Learning React From Project

Form Handling In React

1. `Login form simple`

```
const [loginField, setLoginField] = useState({ email: "", password: "" });
const [loginFieldErr, setLoginFieldErr] = useState({
email: "",password: "",});

const handleChange = (e) => {
const { name, value } = e.target;
setLoginField({ ...loginField, [name]: value });
let checkLogin = LoginValid(name, value);
setLoginFieldErr({ ...loginFieldErr, [name]: checkLogin });

};
const onLogin = async (event) => {
event.preventDefault(); //
for (let key in loginField) {
const checkLogin = LoginValid(key, loginField[key]);
setLoginFieldErr({ ...loginFieldErr, [key]: checkLogin });
if (checkLogin !== "") return false;
}
const LoginData = {
email: loginField.email,
password: loginField.password,
};
const result = await logIn(LoginData);
if (result.status) {
const { token } = result;
localStorage.setItem("jwtToken", token);
const userPermissions = await getPermissions().unwrap();
if (userPermissions) {
dispatch(setCredentials(userPermissions));
toast.success(result.message);
navigate("/dashboard");
}
} else {
toast.dismiss();
toast.error(result.message);
}
};
<form action="#!">
<div className="form-group">
<label htmlFor="email" className="sr-only">
```

```
Email
</label>
<input
type="email"
name="email"
id="email"
className="form-control"
placeholder="Email address"
onChange={handleChange}
value={loginField.email}
/>
<span className="text-danger">
{loginFieldErr.email}
</span>
</div>
<div className="form-group mb-4">
<label htmlFor="password" className="sr-only">
Password
</label>
<input
type="password"
name="password"
id="password"
className="form-control"
placeholder="***********"
onChange={handleChange}
value={loginField.password}
/>
<span className="text-danger">
{loginFieldErr.password}
</span>
</div>
<button
type="submit"
className="btn btn1 btn-block login-btn mb-4 w-100 p-2"
onClick={onLogin}
>
Log In
</button>
</form>
```

What is `event.preventDefault()`?

It's a method that prevents the default behavior of an event from happening.
In the case of a `<form>`:
The default behavior is to reload the page when the form is submitted.
`event.preventDefault()` stops that from happening.

`setLoginField({ ...loginField, [name]: value });`

why the square brackets [] are used.

Dynamic Property Names (aka Computed Property Names)

```
const obj = {
  name: "John"
}
const key = "email";
const value = "john@example.com";
const obj = {
  key: value
};
{ key: "john@example.com" }  // ❌ Not what you want
const obj = {
  [key]: value
};
{ email: "john@example.com" }  // ✅ Correct

setLoginField({ ...loginField, [name]: value });
// Then update or add the field where the key is the value of name
```

`<input>` kya hota hai React ya HTML mein?

`<input>` ek HTML tag hai jo **user se data lene ke liye** use hota hai — form ke andar ya kisi bhi jagah pe.

Input kya-kya le sakta hai? (Important Attributes)

Attribute	Kya karta hai
`type`	Ye batata hai ki input kis tarah ka hoga (text, email, password, etc.)
`name`	Iska use hota hai data identify karne ke liye — jaise `email`, `username`
`id`	Unique ID hota hai, mostly label se link karne ke liye
`value`	React state se bind hota hai (controlled input)
`onChange`	Function call hota hai jab bhi user kuch type kare
`placeholder`	Input box ke andar likha aata hai jab tak user kuch type na kare
`className`	CSS styling ke liye classes lagane ke kaam aata hai
`required`	Form submit hone se pehle validate karta hai (optional nahi)
`disabled`	Agar lagao to input disable ho jaata hai
`readonly`	Sirf padh sakte ho, change nahi kar sakte
`autoFocus`	Page load hote hi cursor us input pe aajata hai

`<form action="#!">` ka breakdown:

```
<form action="#!">
  <!-- form fields -->
</form>
```

`action` attribute kya karta hai?

- `action` batata hai **form submit hone ke baad data kahan bhejna hai**.
- Normal HTML forms mein yeh URL hota hai (jahan form ka data submit hota hai).
  `<form action="/submit-form">`

Toh `action="#!"` ka kya matlab?

- `#!` ek trick hai.
- Yeh basically **form ko kisi server pe data bhejne se rokta hai**.
- Yeh mostly frontend projects mein use hota hai jab:
  - Hum khud JavaScript ya React se `onSubmit` handle kar rahe ho.
  - Hum nahi chahte ke page reload ho ya kahi redirect ho.

