**NOTE ::**

**THOUGH MOST Of THE STUFF IS COPIED FROM THE ARTICLES AND CHATGPT I WAS CURIOUS ABOUT THESE QUESTIONS WHILE LEARNING REACT SO IF THERE ARE ANY KNOWLEDGE GAPS**

**OR SOME DETAILS MISSING OR SOME SHITTY STUFF YEET IT OUT AND FEEL FREE TO MAKE CHANGES TO THIS** 😉

**AND YEAH JUST A REMINDER IF YOU MAKE ANY CHANGES IN THE FORMATTING AND SPELLING MISTAKES ADDING EXTRA SPACES etc AND CONSIDERING THAT AS AN CONTRIBUTION .**

**WELL BRO THEN .**

****

**Thought you wanna know**

The Rendering Process

The rendering process begins when you open a web page in your browser. This process involves several steps that work together to transform the page's code into a visual display on your screen. The steps involved are:

**Parsing HTML**

**Parsing CSS**

**Constructing the Rendering Tree**

**Layout**

**Painting**

**Compositing**

Now that we’ve set the stage let’s take a closer look at each step of the rendering process. While the process can vary slightly depending on the browser and device used, the fundamental steps remain the same.

**1. Parsing HTML**

Parsing HTML is the first step in the rendering process. When the browser receives an HTML document, it parses the HTML code to create a tree-like structure called the Document Object Model (DOM). The DOM represents the web page's structure and includes all of the elements and attributes defined in the HTML code.

**2. Parsing CSS**

The second step in the rendering process is parsing CSS. The browser takes the CSS code associated with the page and creates a separate tree-like structure known as the CSS Object Model (CSSOM). The CSSOM contains all of the style rules defined in the CSS code.

**3. Constructing the Rendering Tree**

After the browser has created the DOM and CSSOM, it combines the two to create the Rendering Tree. The Rendering Tree combines the DOM and CSSOM, representing the elements on the page and their associated styles.

**4. Layout**

The Layout step is where the browser calculates the position and size of each element in the Rendering Tree. This is based on the styles defined in the CSS code and the content of the DOM. The Layout step is important for determining the placement of each element on the screen and ensuring that the page displays correctly.

**5. Painting**

Once the Layout has been calculated, the browser moves on to the Painting step. Painting involves filling in each element with color and creating an image of the page to be displayed on the screen. The browser uses the styles and layout information to paint each element in the Rendering Tree.

**6. Compositing**

Finally, in the Compositing step, the browser combines all of the painted elements into a final image. This image is then displayed on the screen, completing the rendering process.

Understanding the rendering process is important for web developers, as it can help identify performance issues and optimize the display of web pages. In the following sections, we’ll explore each step in more detail and discuss common rendering issues and optimization strategies.

**Parsing HTML**

Parsing is analyzing a piece of code to determine its structure and meaning. In the case of HTML, parsing involves analyzing the HTML code to create a structured representation of the content and elements on the page.

When the browser receives an HTML document, it begins by parsing the code. The browser starts at the top of the HTML code and reads through each line, looking for HTML tags, attributes, and content. As it reads through the code, it builds a tree-like structure called the Document Object Model (DOM).

Common issues with HTML parsing

One common issue with HTML parsing is invalid HTML code. If the HTML code is not well-formed or contains errors, the browser may be unable to create a complete DOM tree. This can result in missing or misplaced elements on the page or errors in the display of the page. Another issue is the presence of JavaScript, which can modify the content and structure of the page after it has been parsed.

Optimization strategies

To optimize HTML parsing, writing well-formed and valid HTML code is essential. This means using correct syntax and adequately nested tags. It’s also essential to keep the HTML code as simple as possible, avoiding unnecessary tags or attributes. In addition, it can be helpful to use tools such as HTML validators to check for errors in the code.

**Parsing CSS**

CSS parsing is the process of analyzing the CSS code associated with a web page to determine the styles that should be applied to the elements on the page. CSS parsing aims to create a structured representation of the style rules defined in the CSS code.

When the browser receives a web page, it parses the HTML and CSS code. The browser starts by creating the Document Object Model (DOM) tree, as discussed in the previous section. Next, the browser creates a separate tree-like structure, the CSS Object Model (CSSOM), representing the style rules defined in the CSS code.

The CSSOM is a tree-like structure representing the style rules defined in the CSS code. Each CSS rule is represented by a node in the CSSOM tree, with properties such as selector, declaration block, and value.

Common issues with CSS parsing

One common issue with CSS parsing is the presence of invalid CSS code. If the CSS code is not well-formed or contains errors, the browser may be unable to create a complete CSSOM tree. This can result in missing or incorrect styles on the page or errors in the display of the page. Another issue is using complex or inefficient selectors, which can slow down the parsing and rendering process.

