lab 1

HTML

```
<h1>A Blue Heading</h1>
<p>A red paragraph.</p>
```

With Styles

```
<h1 style="color:blue;">A Blue Heading</h1>
<p style="color:red;">A red paragraph.</p>
```

Links

```
<a href="url">link text</a>
```

Images

```
<img src="link or path" alt="Flowers in Chania">
```

Class

```
<div class="city">
  <h2>Kathmandu</h2>
  <p>KTM is the capital of Nepal.</p>
</div>
<style>
.city {
  background-color: tomato;
  color: white;
  border: 2px solid black;
  margin: 20px;
  padding: 20px;
}
</style>
```

ID

```
<h1 id="myHeader">My Header</h1>
<style>
#myHeader {
  background-color: lightblue;
  color: black;
  padding: 40px;
  text-align: center;}
```

```
</style>
<h1 class=’myheader’>My header </h1>
```

Css

```
<style>
.myHeader {
  background-color: lightblue;
  color: black;
  padding: 40px;
  text-align: center;}
</style>
h1 {
background-color: lightblue;
  color: black;
  padding: 40px;
  text-align: center;}
}
<h1 class=’myheader’>My header
          <h2>ajshd</h2>
</h1>
<style>
.myHeader {
  background-color: lightblue;
  color: black;
  padding: 40px;
  text-align: center;}
</style>
Myheader h2 {
background-color: lightblue;
  color: black;
  padding: 40px;
  text-align: center;}
}
Css Selector
.center {
  text-align: center;
  color: red;
}
#para1 {
  text-align: center;
  color: red;
}
P .center {
```

```
  text-align: center;
  color: red;
}
* {
  text-align: center;
  color: blue;
}
h1, h2, p {
  text-align: center;
  color: red;
}
```

JavaScript

Let vs const vs var

```
let x="A";
undefined
let x=0
undefined
var x="A";
VM267:1 Uncaught SyntaxError: Identifier 'x' has already been declared    at <anonymous>:1:1
In case of var, only can be used once

In case of {}
{let y = "a";}
undefined
console.log(y)
Uncaught ReferenceError: y is not defined  at <anonymous>:1:13
{let y = "a"; console.log(y)}
 A
var y=10;
undefined
{var y=11;}
undefined
console.log(y)

var x = 2;    // Allowed
let x = 3;    // Not allowed

{
let x = 2;    // Allowed
let x = 3     // Not allowed
}
```

```
{
let x = 2;    // Allowed
var x = 3     // Not allowed
}
```

Allowed

```
carName = "Volvo";
var carName;
```

Not Allowed

```
carName = "Saab";
let carName = "Volvo";
```

```
const x = 10;
// Here x is 10

{
const x = 2;
// Here x is 2
}

// Here x is 10
```

let and const:

Variables declared using let or const are only accessible within the block in which they are defined.
This means they are not accessible outside that block, making them more predictable and less error-prone compared to var.

```
if (true) {
  let x = 10;
  const y = 20;
  console.log(x); // 10
  console.log(y); // 20
}
console.log(x); // ReferenceError: x is not defined
console.log(y); // ReferenceError: y is not defined
```

var:

Variables declared using var do not have block scope.
They are scoped to the nearest function or,

if declared outside of any function, to the global scope. As a result, var can sometimes lead to unexpected behavior.

```
if (true) {
  var z = 30;
}
console.log(z); // 30
```

Here, z is accessible outside the if block because
var does not respect block scope.

Block Scope in Loops:

When using let or const inside loops (e.g., for, while),
each iteration creates a new scope.
This is particularly useful in asynchronous operations inside loops.

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i); // 0, 1, 2
  }, 1000);
}
```

using var inside the loop would cause the same value to be logged three times due to the function-level scoping of var.

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i); // 3 3 3
  }, 1000);
}
```

With let:

Each iteration of the loop creates a new block scope for the variable i,
so inside the setTimeout, the value of i is captured separately for each iteration.
As a result, after 1 second, console.log(i) will print 0, 1, and 2 in sequence (because let respects the block scope
and the i in each setTimeout call has the correct value for that iteration).

With var:

The variable i is declared with function scope,
not block scope. This means that there's only one i shared across all iterations of the loop.
By the time the setTimeout callbacks are executed (after 1 second), the loop has already finished running,
and the value of i is 3 (since i is incremented to 3 at the end of the loop).