Toh kab use karein?

```
<form action="#!" onSubmit={handleSubmit}>
```

Important

Centralized API Endpoint Management in React

```
/src
  └── /constants
        ├── api.js              # All API endpoint paths
        ├── baseUrl.js          # All base URLs from .env
        └── enum.js             # Any enums or labels used across the app
```

`baseUrl.js`

```
// src/constants/baseUrl.js
const baseUrl = process.env.REACT_APP_BASE_URL;
const socketUrl = process.env.REACT_APP_SOCKET_IO_URL;
const imageUrl = process.env.REACT_APP_IMAGE_URL;
export { baseUrl, socketUrl, imageUrl };
```

```
import { baseUrl } from "./BaseUrl";
export const loginApi = baseUrl + "/login";
export const changePasswordApi = baseUrl + "/change-password";
```

`Button.jsx`

```
import React from "react";
import Spinner from "react-bootstrap/Spinner";

// Pre-defined button variants
const variants = {
  primary: "btn-primary",
  inverse: "btn-outline-primary",
  danger: "btn-danger",
};

// Pre-defined button sizes
const sizes = {
  sm: "btn-sm",
  md: "",
  lg: "btn-lg",
};
// Reusable Button Component
const Button = ({
  type = "button",            // HTML button type
  className = "",             // Additional classes if needed
  variant = "primary",        // Controls color style
  size = "md",                // Controls size
  isLoading = false,          // Show loading spinner
  children,                   // Text or content inside the button
  ...props                    // Any other props (e.g. onClick, name)
}) => {
  // Combine all classes into one string
  const combinedClassNames = [
    "btn",
    "flex justify-center items-center disabled:opacity-50 disabled:cursor-
not-allowed shadow-sm focus:outline-none hover:opacity-80",
    variants[variant],
    sizes[size],
    className,
  ].join(" ");

  return (
    <button
      type={type}
      disabled={isLoading}
      className={combinedClassNames}
      {...props}
    >
      {/* Show loading spinner if isLoading is true */}
      {isLoading && (
        <Spinner
          as="span"
          animation="border"
          size="sm"
          role="status"
          aria-hidden="true"
        />
      )}
      {/* Button text or content */}
      <span className="mx-2">{children}</span>
    </button>
  );
};

// Give a display name for debugging or dev tools
Button.displayName = "Button";

// Export the component
export default Button;
```

What is this `CustomModal`?
It's just a **reusable UI component** in React.
It **is NOT** called a *higher order component* (HOC).
This is a **simple functional component** that shows a **popup/modal window** when `isOpen` is true
What this code is doing?

1. **Props it accepts:**
   - `isOpen`: whether the modal is visible or hidden.
   - `onClose`: function to close the modal.
   - `title`: title at the top.
   - `children`: the inner content (you can pass anything inside the modal).
   - `footer`: bottom section (eg. buttons like Save, Cancel).
   - `maxWidth`: how wide the modal is (default `600px`).
2. **What happens inside?**
   - When `isOpen` becomes `true`, it **disables body scroll** (`overflow: hidden` on `<body>`) so you can't scroll the background.
   - When modal is closed, **scrolling is enabled back** (`overflow: auto`).
3. **Styling:**
   - There's an overlay (dark background behind the modal).
   - Modal box is white, centered, and responsive.
   - There's a close `×` button in top right corner.
4. **Events:**

- Clicking outside the modal closes it (because overlay has `onClick={onClose}`).
- Clicking *inside* modal (content) **doesn't close** it (`e.stopPropagation()`).