Optimization strategies

To optimize CSS parsing, writing well-formed and valid CSS code is essential. This means using correct syntax and avoiding common errors such as missing or mismatched braces. Keeping the CSS code as simple as possible is essential, avoiding complex selectors or unnecessary declarations. In addition, it can be helpful to use tools such as CSS validators or pre-processors to optimize the CSS code.

Constructing the Rendering Tree

The rendering tree is a hierarchical representation of the content and styles on the web page. It’s created by combining the Document Object Model (DOM) tree with the CSS Object Model (CSSOM) tree to determine the final layout and appearance of the page.

To construct the rendering tree, the browser matches each element in the DOM tree with the corresponding style rules in the CSSOM tree. It then applies the styles to each element, computing each element's final layout and position on the page. The result is a hierarchical structure known as the rendering tree, which represents the final appearance of the page.

The rendering tree combines the content and styles from the DOM and CSSOM trees. Each node in the rendering tree represents a visible element on the page, with properties such as size, position, and style. The rendering tree also includes non-visual elements such as script and meta tags, which are not displayed on the page but can affect its behavior.

Common issues with rendering tree construction

One common issue with rendering tree construction is the presence of conflicting styles. The browser must determine which rule takes precedence if multiple style rules apply to the same element. This can lead to unexpected behavior or an incorrect layout on the page. Another issue is using complex or inefficient CSS selectors, which can slow the rendering process and affect page performance.

Optimization strategies

To optimize rendering tree construction, it’s important to write efficient and well-structured CSS code. This means avoiding redundant or conflicting styles and using simple and efficient selectors. It’s also essential to keep the HTML code simple, avoiding unnecessary elements or attributes. In addition, it can be helpful to use tools such as CSS pre-processors or layout frameworks to optimize the rendering process.

**Painting and Compositing**

Painting is the process of filling in the content and styles of the elements in the rendering tree to create the final visual appearance of the page. This involves applying the computed styles to each element, including background colors and images, borders, and text.

Once the rendering tree has been constructed, the browser performs the painting process in two stages: layout and paint. During the layout stage, the browser computes the size and position of each element on the page, based on the content and styles in the rendering tree. During the paint stage, the browser fills in the content and styles of each element using the computed values from the layout stage.

Compositing is the process of combining the painted elements into the final visual representation of the page. This involves stacking the elements correctly and applying transparency and blending effects as necessary.

**Common issues with painting and compositing**

One common issue with painting and compositing is complex or inefficient CSS styles, which can slow the rendering process and affect page performance. Another issue is using large or uncompressed images, which can take a long time to load and display on the page.

**Optimization strategies**

To optimize painting and compositing, it’s essential to use efficient and well-structured CSS styles and to avoid unnecessary or redundant elements. It’s also important to use optimized images and other media, such as compressed or cached files. In addition, it can be helpful to use browser dev tools or performance profiling tools to identify and fix any issues with the painting and compositing process.

**Conclusion**

The rendering process is complex and dynamic, determining how web pages are displayed in the browser.

The four steps of the rendering process are: parsing and tokenization, constructing the DOM tree, constructing the rendering tree, and painting and compositing.

Each step of the rendering process has its own challenges and optimization strategies, and it’s essential for web developers to understand how the process works to create efficient and responsive web pages.

The rendering process is not a one-time event, and changes to the page content or styles can trigger a new rendering process.

By following best practices and using tools and techniques to optimize each step of the rendering process, web developers can create web pages that load quickly and deliver a great user experience to their audience.

Remember, understanding the rendering process is just one aspect of creating great web pages. It’s also important to consider other factors such as accessibility, usability, and performance to ensure your web pages are accessible and usable to all users.

**WHAT IN THE WORLD IS REACT**

**AND WHY TO USE IT IN YOUR SHITTY PROJECT**

React is a popular JavaScript library developed by Facebook for building user interfaces, particularly single-page applications. It allows developers to create large web applications that can change data, without reloading the page. Here are its key features and advantages:

**Key Features of React**

1. **Component-Based Architecture**: React applications are built using components, which are reusable pieces of UI. This modular approach makes it easier to manage and scale applications.
2. **JSX**: React uses JSX, a syntax extension that allows you to write HTML-like code within JavaScript. This makes it easy to create and visualize UI components.
3. **Virtual DOM**: React maintains a virtual representation of the UI in memory, which it uses to optimize updates. When the state of a component changes, React updates the virtual DOM, compares it with the previous version, and efficiently updates the real DOM only where necessary.
4. **Unidirectional Data Flow**: React uses a unidirectional data flow, meaning data flows from parent components to child components. This makes it easier to understand and debug applications.