Thus, all the setTimeout callbacks will log the value 3.

Operators

```
let x = 100 + 50;
Datatypes
let length = 16;                              // Number
let lastName = "Johnson";                     // String
let x = {firstName:"John", lastName:"Doe"};    // Object
Function
function mux(p1, p2) {
  return p1 * p2;   // The function returns the product of p1 and p2
}
mux(1,2);
```

Objects

```
const person = {
  firstName: "Sahaj",
  lastName: "Shakya",
  age: 25
};
person.lastName;
person["lastName"];
```

Concat

```
const arr1 = ["BCT", "A"];
const arr2 = ["BCT", "B", "2075"];
const arr3 = arr1.concat(arr2);
```

filter

```
const BCT = [1, 2, 40, 50, 3];
function checkBCTB(no) {
  return no >= 18;

}
const BCTB = BCT.filter(checkBCTB);
```

map

```
const numbers = [65, 44, 12, 4];
function mul(num) {
  return num * 2;
}
```

twice = numbers.map(mul);

In JavaScript, both filter and map are higher-order functions that are used to transform or extract elements from an array. They serve different purposes, and here's a comparison of their functionality:

Array.prototype not modified

filter Method:

Purpose: The filter method is used to filter out elements from an array based on a condition. It returns a new array with only the elements that satisfy the given condition (predicate).

Returns: A new array containing elements that pass the condition. If no elements pass, it returns an empty array.

Original Array: The original array is not modified.

Usage: When you want to extract elements that meet a specific criterion.

```
const numbers = [1, 2, 3, 4, 5];

// Filter out even numbers
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

In this example, filter returns a new array with only the even numbers.

map Method:

Purpose: The map method is used to transform every element in an array according to the given function. It returns a new array where each element is the result of applying the function to the corresponding element in the original array.

Returns: A new array where each element has been transformed (or unchanged if the transformation function doesn't modify it).

Original Array: The original array is not modified.

Usage: When you want to modify the elements of an array (e.g., doubling values, extracting specific properties, etc.).

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(num => num * 2);  // Double each number
console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

to modify and extend the behavior of arrays.
some important methods that modify the array in place:

Original Array: The original array is modified.

1. push():

Adds one or more elements to the end of an array and returns the new length of the array.
Modifies the array in place.

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr); // [1, 2, 3, 4]
```

2. pop():

Removes the last element from an array and returns it.
Modifies the array in place.

```
const arr = [1, 2, 3];
const last = arr.pop();
console.log(last);  // 3
console.log(arr);   // [1, 2]
```

3. shift():

Removes the first element from an array and returns it.
Modifies the array in place.

```
const arr = [1, 2, 3];
const first = arr.shift();
console.log(first); // 1
console.log(arr);   // [2, 3]
```

4. unshift():

Adds one or more elements to the beginning of an array and returns the new length of the array.
Modifies the array in place.

```
const arr = [1, 2, 3];
arr.unshift(0);
console.log(arr); // [0, 1, 2, 3]
```

5. splice():

Changes the contents of an array by removing, replacing, or adding elements at a specific index.
Modifies the array in place.

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 2, 'a', 'b');
console.log(arr); // [1, 2, 'a', 'b', 5]
```

splice() removes 2 elements from index 2 and adds 'a' and 'b'.

6. sort():

Sorts the elements of an array in place, according to a provided comparison function.
Modifies the array in place.

```
const arr = [3, 1, 2];
arr.sort();
console.log(arr); // [1, 2, 3]
```

7. reverse():

Reverses the elements of an array in place.
Modifies the array in place.

```
const arr = [1, 2, 3];
arr.reverse();
console.log(arr); // [3, 2, 1]
```

8. fill():

Changes all elements in an array to a static value, from a start index to an end index.
Modifies the array in place.

