React Basics Review

JavaScript Libraries and Frameworks

- JavaScript libraries and frameworks offer quick solutions to common problems and speed up development by providing pre-built code.
- Libraries are generally more focused on providing solutions to specific tasks, such as manipulating the DOM, handling events, or managing AJAX requests.
- A couple of examples of JavaScript libraries are jQuery and React.
- Frameworks, on the other hand, provide a more defined structure for building applications. They often come with a set of rules and conventions that developers need to follow.
- Examples of frameworks include Angular and Next.js, a meta framework for React.
- **Single page applications** (SPAs) are web applications that load a single HTML page and dynamically update that page as the user interacts with the application without reloading the entire page.
- SPAs use JavaScript to manage the application's state and render content. This is often done using frameworks which provide great tools for building complex user interfaces.
- Some issues surrounding SPAs include:
    - Screen readers struggle with dynamically updated content.
    - The URL does not change when the user navigates within the application, which can make it difficult to bookmark, backtrack or share specific pages.
    - Initial load time can be slow if the application is large as all the assets need to be loaded upfront.

React

- React is a popular JavaScript library for building user interfaces and web applications.
- A core concept of React is the creation of reusable UI components that can update and render independently as data changes.
- React allows developers to describe how the UI should look like based on the application state. React then updates and renders the right components when the data or the state changes.

React Components

- Components are the building blocks of React applications that allow developers to break down complex user interfaces into smaller, manageable pieces.
- The UI is described using JSX, an extension of JavaScript's syntax, that allows developers to write HTML-like code within JavaScript.
- Components are basically JS functions or classes that return a piece of UI.

Here is an example of a simple React component that renders a greeting message:

```
function Greeting() {
  const name = 'Anna';
  return <h1>Welcome, {name}!</h1>;
}
```

To use the component, you can simply call:

```
  <Greeting />
```

Importing and Exporting React components

- React components can be exported from one file and imported into another file.
- Let's say you have a component named `City` in a file named `City.js`. You can export the component using the `export` keyword:

```
// City.js
function City() {
  return <p>New York</p>;
}

export default City;
```

- To import the `City` component into another file, you can use the `import` keyword:

```
// App.js
import City from './City';

function App() {
  return (
    <div>
      <h1>My favorite city is:</h1>
      <City />
    </div>
  );
}
```

- The `default` keyword is used as it is the default export from the `City.js` file.
- You can also choose to export the component on the same line as the component definition like this:

```
export default function City() {
  return <p>New York</p>;
}
```

Setting up a React project using Vite

- Project setup tools and CLIs provide a quick & easy way to start new projects, allowing developers to focus on writing code rather than dealing with configuration.
- Vite, a popular project setup tool and can be used with React.
- To create a new project with Vite, you can use the following command in your terminal:

```
npm create vite@latest my-react-app -- --template react
```

This command creates a new React project named `my-react-app` using Vite's React template. In the project directory, you will see a `package.json` file with the project dependencies and commands listed in it.

- To run the project, navigate to the project directory and run the following commands:

```
cd my-react-app # path to the project directory
npm install # installs the dependencies listed in the package.json file
```

- Once the dependencies are installed, you should notice a new folder in your project called `node_modules`.
- The `node_modules` folder is where all the packages and libraries required by your project are stored.
- To run your project, use the following command:

```
npm run dev
```

- After that, open your browser and navigate to `http://localhost:5173` to see your React application running.
- To actually see the code for the starter template, you can go into your project inside the `src` folder and you should see the `App.jsx` file.

Passing props in React components

- In React, props (short for properties) are a way to pass data from a parent component to a child component. This mechanism is needed to create reusable and dynamic UI elements.
- Props can be any JavaScript value. To pass props from a parent to a child component, you add the props as attributes when you use the child component in the parent's JSX. Here's a simple example:

```
// Parent component
function Parent() {
  const name = 'Anna';
  return <Child name={name} />;
}

// Child component
function Child(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

You can pass multiple props using the spread operator `(...)`, after converting them to an object. Here's an example:

```
// Parent component
function Parent() {
  const person = {
    name: 'Anna',
    age: 25,
    city: 'New York'
  };
```

```
  return <Child {...person} />;
}
```

In this code, the spread operator `{...person}` converts the person object into individual props that are passed to the Child component.

Conditional rendering in React

- Conditional rendering in React allows you to create dynamic user interfaces. It is used to show different content based on certain conditions or states within your application.
- There are several ways to conditionally render content in React. One common approach is to use the ternary operator. Here's an example:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log in</h1>}
    </div>
  );
}
```

- Another way to conditionally render content is to use the logical AND (`&&`) operator. This is useful when you want to render content only if a certain condition is met. Here's an example:

```
function Greeting({ user }) {
  return (
    <div>
      {user && <h1>Welcome, {user.name}!</h1>}
    </div>
  );
}
```

In the code above, the `h1` element is only rendered if the user object is truthy.

You can also use a direct `if` statement this way:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  }
  return <h1>Please sign in</h1>;
}
```

Rendering lists in React

- Rendering lists in React is a common task when building user interfaces.
- Lists can be rendered using the JS array `map()` method to iterate over an array of items and return a new array of JSX elements.
- For example, if you have an array of names that you want to render as a list, you can do the following:

```
function NameList({ names }) {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={`${name}-${index}`}>{name}</li>
      ))}
    </ul>
  );
}
```

- Always remember to provide a unique key for each list item to help React manage the updating and rendering roles. With these techniques, you can create flexible, efficient, and dynamic lists in your React applications.

Inline styles in React

- Inline styles in React allow you to apply CSS styles directly to JSX elements using JavaScript objects.
- To apply inline styles in React, you can use the style attribute on JSX elements. The style attribute takes an object where the keys are CSS properties in camelCase and the values are the corresponding values. Here's an example:

```
function Greeting() {
  return (
    <h1
      style={{ color: 'blue', fontSize: '24px', backgroundColor: 'lightgray' }}
    >
      Hello, world!
    </h1>
  );
}

export default Greeting;
```

You can also extract the styles into a separate object and reference it in the `style` attribute this way:

```
function Greeting() {
  const styles = {
    color: 'blue',
    fontSize: '24px',
    backgroundColor: 'lightgray'
  };
```

```
  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- Inline styles support dynamic styling by allowing you to conditionally apply styles based on props or state. Here is an example of how you can conditionally apply styles based on a prop:

```
function Greeting({ isImportant }) {
  const styles = {
    color: isImportant ? 'red' : 'black',
    fontSize: isImportant ? '24px' : '16px'
  };

  return <h1 style={styles}>Hello, world!</h1>;
}

export default Greeting;
```

- In the code above, the `color` and `fontSize` styles are conditionally set based on the `isImportant` prop.