



TASK

Introducing React Elements and Components

Visit our website

Introduction

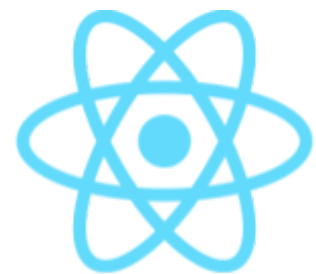
WELCOME TO THE REACT ELEMENTS AND COMPONENTS TASK!

Web developers often use web development frameworks and libraries to create web applications more quickly and efficiently. React.js, also known as ReactJS, but more commonly known as React, is a JavaScript library for building front-end web applications.

In this task, you will learn what React is and how to use a React starter kit to start developing a React application with minimal configuration. You will also learn to create and render elements using React. You will also learn about components, which are the fundamental building blocks of user interfaces, providing independent and reusable UI elements.

What is React?

React generates user interfaces (UIs) that must ultimately convert to HTML for web browsers. While HTML and CSS can create static pages, they can't implement logic like input validation or calculations. JavaScript makes web pages dynamic, and React leverages it to write HTML.



React logo

React is a JavaScript library (not a framework). It provides tools and functionalities to build user interfaces, focusing on rendering components and managing the UI state. It is designed to handle specific tasks, like rendering the Document Object Model (DOM) efficiently, but it doesn't dictate how to structure the entire application, allowing developers to choose additional libraries for routing, state management, and other concerns.

A **framework** offers a more comprehensive solution. While React is just a library, **Next.js**, for example, is a React-based framework. It provides built-in features like routing, server-side rendering (SSR), and API routes, giving developers a complete structure and set of tools to build and scale a full-stack web application.

React was created by [Jordan Walke](#) at Meta and first used on Facebook in 2011. It allows developers to describe UIs declaratively, meaning you specify what the UI should look like, and React handles the rest.

Before we get started with React, there are a few concepts that we need to understand. Each of these core concepts is discussed under the subsequent headings in this task. Some code examples used in this task are from [React's official documentation](#).

Common React terms

Here is a list of common React terms to remember:

- **JSX:** Stands for JavaScript XML. JSX is a JavaScript extension syntax used in React to easily write HTML and JavaScript together.
- **React elements:** The building blocks of React applications.
- **React components:** Small, reusable pieces of code that return a React element to be rendered to the page.
- **Props:** Inputs to a React component. They are data passed down from a parent component to a child component.
- **State:** An object that holds data and information related to a React component. It can be used to store, manage, and update data within the application.
- **Hooks:** These let you use different React features from your components.

The virtual DOM

The image below, from the [World Wide Web Consortium](https://www.w3.org/) (W3C), visualises a Document Object Model (DOM) for an HTML table. The DOM is a programming API for HTML and XML documents that defines the logical structure of documents and how they are accessed and manipulated.

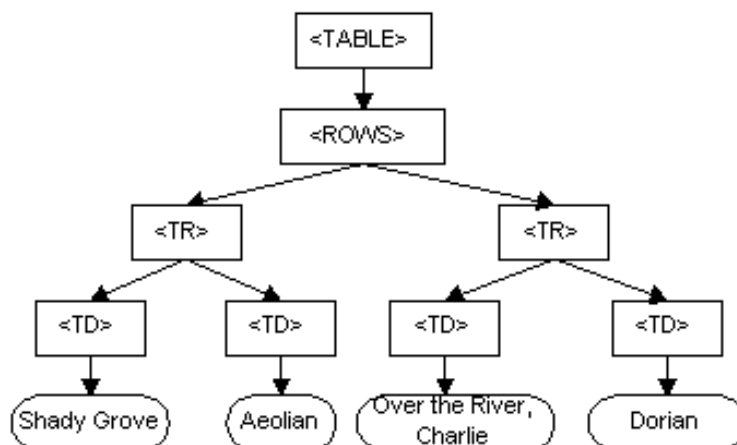


Table (Robie, J., Texcel Research, 1998).

<https://www.w3.org/TR/WD-DOM/introduction.html>

As web developers, we use the DOM to manipulate HTML documents. We can add, delete, or even change HTML elements using the DOM; in fact, you have already been doing so by making use of JavaScript! For instance:

```

// Select the 'div' element with the id 'container'
let div = document.getElementById("container");

// Create an 'img' element
let imgProfile = document.createElement("img");

// Set the source attribute for the image
imgProfile.src = "../pictures/profilePic.png";

// Append the 'img' element to the 'div' container
div.appendChild(imgProfile);

// Create a 'p' element (paragraph)
let paragraph = document.createElement("p");

// Set the text content for the paragraph
paragraph.innerHTML = "Dynamically adding a paragraph to HTML";

// Append the paragraph to the 'div' container
div.appendChild(paragraph);

```

However, constantly updating the DOM every time the HTML content changes can significantly slow down your web application. This is because the browser must re-render parts of the page each time a change occurs, which can be resource intensive.

React solves this problem by using a **virtual DOM**, which is an efficient in-memory representation of the actual HTML document. When changes occur in the application, React updates the virtual DOM first. It then compares the updated virtual DOM with its previous version to identify what has changed. This comparison process, known as "reconciliation", allows React to determine the specific changes needed in the real DOM. Instead of rewriting the entire DOM, only the parts that have changed are updated, this helps to enhance the overall performance greatly.