⚠ **Warning**

Is it a Higher Order Component (HOC)?
× No, it's not an HOC.
HOC = a **function** that **takes a component** and **returns a new component**.

```
import React, { useEffect } from "react";
//  Importing React and useEffect hook to control side-effects (like
blocking scroll).

const CustomModal = ({ isOpen, onClose, title, children, footer, maxWidth
= "600px" }) => {
// Declaring the CustomModal component that accepts props.

  useEffect(() => {
    // Whenever isOpen changes, this runs.
    if (isOpen) {
      document.body.style.overflow = "hidden";
      //  Disable background scroll when modal open.
    } else {
      document.body.style.overflow = "auto";
      //  Enable scroll back when modal closed.
    }
    return () => {
      document.body.style.overflow = "auto";
      //  Clean up if component unmounts (always set scroll back to
normal).
    };
  }, [isOpen]);

  const overlayStyle = {
    display: isOpen ? "flex" : "none",
    //  Show overlay only if modal is open.
    position: "fixed",
    top: 0, left: 0,
    width: "100%", height: "100%",
    background: "rgba(0, 0, 0, 0.5)",
    //  Dark semi-transparent background.
    justifyContent: "center", alignItems: "center",
    zIndex: 1000,
    overflowY: "auto",
  };
  const modalStyle = {
    background: "#fff",
    borderRadius: "5px",
    boxShadow: "0 0 10px rgba(0, 0, 0, 0.2)",
    maxWidth: maxWidth,
    width: "100%",
    textAlign: "left",
    position: "relative",
    maxHeight: "90vh",
    overflow: "inherit",
    scrollbarWidth: "none",
    //  Modal box design.
  };

  const headerStyle = {
    background: "#FFFFFF",
    color: "#08223e",
    padding: "10px",
    borderBottom: "1px solid rgba(120, 130, 140, 0.13)",
    borderTopLeftRadius: "5px",
    borderTopRightRadius: "5px",
    fontSize: "18px",
    // Top bar styling with title inside.
  };

  const closeBtnStyle = {
    cursor: "pointer",
    position: "absolute",
    top: "0px",
    right: "10px",
    fontSize: "30px",
    color: "rgba(120, 130, 140, 3)",
    // '×' close button styling inside modal.
  };

  const footerStyle = {
    borderTop: "1px solid #ddd",
    padding: "10px",
    textAlign: "right",
    //  Footer area styling for bottom buttons or actions.
  };

  return (
    <div style={overlayStyle} onClick={onClose} className="main-model">
    {/* Main background overlay, clicking here triggers onClose */}
      <div style={modalStyle} onClick={(e) => e.stopPropagation()}>
      {/* Modal box, clicking inside will NOT close modal */}
        <div style={headerStyle} className="main-model-header">
          {/*  Header with title and close button */}
          <b>{title}</b>
          <span style={closeBtnStyle} onClick={onClose}>
            &times; {/*  × sign to close the modal */}
          </span>
        </div>
        <div style={{ padding: "20px" }} className="main-model-body">
          {/*  Main body content where children are rendered */}
          {children}
        </div>
        <div style={footerStyle} className="main-model-footer">
          {/* Footer area where footer content (like Save/Cancel buttons)
comes */}
          {footer}
        </div>
      </div>
    </div>
  );
};

export default CustomModal;
//  Exporting the component to use it anywhere else.
```

how use it

```
    <CustomModal
          isOpen={show}
          onClose={handleClose}
          title={t("User Balances")}
          maxWidth="800px"
          footer={
            <Button
              className="btn btn1"
              variant={"primary"}
              isLoading={false}
              onClick={handleSubmit}
            >
              {t("Update")}
            </Button>
          }
        >
```

advanced React form Handling

```
 const [broker, setBroker] = useState({
    name: { value: "", required: true },
    email: { value: "", required: true },
    mobile_number: { value: "", required: true },
    admin: { value: "", required: user?.role === "Admin" ? false : true },
    password: { value: "", required: true },
  });

  // Error defined here
    const [brokerError, setBrokerError] = useState({
    name: "",
    email: "",
    mobile_number: "",
    admin: "",
    password: "",
  });
    const handleChange = (e) => {
    const { name, value } = e.target;
    setBroker((prev) => {
      return { ...prev, [name]: { ...prev[name], value } };
    });

    const validationError = ValidateInputFields(
      name,
      value,
      broker[name].required
    );

    if (name === "adminId") {
      refetchPermissions();
    }
    setBrokerError((prevError) => {
      return { ...prevError, [name]: validationError };
    });
  };
```

Line Breakdown:

```
setBroker((prev) => {
  return {
    ...prev,
    [name]: {
      ...prev[name],
      value,
    },
  };
});
```

And your current `broker` is:

```
{
  name: { value: "Raj", required: true },
  email: { value: "", required: true },
  // other fields
}
 ...prev
 // Copies the entire `broker` object:
 {
  name: { value: "Raj", required: true },
  email: { value: "", required: true },
   ...
}
// `[name]` → `"email"`
[email]: {
   ...prev[email], // This gives { value: "", required: true }
  value: "test@example.com"
}
//So it becomes:

email: {
  value: "test@example.com", // updated value
  required: true             // preserved!
}
// Final Result:
{
  name: { value: "Raj", required: true },
  email: { value: "test@example.com", required: true },
   ...
}
// Only email.value changed. Everything else remains intact.
```

