Assignment 03 - Laying the foundation

1. **What is JSX?**

   JSX stands for JavaScript XML.
   JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.

   - **Code: (with JSX)**

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const myElement = <h1>I Love JSX!</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

   Output:

   I Love JSX!

   - **Code: (without JSX)**

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

   Output:

   I do not use JSX!

   As you can see in the first example, JSX allows us to write HTML directly within the JavaScript code. **Babel** is a JavaScript compiler that converts modern JavaScript code into a code that browser can understand.

2. **Superpowers of JSX**

   JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

The Web has been built on HTML, CSS, and JavaScript. For many years, web developers kept content in HTML, design in CSS, and logic in JavaScript—often in separate files! Content was marked up inside HTML while the page's logic lived separately in JavaScript:

But as the Web became more interactive, logic increasingly determined content. JavaScript was in charge of the HTML! This is why in React, rendering logic and markup live together in the same place—components.

Using curly braces: A window into the JavaScript world

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example:

```
export default function Avatar() {
 const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
 const description = 'Gregorio Y. Zara';
 return (
  <img
   className="avatar"
   src={avatar}
   alt={description}
  />
 );
}
```

3. **Role of type attribute in script tag? What options can I use there?**

The **type** attribute in the <script> tag is used to specify the scripting language of the embedded code. It helps the browser understand how to interpret and execute the script. While in modern web development, specifying the type attribute is not always necessary for JavaScript, it is still considered good practice.

Syntax:

```
<script type="media_type">
```

Attribute Values: It contains a single value i.e media_type which specifies the MIME type of script.

Common "media_type" values are:

- **text/javascript (Default**): This is the default value if the type attribute is not specified. However, since HTML5, the **type** attribute is optional for JavaScript, and you can omit it:

  Example:

  ```
  <script type="text/javascript">
    // JavaScript code here
  </script>
  ```

- **text/ecmascript:** You can use text/ecmascript for ECMAScript 6 scripts, though this is less common nowadays.

  Example:

  ```
  <script type="text/ecmascript">
    // ECMAScript 6 code here
  </script>
  ```

- **application/json:** If you're embedding JSON data in a **<script>** tag, you can use **application/json** as the **type**.

  Example:

  ```
  <script type="application/json">
   {
     "key": "value",
     "array": [1, 2, 3]
   }
  </script>
  ```

- **application/ecmascript**
- **application/javascript**

4. **{TitleComponent} vs {<TitleComponent/>} vs {<TitleComponent></TitleComponent>} in JSX**

   In JSX (JavaScript XML), these three expressions involve the usage of a component named TitleComponent. JSX is a syntax extension for JavaScript that looks similar to XML or HTML and is commonly used with React for building user interfaces.

   a) {TitleComponent}:

   - This expression is used to reference the TitleComponent component without rendering it.
   - It's useful when you want to pass the component as a reference to another component or as a prop without rendering it directly.
   - **Code:**

```
const AnotherComponent = ({ title }) => {
 return <div>{title}</div>;
};
const App = () => {
 return (
  <AnotherComponent title={TitleComponent} />
 );
};
```

   b) {<TitleComponent/>}:

   - This expression renders the **TitleComponent** component.
   - It's a self-closing tag syntax used to create and render an instance of the **TitleComponent** component.
   - **Code:**

```
const App = () => {
 return (
  <div>
   <TitleComponent />
  </div>
 );
};
```

c) {<TitleComponent></TitleComponent>}:

- This expression is equivalent to the previous one, creating and rendering an instance of the **TitleComponent** component.
- It uses an opening and closing tag syntax, which is more traditional in XML and HTML, but in JSX, the self-closing tag is typically preferred.
- **Code:**

```
const App = () => {
 return (
  <div>
   <TitleComponent></TitleComponent>
  </div>
 );
};
```

In summary, the choice between these syntaxes depends on the specific use case.

Use **{TitleComponent}** when you want to reference the component without rendering it, and

Use **{<TitleComponent/>}** or **{<TitleComponent></TitleComponent>}** when you want to render the component. The self-closing tag syntax (**{<TitleComponent/>}**) is more common and concise in JSX.