A virtual DOM is a virtual representation of the HTML document in memory. React works by taking a 'snapshot' of the DOM before changes are made. As changes are made, the virtual DOM is updated. After the changes are complete, the virtual DOM and the snapshot of the DOM (which is also saved in memory) are compared to see where changes were made. Instead of rewriting the entire DOM, the real DOM is only updated with the changes that were made.

Why do we use React?

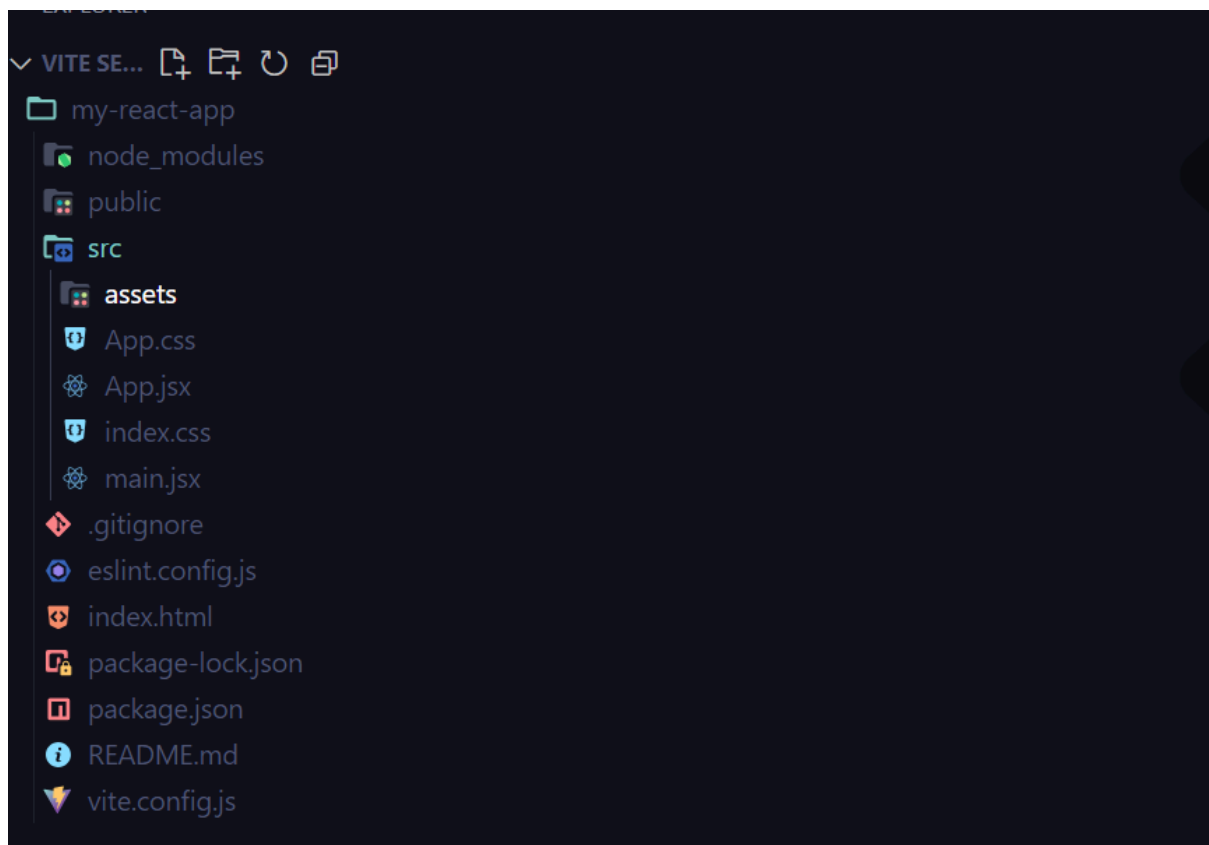
1. **Building interactive websites easily:** React helps developers create websites that respond quickly to user actions. Imagine a social media feed that updates instantly when you like a post – React makes this kind of interactivity much simpler to build.
2. **Reusable pieces:** React lets you create small, reusable chunks of code called components. Think of these like building blocks. Just as you can use the same Lego pieces to build different structures, developers can reuse these components across different parts of a website or even in different projects. This saves a lot of time and effort.
3. **Efficient updates:** React is smart about updating what you see on the screen. Instead of refreshing the entire page when something changes, it only updates the specific parts that need to change, making websites faster and smoother.
4. **Large and helpful community:** React has been around for a while and is used by many big companies. This means there's a large community of developers who use it, create helpful tools for it, and are available to answer questions if you get stuck.
5. **Good for big projects:** As websites get bigger and more complex, they can become harder to manage. React provides a structure that helps keep things organised, even for large-scale applications.
6. **Learn once, use everywhere:** Once you learn React, you can use those skills to build not just websites but also mobile apps (using React Native) and even virtual reality applications.
7. **Job market demand:** Because so many companies use React, there's a high demand for developers who know how to work with it. This makes it a valuable skill for many programmers to learn.
8. **Backed by Meta:** React was created and is maintained by Meta (formerly Facebook). This means it has the support of a major tech company, giving developers confidence that it will continue to be improved and supported in the future.

In simple terms, React makes it easier and faster for developers to build modern, interactive websites that work smoothly. It provides a set of tools and practices that help manage the complexity of creating these sites, which is why it has become so popular in the web development world.

