



# CalTrack

Project Documentation - Mario Sulé Domínguez



<https://github.com/mariosulee/CalTrack.git>

**CalTrack** **Calorie Tracker**  
an App by Mario Sulé



RESTART APP

**Category:**  
Exercise

**Activity:**  
Tennis match

**Calories:**  
700

Record Exercise

**Calories Overview**

690  
Cal consumed



650  
Cal burned

40  
Balance

**Food & Exercise**



Food

Apple  
120 Calories





Exercise

Running 10km  
650 Calories



Food

Chocolate Cake  
445 Calories



CalTrack - Mario Sulé Domínguez. All rights reserved © 2025

## **0. SUMMARY**

The developed web application is an interactive calorie tracker that allows users to log activities in two distinct categories: food and exercise. Each entry is stored in a dynamic history, while the application automatically calculates calories consumed, calories burned, and the overall balance, providing an intuitive and visually clear user experience.

The interface is built with TailwindCSS, delivering a modern, responsive design that adapts to both mobile and desktop screens. Additionally, React Hooks are used to optimize performance, ensuring that calculations are updated only when necessary.

The project incorporates a local storage system to maintain the user's history across sessions, and a reset button that allows all activities to be cleared quickly and safely. This approach combines usability and efficiency, offering a practical tool for the daily tracking of diet and physical exercise.

## 1. IMPLEMENTATION OF THE FORM COMPONENT

Firstly, the header of the application is designed using TailWindCSS, where the name and the author are displayed. Next, the components folder is created, which will host the Form.tsx component, where the main form of the application is designed. The form's label will contain category, and there are only two: one for meals, and another for activities. That is, one category to add calories consumed, and another for calories burned.

The data folder is created to store the two types of category data, identified by an id and a string, and the Categoria type is declared in the new types folder.

In the form, 4 different elements are created using <div>:

- The first is a <select> for the category, food or exercise.
- The second is an <input> for the activity name.
- The third is another <input> for the number of calories consumed or burned.
- The last is a submit input to save the values.

```
export default function Form(){
  <form className="space-y-5 bg-white shadow-xl p-10 rounded-lg">
    { /* CATEGORIA */ }
    <div className="grid grid-cols-1 gap-3">
      <label htmlFor="category" className="font-[Inter] font-bold">Category:</label> { /*htmlfor se usa para asociar la etiq
      <select className="border bg-white border-slate-300 p-2 rounded-lg w-full" id="category">
        {categories.map( category => (
          <option key={category.id} value={category.id}>
            {category.name}
          </option>
        ))}
      </select>
    </div>

    { /* NOMBRE */ }
    <div className="grid grid-cols-1 gap-3 mt-8">
      <label htmlFor="activity" className="font-[Inter] font-bold">Activity:</label>
      <input id="activity" type="text" className="border bg-white border-slate-300 p-2 rounded-lg"
        placeholder="e.g. Tennis match, Dinner at a restaurant, 10km running..."/>
    </div>
  </form>
}
```

```
    { /* NUMERO DE CALORIAS */ }
    <div className="grid grid-cols-1 gap-3 mt-8">
      <label htmlFor="calories" className="font-[Inter] font-bold">Calories:</label>
      <input id="calories" type="number" className="border bg-white border-slate-300 p-2 rounded-lg"
        placeholder="Calories consumed or burned"/>
    </div>

    { /* SUBMIT */ }
    <input type="submit" className="mt-5 bg-gray-900 hover:bg-lime-700 w-full p-2 font-bold text-white font-[Inter] curs
      value="Save"/>
  </form>
</>
```

Next, a state will be created to store what the user types in the form. Since each form field depends on the others, it will be better to use an object state called activity.

```
export default function Form(){  
  
  const [activity, setActivity]=useState({  
    category:'',  
    name:'',  
    calories:0,  
  })  
  
  return(  

```

To handle changes in the inputs, an onChange must be placed in the state and in each form element. The onChange works similarly to an addEventListener. I will use the onChange with the handleChange function, which will read what the user has entered.

```
const handleChange= (e)=>{  
  setActivity({  
    ...activity,  
    [e.target.id]:e.target.value  
    // es el id del elemento q ha cambiado (el input con id name, category, o calories)  
    // : es el valor nuevo q escribe el usuario  
  })  
}
```

Subsequently, the type of the event e must also be specified. Basically, what is done here is updating the activity state by detecting which input has changed using its id, and saving the new value with value, updating only that specific property of the object.

Since when typing in the form the numbers for the category and calories are converted to strings, I created a new Activity type in types.ts and enforced that type in the state declaration. Also, in the handleChange function, I check with isNumberField whether the user is typing in the category or calories fields, and if so, I convert the parameter to a number using the + operator before the event.

