Hyperiondev

TASK

React - Elements

Visit our website

Introduction

WELCOME TO THE REACT - ELEMENTS TASK!

Welcome to the world of React elements! After obtaining a broad overview of this powerful library, you are now ready to delve into the first fundamental aspect of rendering elements using React. In this task, we'll cover essential concepts like React elements, their creation, and the dynamic JSX syntax, along with its transformation using Babel. You'll learn how to render your elements to the DOM, breathe life into your React components, and explore the value of conditional rendering. Moreover, you'll unlock the potential of dynamic rendering for lists, building versatile and interactive user interfaces. And with CodeSandbox at your fingertips, you'll have practical, hands-on experience, turning your creativity into tangible React applications.

REACT ELEMENTS

React elements are similar to DOM elements (like `div`, `body`, `head`, etc.). You should already be comfortable creating static HTML pages using various DOM elements. Like HTML, with React, you can use elements to build user interfaces (UIs).

The official definition of **React** elements is as follows:

"React elements are the building blocks of React applications. One might confuse elements with a more widely known concept of 'components'. An element describes what you want to see on the screen. React elements are immutable[1]*.*

```
const element = <h1>Hello, world</h1>;
```

Typically, elements are not used directly, but get returned from components."

We will learn more about how to create React components and the relationship between React elements and React components in the next task.

[1] Immutable: unable to be changed

CREATING REACT ELEMENTS

There are three basic steps that you will use to create any React elements:

1. Create React and ReactDom objects by importing React and React-DOM
2. Create the React Element using either JavaScript or JSX
3. Render the element that you have created to the DOM using the `render()` method

1. Create React and ReactDom objects by importing React and React-DOM

Before creating any React elements, you need to import the React library and the React-Dom package in the `index.js` file of the project. The React library is required to create React elements (and eventually React components), whereas the React-Dom package provides DOM-specific methods including the `render()` and `createRoot()` methods. You import these as shown below:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

A note from the
HyperionDev Team

The **import statement**, which may be new to you, is used to import functions, objects, or variables that are exported by another module. You will learn more about this later but for now, know that importing from React returns an object that we can use for creating React apps.

2. Create the React Element using either JavaScript or JSX

There are various ways in which you can create a React element. You could simply declare a variable that stores HTML or JSX, or a combination of HTML and JSX, or your variable could consist of more than one HTML element.

So what is JSX?

JSX is a Javascript Syntax Extension, sometimes referred to as JavaScript XML, that can make creating React applications a lot quicker and easier. JSX can be thought

of as a mix of JavaScript and XML. Like XML, JSX tags have a tag name, attributes and children. With JSX, if an attribute value is enclosed in quotes, it is a string and if the value is wrapped in curly braces `{ }`, it is an enclosed JavaScript Expression.

In the example below we declared a variable called `name` with a String value of "*Hyper Dave*" assigned to it. JSX was then used to assign the value of the `name` variable to a variable called `element` by wrapping the `name` variable within curly braces `{ }`.

```
const name = "Hyper Dave";
const element = (<h1>Welcome back, {name}</h1>);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
  </React.StrictMode>
);
```

When interpreted (rendered), the above example would result in the *element* variable containing the value `<h1>Welcome back, Hyper Dave</h1>`. Therefore, a JSX Expression can almost be seen as *"Embedding JavaScript within HTML elements"*. Note, if the curly braces `{ }` were removed in the second line, then the *element* variable would contain the value of `<h1>Welcome back, name</h1>`.

In the previous example, we used JSX to refer to a variable called `name`. However, we can take this even further; it is possible to execute any **JavaScript expression** by placing it within curly braces `{ }` in JSX.

In the following example, we will be creating a JavaScript object named `user` that contains some generic user data, and a JavaScript function named `formatName()`. The `formatName()` function receives one parameter named `person` and returns a String containing the person's full name (the result of calling said function). With JSX we can embed the result of calling the `formatName()` function directly into an `<h1>` element.

```
import profilePicUser from "./profiles/HD.png"

 function formatName(person) {
   return person.firstName + " " + person.lastName;
 }
```

```
const user = {
  firstName: "Hyper",
  lastName: "Dave",
  profilePic:{profilePicUser}
};

const element = (
  <h1>
    Welcome back, {formatName(user)}!
  </h1>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
  </React.StrictMode>
);
```

When interpreted (rendered), the above-mentioned example would result in the *element* variable containing a value of `<h1>Welcome back, Hyper Dave!</h1>`. The `formatName()` function is passed an argument named `user`, which is then used within this function to produce a result. Note, if the curly braces `{ }` were removed within the `<h1>` (in the `element` variable), then the `element` variable would contain a value of `<h1>Welcome back, formatName(user)!</h1>`.