Setting up a React app with Vite

You'll need to refer to the **Additional Reading – Installation, Sharing, and Collaboration** guide to complete your React installation. The guide will walk you through everything you need to know with regards to installing React+Vite. You'll want to reference the 'Setting up a React app with Vite' section in the Additional Reading.

Once you have completed the setup, it will create a project folder with this specific directory structure, which includes:



By default, the React elements you create will be rendered in the **main.jsx** file that has been created for you by React+Vite. To find this file, open the project directory (folder) that was created with React+Vite. In this directory, you will see a directory called **src** that contains the **main.jsx** file. In this file, you will see an instruction to render (**root.render()**) any React elements or components, as shown in the example below:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
```

```
import './index.css'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

To understand where the React element or component you created will be rendered, you should find a file called **index.html** in your root folder above the **package.json**. This HTML file contains a **div** HTML element that has an **id** attribute with a value of **"root"** assigned to it. When the **render()** method is executed in **main.jsx**, it specifies that the React component or element being rendered will become a **child** of a specific **div** in your **index.html**. Here's how it works step by step:

index.html (HTML file): Inside your index.html, you typically have something like this:

```
<body>
  <div id="root"></div>
  <script type="module" src="/src/main.jsx"></script>
</body>
```

The **div** with **id="root"** is a placeholder (also known as a container node) that is empty when the page loads. It's where the React app will be rendered. In the **main.jsx** file, we use **createRoot** from the **react-dom/client** package to attach the React app to this container node:

```
import { createRoot } from "react-dom/client";
import App from "./App";

// Render the App component inside the container node directly
createRoot(document.getElementById("root")).render(<App />);
```

In this code, `document.getElementById('root')` finds the `div` element with `id="root"` in the `index.html` file. This container is where React will render the entire component tree. The `createRoot` function is used to establish a React root, and then the render method is called to insert the `App` component (and all its child components) into the `div#root`. This allows React to interact with that specific part of the DOM, making it the entry point for the React application.

Understanding React Elements

Once your React app is set up, we can dive into the fundamental concept of **React Elements**.

React Elements are the smallest building blocks of a React application. They describe what you want to see on the screen. Unlike a DOM element (like `<div>` or ``), a React Element is a simple object representation of a DOM element.

Key concepts:

- A React Element is a plain JavaScript object that represents the DOM tree.
- React Elements are immutable, meaning once you create an element, it cannot be changed. React only updates the parts of the DOM that need to change.
- The `React.createElement()` function is used to create a React Element, although, in most cases, we use JSX to simplify this.

```
// Without JSX
const element = React.createElement('h1', null, 'Hello, World!');

// With JSX
const element = <h1>Hello, World!</h1>;
```

Creating React Elements

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.

CREATE THE REACT ELEMENT USING EITHER JAVASCRIPT OR JSX

In React, you can create elements in various ways. 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.

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 instance 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.

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";

const name = "Hyper Dave";

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

createRoot(document.getElementById("root")).render(

  <StrictMode>

    {element}

  </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'. **Take 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, as it is possible to execute any [JavaScript expression](#) by placing it within curly braces in JSX.

In the following instance, 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.

```
// Importing the profile picture from the local file system
import profilePicUser from "../profiles/HD.png";

// Function to format the full name of the user
function formatName(person) {
  // Returns the first name followed by the last name
  return person.firstName + " " + person.lastName;
}

// Defining a user object with firstName, lastName, and profile picture
const user = {
  firstName: "Hyper",
  lastName: "Dave",
  profilePic: { profilePicUser } // Storing the imported image in the
  profilePic property
};

// Creating a JSX element that displays a welcome message with the formatted
name
const element = (
  <h1>
    Welcome back, {formatName(user)}! {/* Calls formatName to display the
full name */}
  </h1>
);

// Rendering the element inside the 'root' div in the HTML document
createRoot(document.getElementById("root")).render(
```

```
<StrictMode>

  {element} {/* StrictMode is used to highlight potential problems in the
app */}

</StrictMode>

);
```

When rendered, the above-mentioned instance would result in the **element** variable containing a value of `<h1>Welcome back, Hyper Dave!</h1>`. The **formatName()** function is called with the **user** object as an argument, producing the displayed 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>`.

When working with JSX in React, there are a few key rules and concepts to keep in mind. JSX allows you to write HTML-like code within JavaScript, but with some important differences. Below are the most important guidelines to help you write clean and effective JSX:

Key JSX rules and examples

1. HTML elements in JSX:

- With JSX, you can use any HTML elements such as `<div>`, ``, or `<h1>`, and also create custom user-defined elements.