```
const arr = [1, 2, 3, 4, 5];
arr.fill(0, 1, 4);
console.log(arr); // [1, 0, 0, 0, 5]
```

9. Slice

```
const worldcup = ["JAPAN", "ARGENTINA", "BRAZIL", "KOREA"];
const eliminated = worldcup.slice(1, 3);
```

The splice() method in JavaScript is a powerful and versatile method used to add, remove, or replace elements in an array. It modifies the array in place, meaning the original array is changed directly, and it does not return a new array.

Syntax of splice():

```
array.splice(start, deleteCount, item1, item2, ...)
```

start (required): The index at which to start changing the array.
If the value is greater than the array's length,

the changes will be made at the end of the array.
deleteCount (optional): The number of elements to remove starting from the start index. If omitted or set to 0, no elements are removed.
item1, item2, ... (optional): The elements to add to the array starting from the start index. If no elements are specified, splice() will just remove elements.

How splice() works:

1. Removing Elements: If you specify a start index and a deleteCount, splice() will remove the specified number of elements starting at that index.

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 2); // Remove 2 elements starting from index 2
console.log(arr);  // [1, 2, 5]
```

In this case, starting at index 2, it removes 2 elements (3 and 4), leaving [1, 2, 5].

2. Adding Elements: If you omit the deleteCount or set it to 0, splice() will insert new elements at the start index.

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 0, 'a', 'b'); // Add 'a' and 'b' starting from index 2
console.log(arr);  // [1, 2, 'a', 'b', 3, 4, 5]
```

Here, splice() adds 'a' and 'b' at index 2, and no elements are removed (deleteCount is 0).

3. Replacing Elements: If you provide both a deleteCount and new items to insert, splice() can remove elements and replace them with the new items.

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 2, 'a', 'b'); // Remove 2 elements and add 'a' and 'b'
console.log(arr);  // [1, 2, 'a', 'b', 5]
```

In this case, splice() removes 2 elements (3 and 4) and replaces them with 'a' and 'b'.

4. Return Value of splice():
splice() returns a new array containing the elements that were removed from the original array. If no elements are removed, it returns an empty array.

```
const arr = [1, 2, 3, 4, 5];
const removedItems = arr.splice(2, 2);  // Removes elements from index 2
console.log(removedItems);  // [3, 4]
console.log(arr);           // [1, 2, 5]
```

SpreadOperator

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];

const numbers = [1, 2, 3, 4, 5, 6];
```

const [one, two, ...rest] = numbers;

allows for expanding or spreading elements from an iterable (like arrays, objects, etc.) into individual elements.
It can be used in various contexts such as

1. For Arrays:

    Spreading elements into a new array:

    You can use the spread operator to copy all elements from one array into another.
    ```
    const arr1 = [1, 2, 3];
    const arr2 = [...arr1, 4, 5];
    console.log(arr2); // [1, 2, 3, 4, 5]
    ```

    Merging arrays:

    The spread operator can combine two arrays into one.
    ```
    const arr1 = [1, 2];
    const arr2 = [3, 4];
    const merged = [...arr1, ...arr2];
    console.log(merged); // [1, 2, 3, 4]
    ```

2. For Objects:

    Copying an object:

    You can create a shallow copy of an object with the spread operator.
    ```
    const obj1 = { name: "Alice", age: 25 };
    const obj2 = { ...obj1 };
    console.log(obj2); // { name: "Alice", age: 25 }
    ```

    Merging objects:

    The spread operator can combine the properties of two objects into one.
    ```
    const obj1 = { name: "Alice" };
    const obj2 = { age: 25 };
    const mergedObj = { ...obj1, ...obj2 };
    console.log(mergedObj); // { name: "Alice", age: 25 }
    ```

    Overriding properties:

    If two objects have properties with the same key, the last object spread will override the previous one.
    ```
    const obj1 = { name: "Alice", age: 25 };
    const obj2 = { age: 30 };
    const mergedObj = { ...obj1, ...obj2 };
    ```

```
console.log(mergedObj); // { name: "Alice", age: 30 }
```

3. In Function Calls:

Passing elements of an array as function arguments:

The spread operator can be used to pass elements of an array as separate arguments in a function.

```
const numbers = [1, 2, 3];
function sum(a, b, c) {
  return a + b + c;
}
console.log(sum(...numbers)); // 6
```

4. In Destructuring:

The spread operator can be used in destructuring to collect remaining elements.