To validate that all the form fields are filled before pressing the button, the isValidActivity function is created. In it, the state parameters are extracted and true is returned if they are not empty — meaning they have been filled in. The disabled attribute must also be controlled in the submit button.

```
const isValidActivity=() =>{  
  const{name, calories}=activity //SE EXTRAEN LAS PROPIEDADES DEL ESTADO activity  
  return name.trim() !=='' && calories >0  
}  
  
return(  

```

Next, the submit button is handled to store the entered data using the **handleSubmit** function, which is used in the **onSubmit** parameter of the <form>.

## 2. REDUCER

So far, a custom hook had been implemented to manage all these functions. That worked fine, but for more complex states we can use a **Reducer**, which is a way to manage state in a separate file using the **useReducer** hook.

To incorporate the Reducer into our application, I create the **reducers** folder and inside it the **activityReducer.ts** file (it's like a custom hook). Before implementing the actions, the skeleton of this file would look like this:

```
import type { Activity } from "../types/types"

export type ActivityActions={ //type q va a describir lo que pasa en activityReducer
}

type ActivityState={
  activities:Activity[] //array del tipo Activity definido en types.ts
}

export const initialState: ActivityState= { // el estado inicial es un objeto
  activities: [] //el estado inicial no tiene actividades y a lo largo del día el usuario va metiendo y se van almacenando
}

export const activityReducer= ( // conecta el state con las acciones
  state: ActivityState =initialState,
  action: ActivityActions
) => {
}
```

- The Activity type (category, name, calories) is imported.
- Then, ActivityActions is defined to specify the types of actions the reducer can handle.
- The ActivityState type is defined, which will represent the state managed by the reducer. In my case, it is an object containing an array of type Activity.
- The **initialState** of the defined type is initialized. Each time the user adds something, it will be stored in the activities array.
- The main reducer function is **activityReducer**, which receives the current state and the action AS PARAMETERS, and finally returns a new, updated state.

Then, I define the save-activity action, which will have a payload object called newActivity of type Activity. I use this action in the main reducer function as follows:

```
export type ActivityActions={ //type q va a describir lo que pasa en activityReducer
  type: 'save-activity', payload: {newActivity: Activity} // el payload va a ser un objeto que lo llamo newActivity
}
```

```
export const activityReducer= ( // conecta el state con las acciones
  state: ActivityState =initialState,
  action: ActivityActions
) => {
  if(action.type==='save-activity'){
    //aquí se maneja la logica para actualizar el state
  }
}
```

To execute these actions with the dispatch, I use the useReducer hook in App.tsx. Declared in this main component, the useReducer would look like this:

```
const [state, dispatch] = useReducer(activityReducer, initialState)
```

Then, I pass dispatch as a prop to the Form component, and back in Form.tsx I create a new type for the props of this component, FormProps. We now have access to that dispatch to trigger the action when handleSubmit is called, which looks like this:

```
const handleSubmit= (e:React.FormEvent<HTMLFormElement>) =>{
  e.preventDefault() //se previene la accion por defecto

  //dispatch envia una accion al reducer, diciendole q accion se debe realizar,
  // y los datos q quiere guardar (payload), en este caso el estado activity
  dispatch( {type:"save-activity", payload:{newActivity:activity} })
}
```

In this way, when submitting the form, our components are now fully defined. To write into the reducer's state, I return a copy of the state inside the activityReducer function and create a new array containing all the previous activities plus the new one. The function is like this:

```
//3. LA FUNCION QUE CONECTA EL ESTADO ACTUAL Y LAS ACCIONES
export const activityReducer= (
  state: ActivityState=initialState,
  action: ActivityActions
) => {

  //TODAS ESTAS ACCIONES SERIAN EL DISPATCH
  if(action.type==='save-activity'){
    return{
      ...state, //hago siempre una copia del estado
      activities: [...state.activities, action.payload.newActivity ]
      //creo un nuevo array con todas las actividades anteriores + la nueva
    }
  }

  return state;
}
```

Once that's done, we want the form fields to clear after saving and pressing the button. To achieve this, I go back to the handleSubmit function in the Form component and call setActivity to reset the state back to its initial values:

```
//q se borre todo al pulsar enviar
setActivity({ category:1, name:'', calories:0})
```

To ensure each activity has a unique ID, I modify the initialState in the Form component.

### 3. IMPLEMENTATION OF ACTIVITIES LIST (ActivityList component)

Now that each activity has its own unique ID, we can display them on screen. These activities are stored in the state declared in App.tsx. I created a new section where I'll include a new component called ActivityList. To pass the state from App.tsx to this component, it's done through props by writing state.activities (remember that state is the object managed by the reducer, and activities is a property inside this state, which is the array that stores all activities).