2. Attribute values in JSX:

- If an attribute's value is enclosed in quotes, it is treated as a string (also called a string literal). For example:

```

```

- If the value is enclosed in curly braces, it is treated as a JavaScript expression. For instance:

```
<img src={user.profilePic} />
```

3. Using local images in JSX:

- When using images stored on your local computer, you must import them like this:

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

4. One parent element rule:

- Every React Element must have only one parent element. You can wrap multiple elements in a `<div>`, for example:

```
const element = (  
  <div>  
    <h1>Hello, World!</h1>  
    <img src={profilePicUser} />  
  </div>  
);
```

5. JavaScript expressions:

- JavaScript expressions within JSX must always be enclosed in curly braces `{}`. For instance:

```
const element = <p>{user.name}</p>;
```

6. Multiple child elements:

- Although each React Element must have one parent element, that parent element can contain multiple child elements.

```
const element = (  
  <div>  
    <h1>Welcome, {user.firstName}!</h1>  
    <p>Your profile picture:</p>  
    <img src={user.profilePic} alt="Profile" />  
  </div>  
);
```

```
    </div>
  );
```

7. Curly braces in JSX:

- Any JavaScript expression within JSX (such as variables, functions, or calculations) must be wrapped in curly braces `{}`. For example:

```
const element = (
  <div>
    <h1>I'm learning React with HyperionDev</h1>
    
  </div>
);

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </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.

WHAT IS REACT STRICT MODE?

Consider the example above. The `render()` method contains a `<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.

Render the element that you have created to the DOM using **createRoot** from **react-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 **createRoot()** from **react-dom/client** method, for instance:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";

createRoot(document.getElementById("root")).render(
  <StrictMode>{element}</StrictMode>
);
```

The **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**. This will be explained in more detail later in this task.

NEVER STOP LEARNING NEW TECHNIQUES

As technology advances, programming languages and methods for creating websites and apps evolve. At the time this task was updated, React 18 was the latest version, so all examples are based on it. You may encounter older approaches during your research, such as the way React Elements were previously rendered, as shown in 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 advantage of JSX is conditional rendering, which allows you to use JavaScript expressions to render elements based on a condition. It works similarly to conditional statements in vanilla JavaScript, letting you display elements or groups of elements depending on a condition.

You might wonder: what does that look like? Well, you can use a **ternary operator** (i.e., **condition ? valueA : valueB;**) for conditional rendering. The ternary operator is a concise way to perform conditional logic in a single line. Here's an example:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";

const person = "Dave";

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

createRoot(document.getElementById("root")).render(
  <StrictMode>{element}</StrictMode>
);
```

In the above example, the ternary operator checks if the variable **person** is equal to the string **"Dave"**. If this condition is True, it renders `<p>Hi, Dave!</p>`. If the condition is False (i.e., **person** is not **"Dave"**), it renders `<p>Hi, John!</p>`.

Conditional rendering can be as simple or complex as needed, and is invaluable for creating dynamic web pages, allowing you to change what is displayed based on user interactions or application state.

Dynamic rendering of lists

Another exciting aspect of JSX is the ability to render lists dynamically using the iterative **`Array.prototype.map()`** method. This method takes a callback function as its parameter, which can have up to three arguments: the item of each iteration and its array index. It's similar to vanilla JavaScript, except each list element must have a key assigned. Here's an example:

```
const itemList = ["Apples", "Strawberries", "Bananas", "Nectarines"];
const element = (
  <ul>
    {itemList.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
createRoot(document.getElementById("root")).render(
  <StrictMode>{element}</StrictMode>
);
```

DIFFERENCES BETWEEN FUNCTIONAL COMPONENTS AND CLASS COMPONENTS

Class components were a type of component that was widely used in **older versions** of the React library. However, with the introduction of React hooks, functional components have become the preferred way of writing components due to their simplicity and better code organisation. As a result, class components are now largely considered outdated,

although you may still encounter them in legacy React projects. For more information about class components, refer to [this link for more information](#).

When talking about components moving forward, it will be referring to **functional components**.

The table below compares the key differences between functional and class components:

Functional components	Class components
A pure JavaScript function that accepts props as an argument and returns a React Element (JSX).	Requires you to extend from React.Component and create a render function that returns a React Element.
There's no render method used.	It must have the render() method returning JSX (which is syntactically similar to HTML).
Run from top to bottom; once the function is returned, it can't be kept alive.	Once it is instantiated, a different life cycle method is kept alive and is run and invoked depending on the phase of the class component.
Also known as stateless components, they accept data, display it in some form, and are mainly responsible for rendering UI.	Also known as stateful components, because they implement logic and state.
React life cycle methods (for example, componentDidMount) cannot be used in functional components.	React life cycle methods (for example, componentDidMount) can be used inside class components.
Hooks can be easily used in functional components to make them stateful. example: <code>const [name,SetName]=React.useState('')</code>	Different syntax is required inside a class component to implement hooks. example: <code>constructor(props) { super(props); this.state = {name: ''} }</code>
Constructors are not used.	A constructor is used, as it needs to store the state.

What is a functional component?

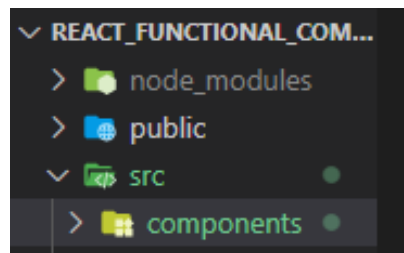
A functional component is a JavaScript function that accepts props (short for properties) as an argument and returns a React Element. Functional components are straightforward and focus on rendering UI based on the provided props. They are simpler to write and understand, making them the standard way of creating React components in modern applications.

Examples of functional components

We will create a React functional component to demonstrate the initialisation of function components and their usage in developing modular UI components. This approach helps you, as a developer, create reusable and manageable blocks of UI for your application.

CREATE A NEW FOLDER FOR COMPONENTS

Navigate to the “src” directory of your existing React project. Inside the “src” folder, create a new folder and name it **components**, as shown in the structure below:



In the next step, we will create a new JavaScript file called **Welcome.jsx** in the **components** directory and implement a standard functional component that will pass an object called **props**. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions. In this instance, we will be passing a ‘name’ prop to the component, which is a string value. The name prop will be accessed using standard **JavaScript object notation**. The object is called **props** and the property we are accessing is called **name**. As such, the object notation for accessing the property of the props object is **{props.name}**. What we can do is destructuring of props.

Destructuring props

Destructuring props in React is like unpacking a box of items so you can easily use them. In React, when a component gets props (data sent from another component), those props are usually bundled up in an object. Destructuring lets you pull out specific pieces from that object in a cleaner way.

For instance, if you have a **Person** component that receives a prop object with **name** and **age**, instead of writing **props.name** and **props.age** every time, you can ‘unpack’ them like this:

```
function Person({ name, age }) {  
  return (  
    <div>  
      <p>Name: {name}</p>  
      <p>Age: {age}</p>  
    </div>  
  );  
}
```

Here, the `{ name, age }` in the function parameters is destructuring the `props` object, so you can directly use `name` and `age` without typing `props.name` or `props.age` repeatedly. It makes the code shorter and easier to read.

WRITING THE WELCOME.JSX COMPONENT

To write this component, we will be using [JSX](#). JSX enables you to write HTML-like markup within a JavaScript file, keeping rendering logic and content seamlessly integrated. Sometimes, you may want to include a little JavaScript logic or reference a dynamic property inside that markup. In such cases, you can use curly braces in JSX, providing a gateway to JavaScript functionality. If you are unsure about JSX syntax, you can use a [converter](#) tool to convert HTML to JSX code.

```
//Create the Welcome Component

function Welcome({name, age}) {

  return (

    <div>

      {/* This h1 element uses JSX to display the name property of the props
      object */}

      <h1>Hello World, {name} you are {age} years old</h1>

    </div>

  );
}
```

```
}  
  
export default Welcome;
```

Welcome is a functional component in React. It is defined as a JavaScript function.

The parameter `{ name }` is the destructured version of the props object. Instead of accessing `props.name`, we directly access `name` by destructuring the props object passed to the component.

Inside the `<h1>` tag, the `{name}` expression uses curly braces to inject the value of the `name` prop passed to the `Welcome` component. For example, if you pass `name="Angus"` when using this component, it will render as **Hello World, Angus**. The export default `Welcome;` statement makes this component available for use in other files by importing it.

IMPORTING AND EXPORTING COMPONENTS

You can declare many components in one file, but large files with many components can get difficult to navigate and maintain. A common practice in React applications is to create each component in its own file, export the component (as can be seen in the code example above), and then import the component into another file.

To implement our newly created `Welcome.jsx` component, we will import the `Welcome` functional component into our `App.jsx` file and initiate some examples:

```
// Import CSS files for styling purposes

import "./App.css";

// importing the Welcome Component

import Welcome from "./components/Welcome";

function App() {

  return (

    <div>

      {/* Display an instance of the Welcome component

      passing a prop name="Joe Soap" */}

      <Welcome name="Joe Soap" />

    </div>

  );
}
```

```

    {/* Display a second instance of the Welcome component
    passing a prop name="John Smith" */}

    <Welcome name="John Smith" />

  </div>

);
}

export default App;

```

Here, we are displaying the name of a specific user using props. We have included the **Welcome** component in **App.jsx** and passed its **name** as an attribute. The **Welcome** component will receive this info as props, which is an object containing a **name** field inside it, and we can use this information in the **Welcome** component wherever we want.

To check whether everything is up and compiling correctly, open the command line and run the command **npm run dev** in the project folder, which we created when we ran the **React+Vite** command. You should see the following in your web browser:

Hello World, Joe Soap
Hello World, John Smith

Well done if everything is working! This is just one of the examples we will show you during this task on how to implement modular functional components. Then adjust the props passed to our function components by adding the age of the user, as below:

```

// Import CSS files for styling purposes
import "./App.css";

// importing the Welcome Component
import Welcome from "./components/Welcome";

function App() {

```

```

return (
  <div>
    {/* Display an instance of the Welcome component
    passing 2 props
    name="Joe Soap"
    age="39" */}
    <Welcome name="Joe Soap" age="39" />
    {/* Display a second instance of the Welcome component
    passing two props
    name="John Smith"
    age=39 */}
    <Welcome name="John Smith" age="52" />
  </div>
);
}
export default App;

```

By now, you will start to see the advantages of components over simple React Elements and how the implementation of functions can help you in future developments.

STYLING REACT COMPONENTS

Now let's add some style to our components. There are several ways to style React components, and some of the more popular ways to style your components are described below.

- **CSS styling:** Standard [CSS styling](#) can be applied to React components using CSS selectors, with a couple of notable exceptions. The HTML **class** attribute is replaced in React with **className**, for example:

```

<div className="App">

```

When applying inline styling to components, remember that inline CSS is written in a JavaScript object. For properties with two names, such as **background-color**, you should use **camelCase syntax**. Instead of “background-color”, use “background**C**olor”. For example:

```
<h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>
```

- **Library of React UI components:** Some common libraries include [React Bootstrap](#) and [Material UI](#), which contain pre-styled components that can be imported and used in your React application.
- **Styled components:** [These](#) allow you to use actual CSS syntax inside your components. Styled components are a variant of CSS-in-JS, which utilises tagged template literals (a recent addition to JavaScript) to style your components.

In this task, we will focus on using Bootstrap to style our components. We'll be working with the project folder (created when we ran the **React+Vite** command). This folder contains the **package.json** file, which **npm** uses to keep track of all the packages installed in your application.

It's important to distinguish between Bootstrap and React Bootstrap:

- Bootstrap is a popular CSS framework.
- React Bootstrap is a set of React components built on top of Bootstrap, which provides a more seamless integration with React applications.

For more details, check out the following links:

- **React Bootstrap:** Click [here](#)
- **Bootstrap:** Click [here](#)

Below, we will explore React Bootstrap and Bootstrap in greater detail.

To start, open up your terminal in your root folder and type in:

```
npm install bootstrap react-bootstrap
```

We'll then need to include the following import into our **main.jsx** file (**Bootstrap** is a front-end toolkit that can be used in standard HTML and JS applications. The **react-bootstrap** library is built as a React extension of standard Bootstrap):

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Next, add the following link to the `<head> </head>` tag of the **index.html** file in the **/public/** folder of your React application. Copy-paste the stylesheet `<link>` into your `<head>` before all other stylesheets to load the Bootstrap CSS.

```
<!-- Latest compiled and minified CSS →  
<link  
  
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.cs  
s"  
  
rel="stylesheet"  
  
>
```

This file links to the Bootstrap CSS stylesheet file from the cloud that contains CSS code that will style the components we import.

Many Bootstrap components require JavaScript to function, specifically Bootstrap's own JavaScript bundle, which includes **Popper.js**. This JavaScript enables dynamic components like modals, tooltips, and dropdowns. We add the Bootstrap JavaScript bundle at the end of the `<body>` section to ensure it loads after the HTML content, allowing these components to function correctly without interfering with the initial page load.

To include Bootstrap's JavaScript, add the following link to the end of the `<body>` tag in your **index.html** file:

```
<!-- Latest compiled JavaScript -->  
  
<script  
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min  
.js"></script>
```

The **index.html** file should now look like this:


```

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <link rel="icon" type="image/svg+xml" href="/vite.svg" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Vite + React</title>

    <!-- Latest compiled and minified CSS →

    <link

href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.cs
s"

    rel="stylesheet"

  />

</head>

<body>

  <div id="root"></div>

  <script type="module" src="/src/main.jsx"></script>

  <!-- Latest compiled JavaScript →

                                                                    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.m
in.js"></script>

  </body>

</html>

```

Now that we have installed, linked, and imported all of the Bootstrap requirements, let's import some Bootstrap components into **App.jsx** and look at how they differ from standard HTML components:

```

import './App.css';

import Welcome from './components/Welcome';

```

```

import Button from 'react-bootstrap/Button';

function App() {

  return (

    <div className="App">

      <hr />

      <Welcome name="Joe Soap" age="23" />

      <Welcome name="John Smith" age="39"/>

      <hr />

      {/* Adding standard CSS styling to an HTML component in React */}

      <h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>

      {/* A Standard HTML Button Component */}

      <button> a Regular HTML Button</button>

      <hr />

      {/* A Bootstrap Button Component */}

      <Button>a Standard Bootstrap Button</Button>

      <hr />

      {/* These are variants of the same Bootstrap button */}

      <Button variant="outline-secondary">a Bootstrap Outline Button</Button>

      <hr />

      <Button variant="success">a Bootstrap Success Button</Button>

      <hr />

      <Button variant="link">Link</Button>

    </div>

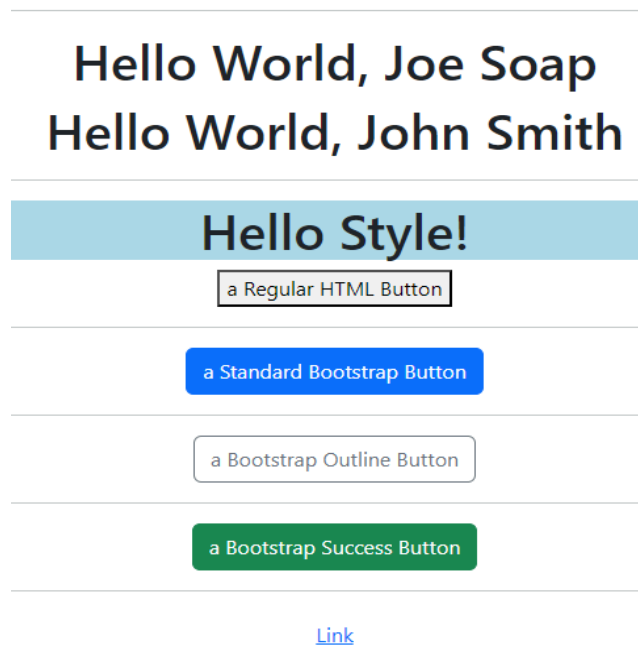
  );

}

export default App;

```

If you open up the React application in your browser, it should look similar to this:




The first button is a standard HTML button without styling.

Next, we have a Bootstrap button component.

The next three Bootstrap buttons are the same component with specific styling applied by using the **variant** keyword.

Bootstrap grid: Bootstrap's grid system uses a series of containers, rows, and columns. It's built with **flexbox** and is fully responsive. Below is an example of how the grid comes together.



Extra resource
New to or unfamiliar with flexbox? [Read this CSS-Tricks flexbox guide](#) for background, terminology, guidelines, and code snippets.

Want to practise using flexbox and grid?

- For using display: flex – just click [here](#)
- For using display: grid – just click [here](#)

First, let's add a class called **red-border** to our **App.css** file to make the rows and columns easier to see:

```

/* red-border Class */
.red-border {
  border: 1px solid red;
  padding: 5px;
  align-self: center;
  margin: 15px
}

```

Next, we'll create a **LayoutExample.jsx** component in the components folder, as seen in the code below. To use the React Bootstrap grid system, we need to **import** **Container**, **Row**, and **Col**:

```

import Container from 'react-bootstrap/Container';
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';

// Create the LayoutExample component
function LayoutExample() {
  return (
    // Create an instance of the Container component which will
    // automate the layout
    <Container>
      /* The Container component can have standard HTML elements as children
      */
      <h2>Layout Example</h2>
      <hr />
      <h3> Rows and Columns Auto Layout </h3>
      /* Using the Row & Col components we can create a Grid Layout */
      <Row>
        <Col className="red-border">Row 1 : Column 1 of 2</Col>

```

```

    <Col className="red-border">Row 1 : Column 2 of 2</Col>

  </Row>

  <Row>

    <Col className="red-border">Row 2 : Column 1 of 3</Col>

    <Col className="red-border">Row 2 : Column 2 of 3</Col>

    <Col className="red-border">Row 2 : Column 3 of 3</Col>

  </Row>

  <h3> Rows and Columns Specified sizes </h3>

  <Row>

    { /* We specify the size of the first column */ }

    <Col xs={3} className="red-border">Row 1 : Column 1 of 2</Col>

    { /* The second column fills up the remaining space */ }

    <Col className="red-border">Row 1 : Column 2 of 2</Col>

  </Row>

  <Row>

    { /* Each of the Columns has a fixed size */ }

    <Col xs={2} className="red-border">Row 2 : Column 1 of 3</Col>

    <Col xs={3} className="red-border">Row 2 : Column 2 of 3</Col>

    <Col xs={4} className="red-border">Row 2 : Column 3 of 3</Col>

  </Row>

</Container>

);
}

export default LayoutExample;

```

Let's take a look at Bootstrap [stacks](#). Stacks are shorthand helpers that build on top of Bootstrap's flexbox utilities to make component layout faster and easier.

Stacks are vertical by default and stacked items are full-width by default. Use the **gap** prop to add space between items. Use **direction="horizontal"** for horizontal layouts. Stacked items are vertically centred by default and only take up their necessary width.

Create a file called **StackExample.jsx**, as below, in the components folder:

```
import Stack from 'react-bootstrap/Stack';

function StackExample() {

  return (

    <div>

      <h3 > Vertical Stack</h3>

      /* Stacks allow for a quick Grid Layout
      without the need for multiple Row & Col components */

      /* a Stack component with a standard vertical layout */

      <Stack className="red-border" gap={2}>

        <div className="red-border">First item</div>

        <div className="red-border">Second item</div>

        <div className="red-border">Third item</div>

      </Stack>

      <h3>Horizontal Stack</h3>

      /* /* a Stack component with a horizontal layout */ */

      <Stack className="red-border" direction="horizontal" gap={2}>

        <div className="red-border">First item</div>

        <div className="red-border ">Second item</div>

        <div className="red-border ">Third item</div>

      </Stack>

    </div>

  );

};
```

```
}  
  
export default StackExample;
```

Now let's import these components into the **App.jsx** file to display them:

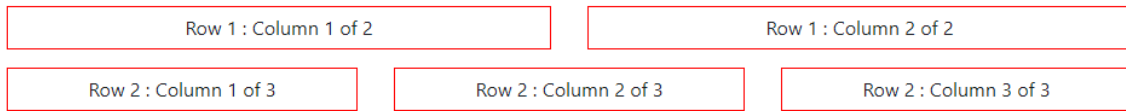
```
import "./App.css";  
  
// Importing the components  
  
import LayoutExample from "./components/LayoutExample";  
import StackExample from "./components/StackExample";  
  
function App() {  
  return (  
    <div className="App">  
      <hr />  
      {/* display the LayoutExample component */}  
      <LayoutExample />  
      <hr />  
      {/* display the StackExample component */}  
      <StackExample />  
    </div>  
  );  
}  
  