Why use this structure?
Because you're storing both value and required inside each field, and you want to:

- Only change the value
- Not overwrite required

Handle submit form

```
 const handleSubmit = async (e) => {
    e.preventDefault();
    const handleUpdateOrAdd = async () => {
      const values = Object.fromEntries(
        Object.entries(broker).map(([key, { value }]) => [key, value])
      );
      for (let key in broker) {
        const validationError = ValidateInputFields(
          key,
          broker[key].value,
          broker[key].required
        );
        setBrokerError((prevError) => ({
          ...prevError,
          [key]: validationError,
        }));
        if (validationError) return;
      }

      const isAdminPermissionValid = values.permissions
        .flatMap(({ permissions }) => permissions.flatMap(Object.values))
        .some((val) => !isNaN(val) && val === "1");

      if (!isAdminPermissionValid) {
        handleToast(t, {
          data: { status: false, message: t("At least one permission
  needed") },
        });
        return;
      }

      const mutationResult = isEdit
        ? await updateBroker({ id: data.id, ...values })
        : await addBroker(values);

      handleToast(t, mutationResult);
      if (mutationResult?.data?.status) {
        refetch();
        if (!isNew) {
          refetchPermissions();
        }
        const tabIndex = isNew ? 1 : 0;
        getNewUser(
          {
            ...mutationResult.data.data,
            id: mutationResult.data.data.id,
          },
          tabIndex
        );
      }
    };

    handleUpdateOrAdd();
  };
```

Most Important Part is

```
    const values = Object.fromEntries(
      Object.entries(broker).map(([key, { value }]) => [key, value])
    );
    for (let key in broker) {
      const validationError = ValidateInputFields(
        key,
        broker[key].value,
        broker[key].required
      );
      setBrokerError((prevError) => ({
        ...prevError,
        [key]: validationError,
      }));
      if (validationError) return;
    }
```

`Object.entries(broker).map(([key, { value }]) => [key, value])`
Suppose your broker object looks like this:

```
const broker = {
  name: { value: "Suraj", required: true },
  email: { value: "suraj@example.com", required: true },
  mobile: { value: "9876543210", required: true },
};
```

Step 1: `Object.entries(broker)`
This converts the `broker` object into an array of key-value pairs:

```
[
  ["name",  { value: "Suraj", required: true }],
  ["email", { value: "suraj@example.com", required: true }],
  ["mobile", { value: "9876543210", required: true }],
]
// Each item is an array: [key, object].
```

Step 2: `.map(([key, { value }]) => [key, value])`
We are using array destructuring and object destructuring together.
Here's what's happening in this line:

- `key` will be `"name"`, `"email"`, or `"mobile"`
- `{ value }` is object destructuring — it extracts only the `value` field from the object.

```
["name", { value: "Suraj", required: true }] → ["name", "Suraj"]
["email", { value: "suraj@example.com", required: true }] → ["email",
"suraj@example.com"]
```

```
["mobile", { value: "9876543210", required: true }] → ["mobile",
"9876543210"]
// Final result:
[
  ["name", "Suraj"],
  ["email", "suraj@example.com"],
  ["mobile", "9876543210"]
]
```

Step 3: Use `Object.fromEntries(...)`
Now this array is passed to `Object.fromEntries`, which converts it back to an object:

```
{
  name: "Suraj",
  email: "suraj@example.com",
  mobile: "9876543210"
}
```

This is the final cleaned object — ideal for sending to a server or API without the `required` part.