Both previous examples have a very similar result, therefore, you might wonder why anyone would take the longer route of the second example. Well, the answer lies in the fact that the JavaScript Expression – in this case, a function – can perform a lot of calculations before returning a result.

There are some rules that you should be aware of when using JSX, some of which have been implemented in the examples above:

- As with HTML, with JSX you can specify various elements that can have different attributes.
    - With JSX you can use any HTML elements like `div`, `img`, or `h1`. You can also create user-specified elements.
- With JSX, if an attribute's value is enclosed within quotes, it is a String, also known as a String literal (follow this **link** if you require a recap on how Strings are used in JavaScript). If the value assigned to the attribute is

wrapped in curly braces, it is an enclosed JavaScript expression. An attribute's value can be assigned using *either*:

- String literals can be used for online images, e.g. `<img src = "https://www.hyperiondev.com/static/images/logo.png" />` String literals should be enclosed in quotation marks.

  Or you could try:

- A JavaScript Expression, e.g. `<img src = {user.profilePic} />` JavaScript Expressions should always be enclosed in curly braces instead of quotation marks, e.g. `{user.profilePic}`.

- When using images from your current local computer, they must always be imported e.g. `import profilePicUser from "./profiles/HD.png"`.

- In both previous examples, our React element (the `element` JavaScript variable) only contained one HTML element, namely an `h1`. It is, however, possible to assign multiple HTML elements to a single React element. To accomplish this, we have to take note of the following:

  - Each React element can only contain one parent element, for example, the `div` HTML element in the next example.
  - The parent element, however, can contain many child elements. You'll notice that the `h1` and `img` HTML elements are nested within the `div` HTML.
  - The entire value has to be contained within braces `( )`.

```
const element = (
  <div>
    <h1>I'm learning React with HyperionDev</h1>
    <img src="https://www.hyperiondev.com/static/images/logo.png"
style={{background:"grey"}}
/>
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
```

```
  </React.StrictMode>
);
```

Note: *The image in the code sample above is a PNG file. As PNG (Portable Network Graphics) supports transparency, it might look like the image is not loaded (as the background of the image is transparent). Adding styling (CSS) to React Elements will be discussed later in this task.*

A note from the
HyperionDev Team

What is the purpose of React Strict Mode?
Consider the example above. The render() method contains a `<React.StrictMode>` component. In the development environment, this component is merely used to indicate any potential errors/problems within your application. You can read more about React Strict Mode **here**.

Babel

Learning any new concept, like JSX, can be a daunting experience at first, however, Babel is a tool that can be used to convert JSX to JavaScript. Go to the **Babel REPL** website and copy and paste the code examples above into the 'Write code here' space provided. Babel will then show you the differences between the examples of React code (using JSX) and code written using just JavaScript. Note that Babel conversion is automatically incorporated into the React compilation process, so you never need to worry about it directly.

3. Render the element that you have created to the DOM

Once you have created a React element, you still have to render it so that it is visible to the user. You will notice that each of the code examples above contains a `render()`, and `ReactDOM.createRoot()` method, for example:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
  </React.StrictMode>
);
```

The `ReactDOM.createRoot()` method is used to obtain a container node to which you will render the React element (or React component) that you have created. If you recall, the container node in all the previous examples was `document.getElementById('root')`, a reference to, in this case, an *HTML* element that contains an `id` attribute with a value of `root` (follow this **link** if you require a recap on how the `document.getElementById()` method works).

Once we have created a container node, we still need to call the `render()` method on this node. The argument you pass to the `render()` method is the content that you want to display to the user, in the examples, it was the React element named `element`.

A note from the
HyperionDev Team

As a developer, you will never stop learning new techniques
As technology improves, the approaches and/or programming languages used to create applications and/or websites/apps are also updated. At the point that this task was last updated, the current version of React was React 18, therefore, all examples are based on this version. When doing your own research, you might come across some previous approaches, which might still be widely used. One of these approaches was how we rendered React elements (and components) as per the example below;

```
ReactDOM.render(element, document.getElementById('root'));
```

In previous versions of React, the first argument passed to the **`render()`** method was the React element/component that should be rendered. The second argument passed was the DOM element to which you want to append the React element/component.

CONDITIONAL RENDERING

One of the advantages of JSX is conditional rendering. Because we can make use of JavaScript expressions in JSX, we can render elements conditionally. So, if you think back to the conditional statements we used in vanilla JavaScript, JSX conditional rendering is almost identical in many ways. An element or even a contained group of elements can be rendered based on some condition. Your next question would probably be: what would that look like? Well, using a **ternary operator** (*condition ? valueA : valueB;*) it would look like this:

```
 const person = 'Dave';

 const element = (
  <div>
    { person === ‘Dave’ ? (
          <p>Hi, Dave!</p>
       ) : (
          <p>Hi, John!</p>
       )
    }
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
  </React.StrictMode>
);
```

As you can see from this code, a ternary operator is used to determine if `person` is equal to the string ‘Dave’ and if so, to render `<p>Hi, Dave!</p>` and if not `<p>Hi, John!</p>`.

Conditional rendering can be as simple or complex as you need it to be and is indispensable when designing dynamic web pages.

DYNAMIC RENDERING OF LISTS

Another exciting aspect of JSX is the ability to render lists dynamically using the iterative **Array.prototype.map()** method that you learnt about in level one. Remember, this method has one parameter, a callback function, and the callback function can take up to three parameters, two of which is the item of each iteration and then the array index of that item. The use of this method is again almost identical to vanilla JavaScript with the exception that it is always necessary to assign a key to each list element. In practice, this method would look like this:

```
 const itemList = [‘Apples’, ‘Strawberries’, ‘Bananas’, ‘Nectarines’];
```

```
const element = (
   <div>
      {itemList.map((item, index) => {
         return (
            <li key={index}>{item}</li>
          )
         })
      }
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {element}
  </React.StrictMode>
);
```

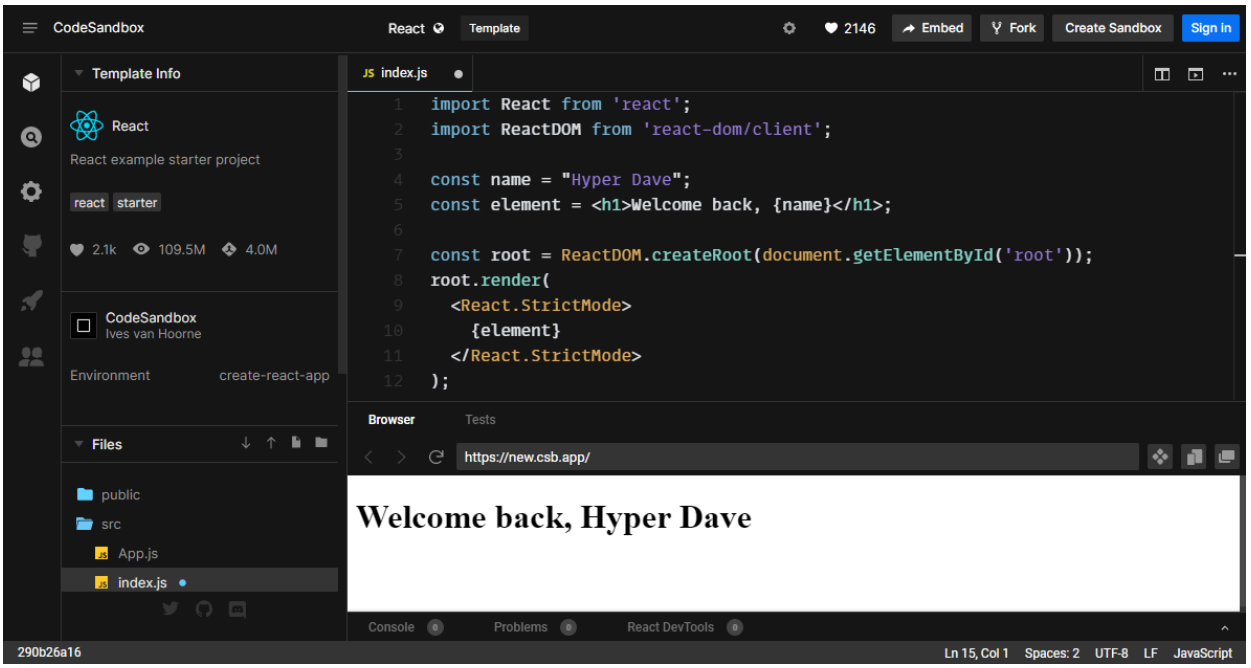
CODESANDBOX

Now that we have some introduction theory out of the way, let's implement the examples used so far. A bit later in this task, we will make use of Create React App (CRA), which is used to create a React app, for development, on your local machine. However, a lot of online code editors exist, one of which is **CodeSandbox**. CodeSandbox is an online code editor that allows us to easily create, test, and share basic React apps.

Let's execute some of our previous examples:

1. Follow the following **link** and click on the React "start from template" block to open CodeSandbox's online editor within your browser.
2. Select the **`index.js`** file (it should be located under Files, within the src folder).
3. Delete everything in **`index.js`** and add the two import statements (i.e., **`import React, {StrictMode} from 'react';`** and **`import ReactDom from 'react-dom/client';`**. Remember, you'll need these import statements every time you create React elements.
4. Copy and paste each of the code examples that we have considered above (code examples in the JSX section of this document) into CodeSandBox one at a time. For each element, study the code and the output in CodeSandbox. Be sure you understand the code in each code example.

5. Amend some of the React elements (code examples) and not the changes in output, to familiarise yourself with how they are rendered.



The first example is rendered in CodeSandbox

With this done, you are familiar with the rendering of elements in React and are ready to go to the next exciting section, which is the creation of your own, custom elements, called functional components.

Compulsory Task 1

Follow these steps:

- Use **CodeSandbox** to create the following elements.
    - A button that links to your LinkedIn/Instagram/Twitter profile when clicked (you decide where it links to).
    - A heading that contains the current date and time, formatted for easy readability. Follow this **link** if you require help.

    - A bullet-point summary of the advantages of JavaScript frameworks/libraries (like React) when compared to vanilla JavaScript.
- Copy and paste the code for each element you create from CodeSandbox into a file called `my-React_elements.txt.`
- Make sure that your `my-React_elements.txt` file is saved to your Dropbox folder for this task.