```
const arr = [1, 2, 3, 4, 5];
const [first, second, ...rest] = arr;
console.log(first);  // 1
console.log(second); // 2
console.log(rest);   // [3, 4, 5]
```

Similarly, with objects:

```
const obj = { name: "Alice", age: 25, city: "Wonderland" };
const { name, ...rest } = obj;
console.log(name);   // Alice
console.log(rest);   // { age: 25, city: "Wonderland" }
```

React

Create app

```
npm create vite@latest my-react-app --template react
cd my-react-app
npm i
npm run dev
```

install package

```
npm install bootstrap
```

To add the package

on main app

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

The useState hook is a fundamental part of React's functional components.
It allows you to add state to your functional components, which is essential for managing dynamic content in a React app.

Syntax of useState:

const [state, setState] = useState(initialState);

state: The current value of the state.

setState: A function that you use to update the state.

initialState: The initial value of the state, which can be a primitive value (like a number or string), an array, an object, or even a function.

Basic Usage of useState

How useState works.

```
import React, { useState } from 'react';

function Counter() {
  // Declare a state variable 'count' initialized to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}

export default Counter;
```

useState(0) initializes the count state variable to 0.

setCount(count + 1) updates the count state each time the button is clicked.

In React, className is used to apply CSS classes to elements, instead of the traditional class attribute used in plain HTML. This is because class is a reserved keyword in JavaScript, and React uses className to avoid conflicts.

Using className in JSX:

When you define JSX in React, you use className to assign CSS classes to elements, just like you would in plain HTML

using className to style a div element in React:

```
import React from 'react';

function MyComponent() {
  return (
    <div className="container">
      <h1 className="title">Hello, React!</h1>
      <p className="description">This is a simple example using className in React.</p>
    </div>
  );
}

export default MyComponent;
```

Here

- The div has a className of container.
- The h1 has a className of title.
- The p has a className of description.

These classes are mapped to corresponding styles in your CSS file.

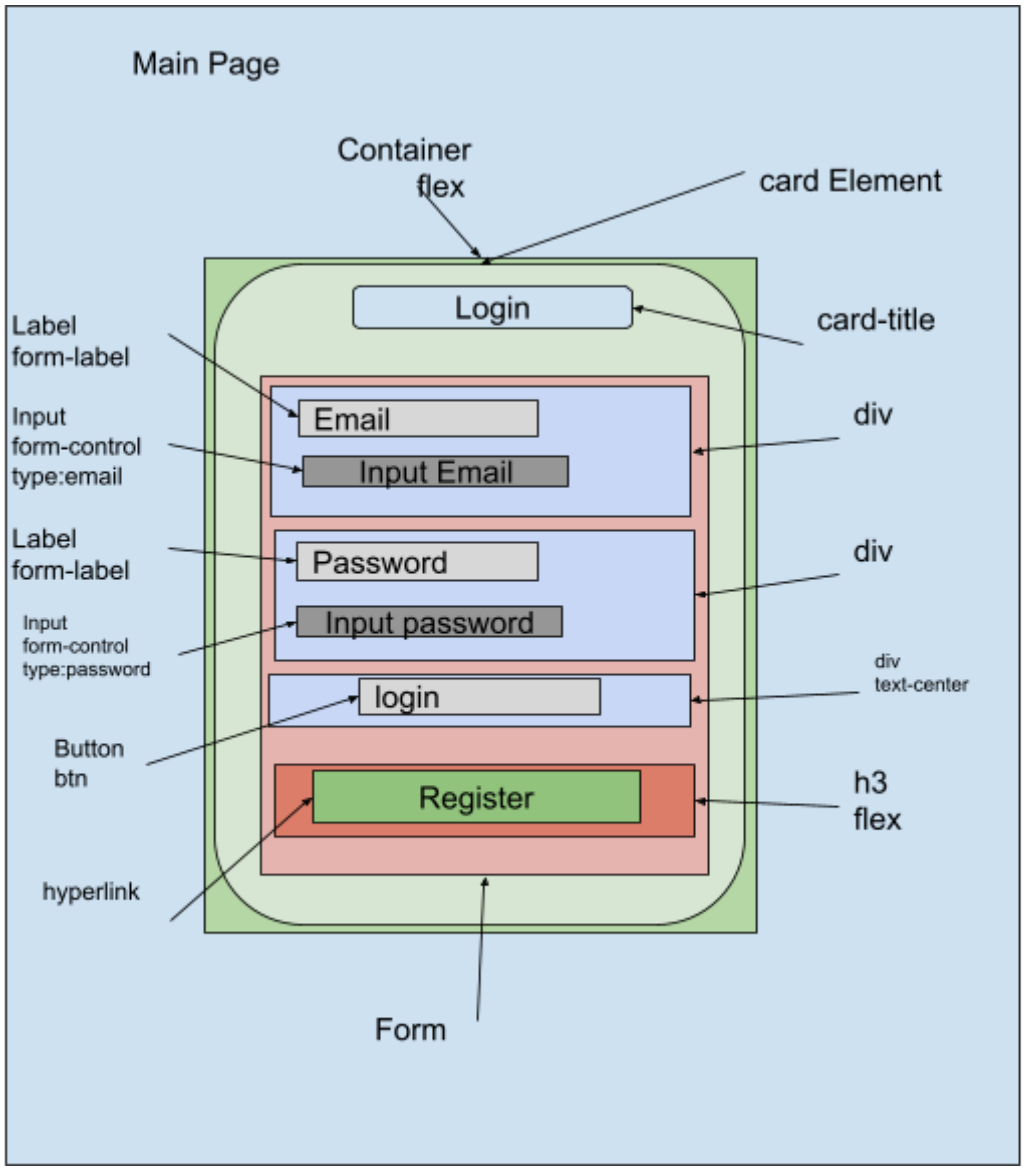
```
/* File: App.css or MyComponent.css */
/* Style for the container div */
.container {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
  background-color: #f4f4f4;
  border-radius: 8px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