how Download React Table in `.csv` Format

```
import {
  MaterialReactTable,useMaterialReactTable} from "material-react-table";
  import { Box, Button, Tooltip, IconButton } from "@mui/material";
  import FileDownloadIcon from "@mui/icons-material/FileDownload";
  import EditIcon from "@mui/icons-material/Edit";
  import DeleteIcon from "@mui/icons-material/Delete";
  import { mkConfig, generateCsv, download } from "export-to-csv";// csv
const GenericTable = ({ columnsData, rowsData }) => {
  // Setup CSV config
  const csvConfig = mkConfig({
    fieldSeparator: ",",
    useKeysAsHeaders: true,
  });
  // Memoize the columns
const columns = useMemo(
    () => [
      {
        header: t("Serial No."),
        size: 20,
        Cell: ({ row }) => <span>{row.index + 1}</span>,
      },
      {
        accessorKey: "name",
        header: t("Name"),
      },
      {
        accessorKey: "status",
        header: t("Status"),
        Cell: ({ row: { original } }) => <span>{original.status}</span>,
      },
    ],
    [data]
  );
  const memoizedColumns = useMemo(() => columnsData, [columnsData]);
  // Setup the table
  const table = useMaterialReactTable({
    columns: memoizedColumns,
    data: rowsData,
    enableRowSelection: true,
    enableStickyHeader: true,
    getRowId: (row) => row.id,
    renderTopToolbarCustomActions: ({ table }) => (
      <Box sx={{ display: "flex", gap: "1rem", p: "8px" }}>
        <Button
          onClick={() => {
            const csv = generateCsv(csvConfig)(memoizedData);
            download(csvConfig)(csv);
          }}
          variant="contained"
        >
          Export All
        </Button>
        <Button
          disabled={!table.getIsSomeRowsSelected()}
          onClick={() => {
            const selectedRows = table.getSelectedRowModel().rows.map(r =>
r.original);
            const csv = generateCsv(csvConfig)(selectedRows);
            download(csvConfig)(csv);
          }}
          variant="outlined"
        >
          Export Selected
        </Button>
      </Box>
    ),
  });
  return <MaterialReactTable table={table} />;
};
export default GenericTable;
```

`const memoizedColumns = useMemo(() => columnsData, [columnsData]);`
Only recalculates columns if `columnsData` changes. Otherwise reuses old columns.

`const memoizedData = useMemo(() => rowsData, [rowsData]);` Only recalculates data if rowsData changes.
Using `memoizedColumns`, `memoizedData` inside `useMaterialReactTable`.

Create Own Custom Hooks

`useDropdown.js` (Custom Hook)

```
// useDropdown.js
import { useState, useEffect, useRef } from "react";

export const useDropdown = (initialState = false) => {
  const [isOpen, setIsOpen] = useState(initialState);
  const ref = useRef(null);

  const handleClickOutside = (event) => {
    if (ref.current && !ref.current.contains(event.target)) {
      setIsOpen(false);
    }
  };

  useEffect(() => {
    if (isOpen) {
      document.addEventListener("mousedown", handleClickOutside);
    } else {
      document.removeEventListener("mousedown", handleClickOutside);
    }

    return () => {
      document.removeEventListener("mousedown", handleClickOutside);
    };
  }, [isOpen]);

  return { isOpen, setIsOpen, ref };
};
```

```
// App.js
import React from "react";
import DropdownComponent from "./DropdownComponent";

function App() {
  return <DropdownComponent />;
}

export default App;
```

