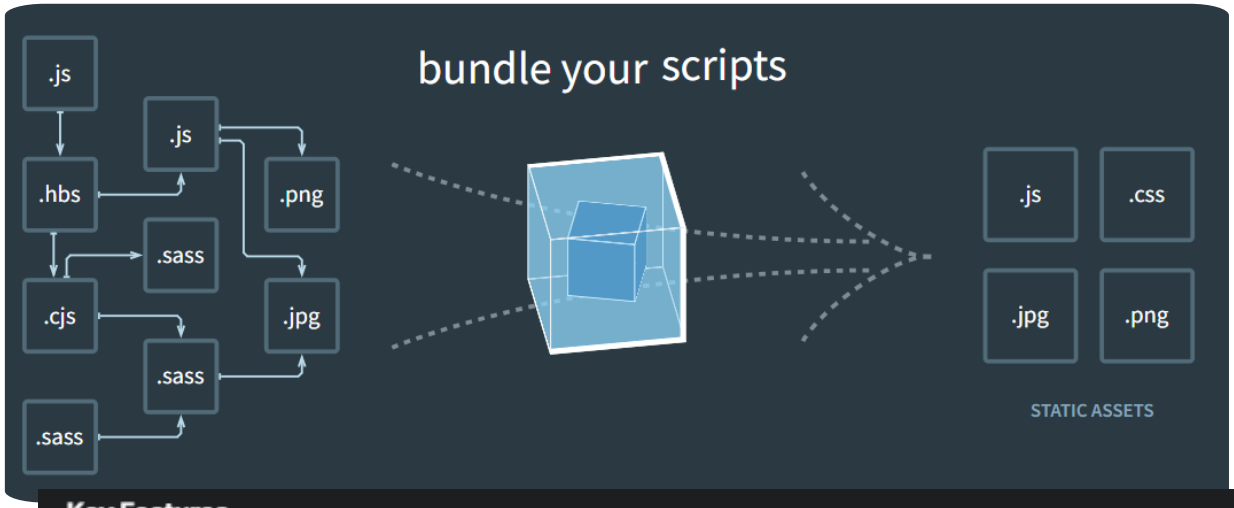
How It Works

Process: A bundler analyzes your project's dependency graph (e.g., import statements) and merges all files into one (or a few) bundles.
Example: 10 JS files → 1 bundle.js.
Tools: Webpack, Parcel, Rollup, etc.

Definition: An open-source module bundler for web development.
Purpose: Bundles JavaScript and assets (CSS, images, fonts) into optimized files.

Webpack

- Webpack is an open-source bundling tool and module bundler used in web development.
- It is used to bundle JavaScript files and other assets like CSS, images, and fonts for web applications.
- Webpack analyzes the dependencies between modules in the application and generates a single optimized bundle file.
- It includes a built-in development server that allows developers to preview their applications in a browser and see live changes.
- Webpack offers features like code splitting, lazy loading, tree shaking, and hot module replacement to optimize and speed up performance.
- It is highly configurable and can be customized according to project requirements.
- Webpack is widely used in modern web development frameworks like React, Vue, and Angular.
- https://webpack.js.org/



Key Features

- **Dependency Analysis:** Uses a `module` system (e.g., ES6 `import`, CommonJS `require`) to build a dependency graph.
- **Output:** Generates a single `bundle.js` (or multiple with code splitting).
- **Development Server:** `webpack-dev-server` offers live reloading and previews.
- **Advanced Features:**
  - **Code Splitting:** Splits code into chunks loaded on demand (e.g., lazy-loaded routes).
  - **Lazy Loading:** Loads modules only when needed.
  - **Tree Shaking:** Removes unused exports.
  - **Hot Module Replacement (HMR):** Updates modules in real-time without full page reloads.

Parcel

Definition: A zero-config bundler, simpler alternative to Webpack.
Purpose: Bundles JavaScript, CSS, HTML, images, etc., with minimal setup.

- Parcel is an open-source web application bundler and build tool, similar to Webpack.
- It is designed to be zero-config, meaning that it requires minimal setup and configuration to get started.
- Parcel can bundle a variety of web assets, including JavaScript, CSS, HTML, images, and more.
- It automatically handles file transformations, such as transpiling ES6 code to ES5 or compressing images.
- Parcel can use multiple cores to build and bundle projects in parallel, which can reduce build times.
- It includes a built-in development server that allows developers to see their changes in real-time as they make modifications to their code.
- Parcel has gained popularity among developers due to its ease of use and simplicity.
- It has been adopted by popular frameworks such as React, Vue, and Angular.
- https://parceljs.org/

dev dependancy

Key Features

- **Zero-Config**: Works out of the box (no `webpack.config.js` needed initially).
- **Transformations**: Auto-transpiles ES6 to ES5 (via Babel), compresses assets.
- **Parallel Builds**: Uses multiple CPU cores for faster builds.
- **Dev Server**: Real-time updates with HMR.
- **Ease of Use**: Popular for its simplicity in React, Vue projects.

Example

- Just run `parcel index.html`, and it bundles everything automatically.

Setting Up React

Why: Raw React lacks built-in solutions for routing, data fetching, and server-side rendering (SSR), which most apps need.
Frameworks: Add these features, streamlining development.

- Official Documentation recommend to use a framework for react
- Frameworks provide features that most apps and sites eventually need, including routing, data fetching, and generating HTML.
- Popular Frameworks
  - Next.js
  - Remix
  - Gatsby
  - Expo (for native apps)

Popular Frameworks

1. **Next.js**:
   - Features: SSR, static site generation (SSG), file-based routing, API routes.
   - Use Case: Full-stack React apps.
2. **Remix**:
   - Features: Nested routing, data loading, error boundaries.
   - Use Case: Dynamic, data-driven apps.
3. **Gatsby**:
   - Features: SSG, GraphQL data layer, plugin ecosystem.
   - Use Case: Static sites, blogs.
4. **Expo**:
   - Features: React Native tooling, cross-platform mobile dev.
   - Use Case: Native mobile apps.

Definition: A fast build tool and dev server by Evan You (Vue.js creator).
Purpose: Speeds up development and bundling for modern web apps

Vite

- Vite is a build tool and development server for modern web applications.
- It was created by Evan You, the creator of Vue.js.
- Vite is known for its fast startup time.
- It uses a development server that leverages modern browser features like native ES modules and HTTP/2 to quickly serve application code.
- Vite provides a build tool that includes features like code splitting and tree shaking to optimize the final application bundle for deployment.
- Vite is primarily used for developing front-end applications built with frameworks like React or Vue.js.

Webpack vs Parcel vs Vite

eliminates unused code from your JavaScript bundles

	Webpack	Parcel	Vite
Developer Experience	Moderate	Easy	Very Easy
Configuration	Highly configurable but complex	Minimal configuration	Minimal configuration
Build Time	Slow for large projects	Fast for small to medium-sized projects	Very fast for small to medium-sized projects
Hot Reloading	Built-in but slow	Built-in and fast	Built-in and very fast
Tree-Shaking	Yes	Yes	Yes
Code Splitting	Yes	Yes	Yes
Plugin Ecosystem	Large and mature	Small but growing	Small but growing
Community Support	Large and active	Active but smaller than Webpack	Active but smaller than Webpack
Popularity	Very popular and widely used	Popular but less widely used than Webpack	Newer but gaining popularity