/* Style for the title (h1) */
.title {
  font-size: 2rem;
```

```
  font-weight: bold;
  color: #333;
  text-align: center;
  margin-bottom: 15px;
}

/* Style for the description (p) */
.description {
  font-size: 1.2rem;
  color: #555;
  line-height: 1.6;
  text-align: center;
}
```

Lab 1



```
<div className="container d-flex justify-content-center align-items-center vh-100">
    <div className="card p-4">
      <h3 className="card-title text-center mb-4">Login</h3>
      <form>
        <div className="mb-3">
          <label htmlFor="username" className="form-label">
            Email
          </label>
          <input
            type="email"
            className="form-control"
            id="email"
            value={email}
            onChange={handleEmailChange}
          />
        </div>
        <div className="mb-3">
          <label htmlFor="password" className="form-label">
            Password
          </label>
          <input
            type="password"
            className="form-control"
            id="password"
            value={password}
            onChange={handlePasswordChange}
          />
        </div>
        <div className="text-center">
          <button type="button" className="btn btn-primary" onClick={handleLogin}>
            Login
          </button>
        </div>
                            <h3 className='d-flex justify-content-center align-items-center'><a
href="/register">Register</a></h3>
      </form>
    </div>
  </div>
```

breakdown

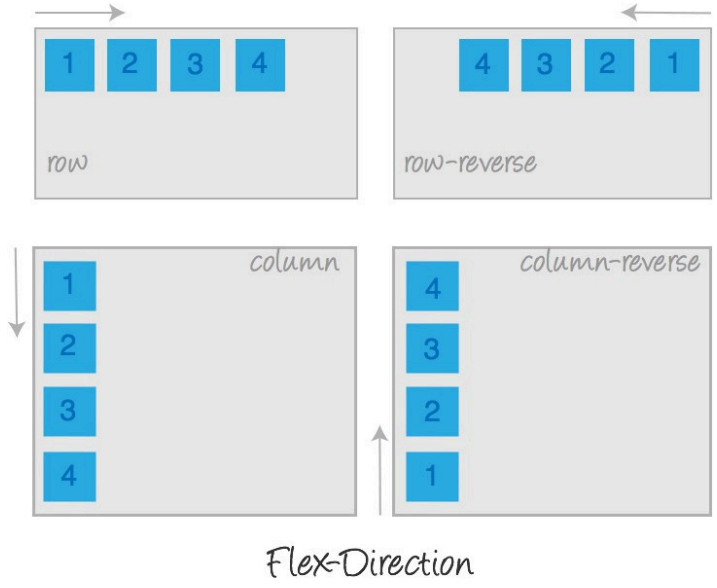
Structure wise Breakdown:

1. Main div - Container

container: This is a Bootstrap class that provides a responsive container with fixed-width, which adapts to screen size.
css alternative