How You can handle

```
const [oldPassword, setOldPassword] = useState("");
 const [oldPasswordErr, setOldPasswordErr] = useState("");
 const [newPassword, setNewPassword] = useState("");
 const [btnnDisable, setBtnnDisable] = useState(false);
 const [confirmPassword, setConfirmPassword] = useState("");
 const [type, setType] = useState("password");
 const [newType, setNewType] = useState("password");
 const [confirmType, setConfirmType] = useState("password");
 const [passwordShow, setPasswordShow] = useState({
   eye: "bi-eye-slash",
   type: "password",
 });
 const [passwordFiledErr, setPasswordFiledErr] = useState({
   newPassword: "",
   confirmPassword: "",
 });
 const [resetPassword] = useChangePasswordMutation();

 const handleChangeInput = (e) => {
   const { name, value } = e.target;

   if (name === "oldPassword") {
     if (!value) {
       setOldPasswordErr("This field is required");
     } else {
       setOldPasswordErr("");
     }
     setOldPassword(value);
   }

   if (name === "newPassword") {
     if (!value) {
       setPasswordFiledErr({
         ...passwordFiledErr,
         newPassword: "This field is required",
       });
     }
     setNewPassword(value);
   }
   if (name === "confirmPassword") {
     if (!value) {
       setPasswordFiledErr({
         ...passwordFiledErr,
         confirmPassword: "This field is required",
       });
     } else {
       setPasswordFiledErr({
         ...passwordFiledErr,
         confirmPassword: "",
       });
     }
     setConfirmPassword(value);
   }
     const changePassword = async (e) => {
   e.preventDefault();
   if (!newPassword && !confirmPassword && !oldPassword) {
     setPasswordFiledErr({
       ...passwordFiledErr,
       newPassword: "This field is required",
       confirmPassword: "This field is required",
     });
     setOldPasswordErr("This field is required");
     return false;
   } else {
     setPasswordFiledErr({
       ...passwordFiledErr,
       newPassword: "",
       confirmPassword: "",
     });
   }

   if (newPassword !== confirmPassword) {
     setPasswordFiledErr({
       ...passwordFiledErr,
       confirmPassword: "Password does not match",
     });
     return;
   }

   const ForgotData = {
     oldPassword: oldPassword,
     newPassword: newPassword,
     confirmPassword: confirmPassword,
   };
   const changePass = await resetPassword(ForgotData);
   const resp = changePass.data;
   console.log("changePass", resp);
   if (resp.status === false) {
     toast.error(t(resp.message));
   } else {
     toast.success(t(resp.message));
     localStorage.clear();
     setTimeout(() => {
       window.location.href = "/login";
     }, 3000);
   }
 };

 const showcurrentPassword = () => {
   if (type === "password") {
     setPasswordShow({ eye: "bi-eye", type: "text" });
     setType("text");
   } else {
     setPasswordShow({ eye: "bi-eye-slash", type: "password" });
     setType("password");
   }
 };

 const showNewPassword = () => {
   if (newType === "password") {
     setNewType("text");
   } else {
     setNewType("password");
   }
 };

 const showConfirmPassword = () => {
   if (confirmType === "password") {
     setConfirmType("text");
   } else {
     setConfirmType("password");
   }
 };
 return(
 <div>
                 <div className="form-group">
                   <label className="fw-500">{t("Old Password")}
</label>
                   <div className="eye_pass position-relative">
                     <input
                       className="form-control"
                       name="oldPassword"
                       id="oldPassword"
                       placeholder={t("Enter Old Password")}
                       onChange={handleChangeInput}
                       value={oldPassword}
                       type={type}
                     />
                     <span
                       className="password__show position-absolute
eye1"

                       onClick={showcurrentPassword}
                     >
                       <i className={`bi ${passwordShow.eye}`}></i>
                     </span>
                   </div>
                   <span style={{ color: "red" }}>{t(oldPasswordErr)}
</span>
                 </div>
```

```
                      <div className="form-group">
                        <label className="fw-500">{t("New Password")}
</label>
                        <div className="eye_pass position-relative">
                          <input
                            className="form-control"
                            name="newPassword"
                            id="newPassword"
                            placeholder={t("Enter New Password")}
                            onChange={handleChangeInput}
                            value={newPassword}
                            type={newType}
                          />
                          <span
                            className="password__show position-absolute
eye1"
                            onClick={showNewPassword}
                          >
                            {newType === "password" ? (
                              <i className="bi bi-eye-slash"></i>
                            ) : (
                              <i className="bi bi-eye"></i>
                            )}
                          </span>
                        </div>
                        <span style={{ color: "red" }}>
                          {t(passwordFiledErr.newPassword)}
                        </span>
                      </div>
                      <div className="form-group">
                        <label className="fw-500">{t("Confirm New
Password")}</label>
                        <div className="eye_pass position-relative">
                          <input
                            className="form-control"
                            name="confirmPassword"
                            id="confirmPassword"
                            placeholder={t("Enter Confirm Password")}
                            onChange={handleChangeInput}
                            value={confirmPassword}
                            type={confirmType}
                          />
                          <span
                            className="password__show position-absolute
eye1"
                            onClick={showConfirmPassword}
                          >
                            {confirmType === "password" ? (
                              <i className="bi bi-eye-slash"></i>
                            ) : (
                              <i className="bi bi-eye"></i>
                            )}
                          </span>
                        </div>
                        <span style={{ color: "red" }}>
                          {t(passwordFiledErr.confirmPassword)}
                        </span>
                      </div>
                      <div className="form-group">
                        <button
                          onClick={changePassword}
                          className="btn w100px btn_man "
                          disabled={btnnDisable}
                        >
                          {t("Change Password")}
                        </button>
                      </div>
                    </div>
                  </form>
                </div>
              </div>
    )
```

ffghd