export default App;
```

Now, run the React application with the **npm run dev** command in your terminal and open the application in your browser.

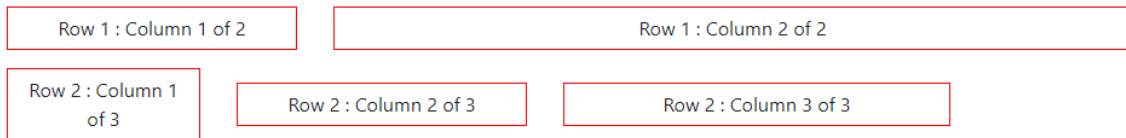
The result should look similar to this:

Layout Example

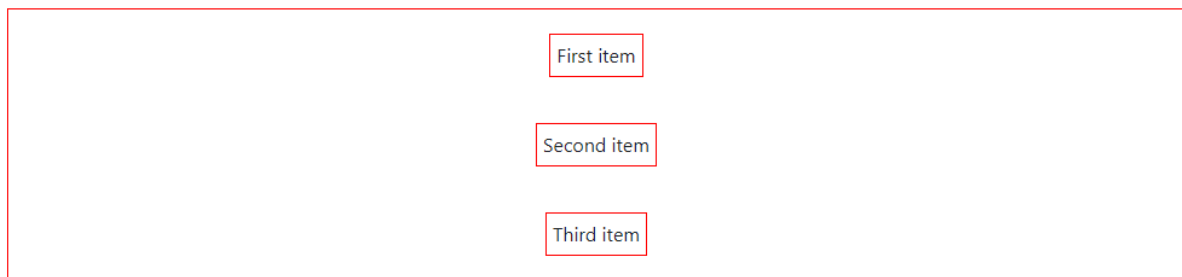
Rows and Columns Auto Layout



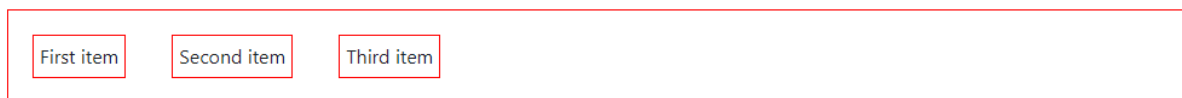
Rows and Columns Specified sizes



Vertical Stack



Horizontal Stack



Take note

The tasks) below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click "Request review" on your student dashboard, you will receive a 100% pass grade if you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded task 1

Follow these steps:

- On your computer, create an empty folder where you want your React project to be created.
- Open the terminal and navigate to the empty folder you just created.
- Once you're inside the folder, run the following command to create a new React project:

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

- Replace **my-app** with the desired name for your project. You can choose any name that fits your project.
- After Vite creates the project, navigate into the **my-app** folder:

```
cd my-app
```

- Install the necessary dependencies by running the following command:

```
npm install
```

- To start the React app, run the following command inside the **my-app** folder:

```
npm run dev
```

- The application will typically be accessible at **http://localhost:5173**. If this port is already in use, Vite will automatically assign a different available port, and you will see the new address in the terminal.
- Take a screenshot of the React application running in your browser and upload it to the task folder.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded 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 at the "Setting up a React app with Vite" section to create a React application and name the application **react-hello**.
- Once you have started your front-end server (with **npm run dev**), test the default React app that was created by React+Vite by navigating to **http://localhost:5173/** in your browser.
- Modify **App.jsx** 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 dev**.
- Again, once you are ready to have your code reviewed, delete the **node_modules** folder.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded task 3

Follow these steps:

- **Choose a website:** Find a published web page that you particularly like, such as Netflix, Instagram, Takealot, UCOOK, [HyperionDev](#), or any other website that inspires you. Ensure that the website you choose is feasible to **recreate** within the scope of this task. Keep in mind that you are tasked to recreate **a web page**, not the entire website.
- **Create a clone:** Create a React application, using **React+Vite**, that serves as a clone of the chosen website. Don't worry about adding state changes to your application.
- **Functional components:** Analyse the chosen website and identify its major components and sections. Implement functional components for different sections of the website, such as headers, navigation bars, content sections, and footers.
- **Pass props:** Choose at least one functional component to pass props to another applicable component. This will allow you to render specific information tailored to your webpage.
- **Styling:** Apply custom CSS rules and possibly Bootstrap components to style your website and make it visually similar to the original website.
- **Link to website:** Include the URL of the website you are recreating as a link at the bottom of your webpage, to provide a reference to the original source for comparison.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.



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

CodeSandbox. (2022). *Instant dev environments*. CodeSandbox.

<https://codesandbox.io/docs>

CodeSandbox. (2022). *React – CodeSandbox*. CodeSandbox (Web).

<https://codesandbox.io/s/new>

Create React App. (2020). *Adding a Stylesheet*. Meta Open Source.

<https://create-react-app.dev/docs/adding-a-stylesheet/>

GitHub. (2022). *react-devtools*. GitHub.

<https://github.com/facebook/react/tree/main/packages/react-devtools>

React. (2023). *Quick Start*. React. **<https://react.dev/learn>**

React. (2024). *Glossary of React terms*. Meta Open Source.

<https://reactjs.org/docs/glossary.html>

React. (2022). *<StrictMode>*. React. **<https://react.dev/reference/react/StrictMode>**

React. (2022). *Client React DOM APIs*. React. **<https://react.dev/reference/react-dom/client>**

Reed. (2024). *How to Create a React App in 2024*. freeCodeCamp.

<https://www.freecodecamp.org/news/how-to-create-a-react-app-in-2024/>

React. (2022). *The library for web and native user interfaces*. React. **<https://reactjs.org>**

React Bootstrap. (2024). *React Bootstrap*. React Bootstrap.

<https://react-bootstrap.netlify.app/>

reactstrap. (2022). *Installation*. reactstrap. **<https://reactstrap.github.io/>**

MDN. (2024). *Expressions and operators*. MDN Web Docs.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions and Operators#expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#expressions)

MDN. (2024). *String*. MDN Web Docs.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

MDN. (2024). *Document: getElementById() method*. MDN Web Docs.

<https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById>

Webpack. (2022). *Concepts*. Webpack. <https://webpack.js.org/concepts/>

Robie, J., Texcel Research. (1998). *What is the Document Object Model?* [Table].

W3C. <https://www.w3.org/TR/WD-DOM/introduction.html>