**Advantages of React**

1. **Performance**: The use of a virtual DOM and efficient diffing algorithms means React can update the UI quickly without the need for a full page reload.
2. **Reusability**: Components can be reused throughout the application, which reduces duplication and makes the code more maintainable.
3. **Developer Tools**: React comes with a set of developer tools that help with debugging and optimizing applications. The React Developer Tools extension for browsers is particularly useful.
4. **Strong Community and Ecosystem**: React has a large and active community, which means there are plenty of resources, libraries, and tools available to help developers.
5. **SEO-Friendly**: React can be rendered on the server side, which can improve the SEO of applications. Libraries like Next.js are built on top of React to provide server-side rendering out of the box.
6. **Easy to Learn**: For developers familiar with JavaScript, React is relatively easy to learn. Its component-based structure and straightforward API make it accessible even for beginners.
7. **Integration with Other Frameworks**: React can be easily integrated with other libraries or frameworks, such as Redux for state management or React Router for navigation.

In summary, React's component-based architecture, performance optimizations, and strong community support make it a powerful tool for building modern web applications.

**What is a Build Process in React Anyway**

The build process in React involves converting your React code into a format that web browsers can understand and efficiently run. Here’s a simple breakdown of what happens during the build process:

1. **Transpiling**: Your modern JavaScript (ES6+ and JSX) is converted into older JavaScript (ES5) that all browsers can understand. This is done using a tool called Babel.
2. **Bundling**: All your JavaScript files, along with their dependencies, are combined into a single file (or a few files). This is handled by a tool called Webpack or similar.
3. **Minification**: The combined JavaScript file is then minimized by removing unnecessary spaces, comments, and shortening variable names. This helps the code load faster in the browser.
4. **Optimization**: Additional steps, like splitting the code into smaller chunks (code splitting), and optimizing images and other assets, ensure your app performs well.

When you run a command like npm run build, all these steps are executed, resulting in a set of files that are ready to be deployed to a web server.

### What is Vite?

Vite is a tool that helps you set up and develop web applications quickly. It is particularly known for its speed and simplicity. It was created by Evan You, the same person who created Vue.js.

### Key Features of Vite

1. **Instant Start**: Vite uses native ES modules in the browser, which means it doesn't need to bundle your code before serving it. This results in an almost instant startup time.
2. **Fast Hot Module Replacement (HMR)**: When you make changes to your code, Vite updates only the parts of the code that changed, without needing to refresh the entire page. This makes development much faster and smoother.
3. **Optimized Build**: **For production**, Vite uses Rollup, a powerful bundler, to create optimized and minified bundles of your code.
4. **Support for Modern JavaScript**: Vite works out of the box with modern JavaScript features and also supports TypeScript, JSX, and CSS modules.

### How Vite Works

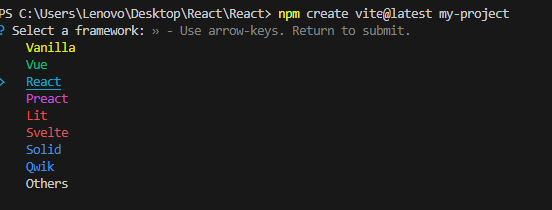
1. **Development Mode**: During development, Vite serves your source files over a local server. It uses the browser's native support for ES modules to load and run your code. This eliminates the need for a bundling step and allows for fast updates and hot module replacement.
2. **Build Mode**: When you're ready to deploy your application, Vite bundles your code using Rollup. This step optimizes your code, ensuring it loads quickly and efficiently in production.

### Advantages of Vite

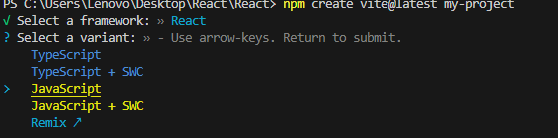
1. **Speed**: Vite is designed to be fast. It reduces startup time and provides a smooth development experience with fast updates.
2. **Simplicity**: Setting up a project with Vite is straightforward. It comes with sensible defaults and minimal configuration.
3. **Modern Features**: Vite supports the latest web standards and JavaScript features, making it easy to build modern web applications.

### Example Usage

To create a new project with Vite, you can run the following commands:



* Select react if you want to create a react project



* The language you want to use

Then Run these commands

* cd my-project
* npm install
* npm run dev

This will set up a new Vite project, install the necessary dependencies, and start a development server.

