**1. Crear el proyecto con Vite+React**

Primero, debes generar un nuevo proyecto con Vite y configurar el entorno de desarrollo.

1. En la terminal, crea el proyecto:

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

cd my-patient-app

1. Instala las dependencias necesarias:

npm install

1. Instala **Bootstrap**:

npm install Bootstrap

1. Importa Bootstrap en tu proyecto. Abre el archivo main.jsx y agrega:

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

1. Ejecuta el servidor de desarrollo:

npm run dev

### 2. ****Estructura del proyecto****

Organiza tu proyecto para que sea escalable y fácil de mantener. Aquí hay una estructura inicial recomendada:

src/

│

├── components/

│ ├── PatientForm.jsx // Formulario para agregar pacientes

│ ├── PatientList.jsx // Listado de pacientes

│ ├── PatientItem.jsx // Cada paciente dentro del listado

│

├── pages/

│ ├── HomePage.jsx // Página principal que contendrá el formulario y el listado

│

├── App.jsx // Componente raíz

└── main.jsx // Archivo de entrada principal

### 3. ****Formulario para nuevos pacientes****

Crea un formulario controlado que capture los datos del paciente (nombre, edad, diagnóstico).

Ejemplo básico de PatientForm.jsx:

import { useState } from 'react';

const PatientForm = ({ onAddPatient }) => {

const [name, setName] = useState('');

const [age, setAge] = useState('');

const [diagnosis, setDiagnosis] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

onAddPatient({ name, age, diagnosis });

setName('');

setAge('');

setDiagnosis('');

};

return (

<form onSubmit={handleSubmit} className="mb-3">

<div className="mb-3">

<label className="form-label">Name</label>

<input type="text" className="form-control" value={name} onChange={(e) => setName(e.target.value)} required />

</div>

<div className="mb-3">

<label className="form-label">Age</label>

<input type="number" className="form-control" value={age} onChange={(e) => setAge(e.target.value)} required />

</div>

<div className="mb-3">

<label className="form-label">Diagnosis</label>

<input type="text" className="form-control" value={diagnosis} onChange={(e) => setDiagnosis(e.target.value)} required />

</div>

<button type="submit" className="btn btn-primary">Add Patient</button>

</form>

);

};

export default PatientForm;

### 4. ****Listado de pacientes registrados****

Muestra los pacientes en un componente separado. Cada paciente puede tener una opción para editar.

Ejemplo de PatientList.jsx:

import PatientItem from './PatientItem';

const PatientList = ({ patients, onEditPatient }) => {

return (

<div>

<h3>Registered Patients</h3>

<ul className="list-group">

{patients.map((patient, index) => (

<PatientItem key={index} patient={patient} onEditPatient={() => onEditPatient(index)} />

))}

</ul>

</div>

);

};

export default PatientList;

Y el componente PatientItem.jsx:

const PatientItem = ({ patient, onEditPatient }) => {

return (

<li className="list-group-item d-flex justify-content-between align-items-center">

<div>

<h5>{patient.name}</h5>

<p>Age: {patient.age}</p>

<p>Diagnosis: {patient.diagnosis}</p>

</div>

<button className="btn btn-secondary" onClick={onEditPatient}>Edit</button>

</li>

);

};

export default PatientItem;

### 5. ****Editar o actualizar información del paciente****

Puedes controlar la edición utilizando el mismo formulario. Cuando el usuario haga clic en "Editar", precargas los datos en el formulario.

Dentro de App.jsx puedes gestionar la lógica:

import { useState } from 'react';

import PatientForm from './components/PatientForm';

import PatientList from './components/PatientList';

const App = () => {

const [patients, setPatients] = useState([]);

const [editingIndex, setEditingIndex] = useState(null);

const handleAddPatient = (newPatient) => {

if (editingIndex !== null) {

const updatedPatients = patients.map((patient, index) =>

index === editingIndex ? newPatient : patient

);

setPatients(updatedPatients);

setEditingIndex(null);

} else {

setPatients([...patients, newPatient]);

}

};

const handleEditPatient = (index) => {

setEditingIndex(index);

};

return (

<div className="container">

<h1>Patient Management</h1>

<PatientForm

onAddPatient={handleAddPatient}

editingPatient={editingIndex !== null ? patients[editingIndex] : null}

/>

<PatientList patients={patients} onEditPatient={handleEditPatient} />

</div>

);

};

export default App;

**6. Diseño responsive**

Bootstrap ya incluye clases para hacer que tu aplicación sea responsive. Al utilizar clases como container, form-control, btn, y las grillas (col, row), tu aplicación será automáticamente adaptable a diferentes pantallas.

**7. Buenas prácticas**

* **Componentización**: Divide los componentes en piezas pequeñas y reutilizables.
* **Estado local vs global**: Usa useState para manejar el estado local de los formularios. Si el estado se vuelve más complejo, considera usar una librería como **Redux** o **Context API**.
* **PropTypes**: Usa prop-types para definir los tipos de las props que cada componente recibe.
* **Validación de formularios**: Puedes mejorar la validación de datos usando librerías como **Formik** o **Yup**.
* **Manejo de errores**: Implementa feedback en caso de errores al agregar o editar pacientes.

