USER REGISTRATION

Introduction

This document explains how to implement Create, Read, Update, and Delete (CRUD) operations in a React application using local storage. This approach allows us to manage user data without using a backend database.

Components Used

useState: Manages component state.

useEffect: Retrieves and updates local storage data.

localStorage: Stores user data persistently in the browser.

Code Breakdown

1. State Initialization

jsx

```
const [users, setUsers] = useState([]);

const [data, setData] = useState({ name: "", email: "", password: "" });

const [editIndex, setEditIndex] = useState(null);
```

- users stores the list of registered users.

- data holds the form input values.

- editIndex tracks the index of the user being edited.

2. Load Data from Local Storage

jsx

```
useEffect(() => {
  const storedUsers = JSON.parse(localStorage.getItem("users")) || [];
```

```
  setUsers(storedUsers);
}, []);
```

- This effect runs once when the component loads.
- It fetches stored user data from local storage and updates the state.

3. Save Data to Local Storage

jsx

```
useEffect(() => {
  localStorage.setItem("users", JSON.stringify(users));
}, [users]);
```

- Whenever users state changes, the new list is stored in local storage.

4. Handling Form Input Changes

jsx

```
const handleChange = (e) => {
  setData({ ...data, [e.target.name]: e.target.value });
};
```

- Updates data state when the user types in the input fields.

5. Handling Form Submission

jsx

```
const handleSubmit = (e) => {
  e.preventDefault();
  if (editIndex !== null) {
```

```
    const updatedUsers = [...users];
    updatedUsers[editIndex] = data;
    setUsers(updatedUsers);
    setEditIndex(null);
  } else {
    setUsers([...users, data]);
  }
  setData({ name: "", email: "", password: "" });
};
```

- If a user is being edited (editIndex !== null), update their details.
- Otherwise, add a new user.
- Clears the input fields after submission.

6. Editing a User

jsx

```
const handleEdit = (index) => {
  setData(users[index]);
  setEditIndex(index);
};
```

- Loads user data into the form for editing.
- Sets editIndex to the selected user's index.

7. Deleting a User

jsx

```
const handleDelete = (index) => {
```

```
  const updatedUsers = users.filter((_, i) => i !== index);
  setUsers(updatedUsers);
};
```

- Removes the selected user from the users list.
- Updates local storage with the new list.

User Interface (UI) Explanation

Form Section

jsx

```
<form className='form' onSubmit={handleSubmit}>
  <h1>{editIndex !== null ? "Edit User" : "Registration Form"}</h1>
  <input type="text" name="name" placeholder="Enter the name" value={data.name} onChange={handleChange} required />
  <input type="email" name="email" placeholder="Enter the email" value={data.email} onChange={handleChange} required />
  <input type="password" name="password" placeholder="Password" value={data.password} onChange={handleChange} required />
  <button type="submit">{editIndex !== null ? "Update" : "Register"}</button>
</form>
```

- Displays "Edit User" if updating a user, otherwise "Registration Form".
- Button text changes to "Update" when editing.

User List Section

jsx

```
<ul>
  {users.map((user, index) => (
    <li key={index}>
```

```
      {user.name} - {user.email}
      <button onClick={() => handleEdit(index)}>Edit</button>
      <button onClick={() => handleDelete(index)}>Delete</button>
    </li>
  ))}
</ul>
```

- Displays the list of registered users.

- Includes "Edit" and "Delete" buttons for each user.

Conclusion

This implementation enables users to:

1. Create a new user.

2. Read existing users from local storage.

3. Update user details.

4. Delete a user from the list.