Compulsory Task 2

Create a web page with React and JSX with the following content:

- Create a folder called `ReactElements` on your local machine.
- Open the command line interface/terminal and `cd` to the folder you have created above.
- Follow the instructions found **here** to install React using the Create React App Starter Kit. Name your React App `react-hello`.
- Once you have started your front-end server (with `npm start`), test the default React App that was created by CRA by navigating to `http://localhost:3000/` in your browser.
- Modify `App.js` file (within the `src` directory):
    - Delete all the code inside the return( ) statement in the App function (Lines 6 - 21) and replace it with a new JSX component that will be exported to the index.js file to be rendered.
    - Create a JavaScript object called **user,** that stores all the details for a particular user of your app, above the App function but below the import statements within the App.js file.
        - This object should have at least the following properties: **name**, **surname**, **date_of_birth**, **address**, **country**, **email**, **telephone**, **company**, **profile_picture** (source of where

the image can be found), and **shopping_cart**. The **shopping cart** property should be used to store an array of items in the user's shopping cart.

    - Edit the App component to use **JSX** to display all the information about the user object that you created earlier in an attractive way. This element should also make use of a custom stylesheet that you have created.
    - The App component will be exported to the index.js file which will render the component automatically when you run the app using `npm run start`.
- Once you are ready to have your code reviewed, **delete** the `node_modules` folder (please note: this folder typically contains hundreds of files which, if you're working directly from Dropbox, has the potential to **slow down Dropbox sync and possibly your computer**), compress your project folder and add it to the relevant task folder in Dropbox.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

- React Official Website (2023) Retrieved on 19 July 2023 from **React official documentation**
- MDN Webdocs - Import (2023) Retrieved on 19 July 2023 from **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import**
- MDN Webdocs - JavaScript expressions and operators (2023) Retrieved on 20 July 2023 **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions**
- MDN Webdocs - String (2023) Retrieved on 20 July 2023 **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String**
- React Official Website - Strict Mode (2023) Retrieved on 20 July 2023 **https://react.dev/reference/react/StrictMode**
- BABEL official REPL Retrieved on 20 July 2023 **https://babeljs.io/repl/**
- MDN Webdocs - Document: getElementById() method (2023) Retrieved on 20 July 2023 **https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById**
- MDN Webdocs - Conditional (ternary) operator (2023) Retrieved on 20 July 2023 **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator**
- MDN Webdocs - Array.prototype.map() method (2023) Retrieved on 20 July 2023 **https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map**
- Codesandbox online editor Retrieved on 20 July 2023 **https://codesandbox.io/s/**
- React official website - Writing markup with JSX (2023) Retrieved on 20 July 2023 **https://react.dev/learn/writing-markup-with-jsx**
- React official website - Start a new React project (2023) Retrieved on 20 July 2023 **https://react.dev/learn/start-a-new-react-project#create-react-app**
- React Bootstrap official website - Getting started (2023) Retrieved on 20 July 2023 **https://react-bootstrap.netlify.app/docs/getting-started/introduction/**