Para guardar la información del formulario en el **Local Storage** y asegurarte de que los datos persistan entre recargas de la página, puedes integrar esta funcionalidad en tu proyecto de Vite+React. A continuación, te muestro cómo hacerlo paso a paso:

### 1. ****Guardar en Local Storage****

Para guardar y cargar los pacientes desde Local Storage, podemos utilizar los hooks useEffect y useState. La idea es que cuando la lista de pacientes cambie, la guardemos en el Local Storage, y cuando la aplicación cargue por primera vez, recuperemos los datos desde allí.

Modifica el archivo App.jsx de la siguiente manera:

### App.jsx (con Local Storage)

import { useState, useEffect } from 'react';

import PatientForm from './components/PatientForm';

import PatientList from './components/PatientList';

const App = () => {

const [patients, setPatients] = useState([]);

const [editingIndex, setEditingIndex] = useState(null);

// Cargar pacientes desde el Local Storage cuando la app se inicia

useEffect(() => {

const storedPatients = localStorage.getItem('patients');

if (storedPatients) {

setPatients(JSON.parse(storedPatients));

}

}, []);

// Guardar pacientes en Local Storage cada vez que cambie la lista de pacientes

useEffect(() => {

localStorage.setItem('patients', JSON.stringify(patients));

}, [patients]);

const handleAddPatient = (newPatient) => {

if (editingIndex !== null) {

const updatedPatients = patients.map((patient, index) =>

index === editingIndex ? newPatient : patient

);

setPatients(updatedPatients);

setEditingIndex(null);

} else {

setPatients([...patients, newPatient]);

}

};

const handleEditPatient = (index) => {

setEditingIndex(index);

};

return (

<div className="container">

<h1>Patient Management</h1>

<PatientForm

onAddPatient={handleAddPatient}

editingPatient={editingIndex !== null ? patients[editingIndex] : null}

/>

<PatientList patients={patients} onEditPatient={handleEditPatient} />

</div>

);

};

export default App;

**2. Explicación de los cambios**

1. **useEffect para cargar desde Local Storage**:
   * Al cargar la aplicación, utilizamos useEffect para verificar si hay pacientes guardados en el Local Storage. Si los encontramos, los convertimos de formato JSON a un arreglo de objetos JavaScript y los almacenamos en el estado patients.
2. **useEffect para guardar en Local Storage**:
   * Cada vez que la lista de patients cambie (ya sea porque se agrega, edita o elimina un paciente), guardamos el nuevo estado en el Local Storage usando localStorage.setItem.

**3. Modificaciones adicionales**

Si quieres que el formulario cargue los datos del paciente a editar directamente en los campos del formulario, actualiza el componente PatientForm de la siguiente manera para manejar la edición de pacientes:

**PatientForm.jsx (modificado para manejar edición)**

import { useState, useEffect } from 'react';

const PatientForm = ({ onAddPatient, editingPatient }) => {

const [name, setName] = useState('');

const [age, setAge] = useState('');

const [diagnosis, setDiagnosis] = useState('');

// Cargar los datos del paciente en el formulario cuando se está editando

useEffect(() => {

if (editingPatient) {

setName(editingPatient.name);

setAge(editingPatient.age);

setDiagnosis(editingPatient.diagnosis);

}

}, [editingPatient]);

const handleSubmit = (e) => {

e.preventDefault();

onAddPatient({ name, age, diagnosis });

setName('');

setAge('');

setDiagnosis('');

};

return (

<form onSubmit={handleSubmit} className="mb-3">

<div className="mb-3">

<label className="form-label">Name</label>

<input

type="text"

className="form-control"

value={name}

onChange={(e) => setName(e.target.value)}

required

/>

</div>

<div className="mb-3">

<label className="form-label">Age</label>

<input

type="number"

className="form-control"

value={age}

onChange={(e) => setAge(e.target.value)}

required

/>

</div>

<div className="mb-3">

<label className="form-label">Diagnosis</label>

<input

type="text"

className="form-control"

value={diagnosis}

onChange={(e) => setDiagnosis(e.target.value)}

required

/>

</div>

<button type="submit" className="btn btn-primary">

{editingPatient ? 'Update Patient' : 'Add Patient'}

</button>

</form>

);

};

export default PatientForm;

**4. Pruebas**

1. Agrega un paciente en el formulario y verifica que, al recargar la página, los pacientes agregados sigan estando en la lista.
2. Verifica que la funcionalidad de editar también funcione correctamente, y que después de actualizar un paciente, los cambios se guarden en el Local Storage.

**5. Consideraciones adicionales**

* **Eliminar pacientes**: Si también necesitas agregar la opción de eliminar pacientes, puedes agregar un botón "Eliminar" en el componente PatientItem y actualizar el estado de patients filtrando los elementos.
* **Validación avanzada**: Si quieres mejorar la validación del formulario o agregar mensajes de error, puedes usar librerías como **Formik** y **Yup**.

Con esta configuración, tu aplicación guardará los datos en **Local Storage** y los mantendrá disponibles entre sesiones, lo cual es útil para aplicaciones pequeñas o para prototipos que no requieren un backend completo.