In summary, Vite is a powerful tool that makes web development faster and simpler by leveraging modern browser capabilities and efficient bundling for production

**Structure of a react project**

A typical React project has a well-organized structure to keep code manageable and understandable. Here’s a simple breakdown of a common React project structure:

### Root Directory

* **node\_modules/**: This folder contains all the project dependencies installed via npm (Node Package Manager). You usually don't interact with this directly.
* **public/**: This folder contains the static files of your project, such as the HTML file, images, and other assets. The main file here is:
  + **index.html**: The entry point for your React application. This file contains a <div> with an id="root" where your React app will be rendered.
* **src/**: This is where your actual React code lives. This folder typically includes:
  + **index.js**: The main JavaScript file that starts your React application. It renders the root component (usually called App) into the index.html file.
  + **App.js**: The root component of your application. It serves as the main layout and can contain other components.
  + **App.css**: The main CSS file for styling your App component.

### Inside src/

To keep things organized, you might have several subfolders:

* **components/**: This folder contains all the reusable components of your application. For example:
  + **Header.js**: A component for the header section.
  + **Footer.js**: A component for the footer section.
  + **Button.js**: A reusable button component.
* **assets/**: This folder contains static assets like images, fonts, and icons.
* **styles/**: This folder contains global styles and CSS files.
* **utils/**: This folder contains utility functions and helper files.
* **services/**: This folder contains files related to API calls or any other external services.

### Example Structure

Here’s an example structure of a simple React project:

css

Copy code

my-react-app/

├── node\_modules/

├── public/

│ ├── index.html

│ ├── favicon.ico

│ └── other-assets/

├── src/

│ ├── components/

│ │ ├── Header.js

│ │ ├── Footer.js

│ │ └── Button.js

│ ├── assets/

│ │ ├── logo.png

│ │ └── icons/

│ ├── styles/

│ │ ├── App.css

│ │ └── index.css

│ ├── utils/

│ │ └── helpers.js

│ ├── services/

│ │ └── api.js

│ ├── App.js

│ ├── index.js

│ └── index.css

├── .gitignore

├── package.json

└── README.md

### Key Files Explained

* **index.js**: The entry point for your React application. It renders the App component into the root DOM node in index.html.
* **App.js**: The main component that serves as the top-level layout of your application.
* **index.html**: The main HTML file that loads your React app.
* **.gitignore**: A file specifying which files and directories should be ignored by Git.
* **package.json**: A file that contains metadata about your project and lists the dependencies.

This structure helps keep your code organized, making it easier to manage and scale your application as it grows.

**WHAT ARE COMPONENTS**

In React, a **component** is a building block for creating user interfaces. Think of it as a piece of a webpage, like a button, a header, or a form. There are two main types of components:

1. **Functional Components**: These are simple JavaScript functions that return HTML elements. They are easy to write and understand. Here’s a basic example:

jsx

Copy code

function Greeting() {

return <h1>Hello, world!</h1>;

}

1. **Class Components**: These are more complex and are defined using ES6 classes. They can hold and manage their own state (data). Here’s a basic example:

jsx

Copy code

class Greeting extends React.Component {

render() {

return <h1>Hello, world!</h1>;

}

}

Components can also accept **props** (short for properties), which are like parameters that you pass to functions. Props allow you to pass data from one component to another. For example:

jsx

Copy code

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

// Using the component with a prop

<Greeting name="Alice" />

In this example, Greeting is a component that receives a name prop and uses it to display a personalized message.

By combining these components, you can build complex user interfaces that are easy to manage and reuse.

**What I understand after learning components is that it’s just a freaking function that returns a jsx type of thingy and since this is a function this can be utilized again and again like you excuses.**

**Always remember that the components name are written in pascal case i..e “MyCumponent” and they should return a renderable value like <p> small pp </p>.**

while inserting image do not use the complete path but import it from the assets by using this the image might not get lost during build process

|  |  |
| --- | --- |
| **This bad 😡**  function Pics()  {  return (  <div>  <p><img src=”**src**/**your\_childhood\_photos/small\_pp.png”**></p>  </div>  ) |  |
| **This good 😇**  **Import PP from ‘src/you\_childhood\_photos/small\_pp.png’;**  function Pics()  {  return (  <div>  <p><img src=**{ pp }**></p>  </div>  ) |  |

**“ Who said that smug face ? you may say, well the guy who’s video lectures I am following ,suckers 😅 ”**

**Each component has a file named after it in the components folder Ex: “Fyou.jsx”**

The components have a special prop known as **children**  **that will contain the text between the tags Example :**

**<Your\_tag> something </Your\_tag>.**

**How to add event listener to your component ?**

function YourTag({children})

{

return

(

<li>

<button onClick={function}>{children}</button>

</li>

);

}