```
/* File: App.css or your specific CSS file */
/* Style for the container */
.container {
  max-width: 1200px; /* Set the maximum width of the container */
  margin: 0 auto; /* Center the container horizontally */
  padding: 20px; /* Add padding inside the container */
  background-color: #f8f9fa; /* Light background color */
  border-radius: 8px; /* Optional: Add rounded corners to the container */
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1); /* Optional: Add shadow for a subtle card effect */
  overflow: hidden; /* Prevent content overflow */
}
/* Media query for responsiveness - adjusting the container width on small screens */
@media (max-width: 768px) {
  .container {
    padding: 15px; /* Reduce padding on smaller screens */
  }
}
```

d-flex: This is a Bootstrap utility class that applies the Flexbox layout to this div, allowing its child elements to be easily aligned and positioned.



justify-content-center: This Flexbox utility horizontally centers the child elements.
align-items-center: This Flexbox utility vertically centers the child elements.
vh-100: This class sets the height of the container to be 100% of the viewport height. This ensures that the form is vertically centered in the entire screen.

2. Card Wrapper for Form

card: A Bootstrap class that gives the div a card-like appearance, including a border, padding, and rounded corners.
p-4: This Bootstrap utility adds padding to the card, ensuring the content inside has some space from the edges (4 units of padding).

3. Login Title

<h3 className="card-title text-center mb-4">Login</h3>
card-title: A Bootstrap class that applies the default styling for card titles.
text-center: Centers the text horizontally inside the h3 tag.
mb-4: A Bootstrap margin utility that adds margin-bottom (spacing) of 4 units to give space between the title and the next form field.

4. Form Element

<form>
This represents the HTML form where the user will input their login credentials (email and password).

5. Email Input Field

```
<div className="mb-3">
  <label htmlFor="username" className="form-label">Email</label>
  <input
    type="email"
    className="form-control"
    id="email"
    value={email}
    onChange={handleEmailChange}
  />
</div>
```

mb-3: A Bootstrap class that applies margin-bottom to create space between this field and the next one.
form-label: A Bootstrap class that styles the label (in this case, "Email").

type="email": Specifies that this input should accept email addresses and will automatically validate the email format (e.g., user@example.com).
className="form-control": This Bootstrap class styles the input field with a standard look for form elements (including borders and padding).
value={email}: The value of the email input is tied to the React state variable email, making it a controlled component. This allows React to manage the form's state.
onChange={handleEmailChange}: This event handler updates the email state variable whenever the user types in the email input field.

6. Password Input Field

```
<div className="mb-3">
  <label htmlFor="password" className="form-label">Password</label>
  <input
    type="password"
    className="form-control"
    id="password"
    value={password}
    onChange={handlePasswordChange}
  />
</div>
```

mb-3: This Bootstrap class adds margin-bottom to space this field from the next one.
type="password": Specifies that this input field is for passwords, meaning the text entered will be hidden.
value={password}: Similar to the email input, the value of this field is tied to the password state variable, making it controlled by React.
onChange={handlePasswordChange}: This event handler updates the password state variable as the user types in the password field.

7. Login Button

```
<div className="text-center">
  <button type="button" className="btn btn-primary" onClick={handleLogin}>
    Login
  </button>
</div>
```

text-center: This class centers the button horizontally.
btn btn-primary: These are Bootstrap classes that style the button, giving it a primary button appearance (usually blue).
onClick={handleLogin}: This is the click event handler for the button. When the button is clicked, the handleLogin function is executed, which should contain the login logic (like making an API request with the email and password).

8. Register Link

```
<h3 className="d-flex justify-content-center align-items-center">
  <a href="/register">Register</a>
</h3>
```

d-flex justify-content-center align-items-center: These Bootstrap Flexbox utility classes center the <h3> tag both horizontally and vertically.

Register: This is a link that directs the user to the registration page (/register). If the user doesn't have an account, they can click this link to navigate to the registration form.

9. React State and Handlers:

For this code to work, you'll need to have React state for email and password and functions like handleEmailChange, handlePasswordChange, and handleLogin to manage the form input and login process.