In the ActivityList component, the activities array is iterated over to render the category, name, and calories of each activity, as follows:

```
export type ActivityListProps={
  activities:Activity[]
}

export default function ActivityList( {activities}: ActivityListProps){
  return(
    <>
      <h2 className="text-4xl font-bold text-slate-600 text-center">Food & Exercise</h2>

      {activities.map( (act) => (
        <div key={act.id} className="px-5 py-10 bg-white mt-5 flex justify-between">

          <div className="space-y-2 relative"> {/ESTE AL LADO IZQ POR EL JUSTIFY-BETWEEN */}

            <p>Categoría {act.category}</p>

            <p className="text-2xl font-bold pt-5">{act.name}</p>

            <p className="font-black text-4xl text-lime-600">{act.calories} Calorias</p>
          </div>

          <div> {/ESTE AL LADO DCHO POR EL JUSTIFY-BETWEEN */}

        </div>
      ))}
    </div>
  )
}
```

Then it looks like this:



Now we need to change how the category is displayed so that it shows Food or Exercise in different colors when recording an activity. To do this, I created the categoryName() function, which uses the useMemo hook so that it doesn't execute unless the activities dependency array changes. Inside this hook, another function is returned that goes through the two

different categories declared in the project's data folder. As an example, categoryName(1) would return 'Food'.

```
export default function ActivityList({activities}: ActivityListProps){

  const categoryName=useMemo( () =>
    (cate:Activity['category']) =>
      categories.map(c => c.id===cate ? c.name : '')

    , [activities]) //activities es el array de dependencias

  return(
```

In the return section, I specify with JS and by calling this function that if the category of the object being iterated with map is 1, it returns its name in red, and otherwise in green.

```
{activities.map( (act) => (
  <div key={act.id} className="px-5 py-10 bg-white mt-5 flex justify-between">

    <div className="space-y-2 relative"> {/*ESTE AL LADO IZQ POR EL JUSTIFY-BETWEEN */}

      <p className={`absolute -top-8 -left-8 px-10 py-2 text-white font-bold
        ${act.category===1 ? 'bg-red-500' : 'bg-lime-500'}`}>
        {categoryName(act.category)}
      </p>
    </div>
  </div>
)}
```

#### 4. EDITION OF ACTIVITIES

Once that is done, a button is needed to allow editing the registered activities. For this, Heroicons is installed, which is an SVG icon library for integrating with TailWindCSS and React. A new attribute will need to be created in my reducer, which will be the active ID of the activity being pressed for editing. From there, all the fields (which were already filled) should be populated in the form.

To identify which activity is being clicked on for editing, I go back to the reducer to modify the state, which will now have another property called activeId, and a new action set-ActiveId is created. Since all I need is the ID, the payload will now be an ID.

```
export type ActivityActions=
  { type: 'save-activity', payload: {newActivity: Activity} } | //
  // el payload va a ser un objeto que lo llamo newActivity y va a ser de
  // tipo Activity
  { type: 'set-activeId', payload: {id:Activity['id']}
}

}
```

Inside the activityReducer function, the action is implemented as shown:

```
if(action.type==='set-activeId'){
  return{ //devuelvo copia del state y se actualiza en
    activeId lo que le estoy pasando como payload
  }
}
```



```

        ...state,
        activeId: action.payload.id
    }
}

```

Now that action needs to be triggered with dispatch. I pass it as a prop to the ActivityList component, and in this component I update the props list and destructure dispatch so that I can use it within the component (which is what I have been doing for a while).

Then I can include it in the onClick parameter of the edit <button> as follows:

```

<div className="flex gap-5 items-center"> { /*ESTE AL LADO DCHO POR EL JUSTIFY-BETWEEN */}
  <button onClick={ () => dispatch( {type: "set-activeId", payload: {id:act.id} } )}>
    <PencilSquareIcon className="h-8 w-8" text="gray-800"/>
  </button>
</div>

```

To populate all the form fields for editing an action, first the state is passed as a prop to the Form component (remember that the state contains the array of activities and the activeId). To know when our state has an activeId (that is, when the edit button is pressed), we can use a useEffect. A variable called selectedActivity is then created, which goes through the activities state and identifies the id of the activity selected for editing. After that, the Form component's state is updated using setActivity:

```

// detectar el activeId cuando se pulsa el boton de edicion de una actividad
useEffect( () => {
  if(state.activeId){ //si hay algo
    const selectedActivity=state.activities.filter( a => a.id === state.activeId ) [0] //recorro las actividades y meto
    // en selectedActivity la actividad q tenga el mismo id q la que yo presione en editar
    setActivity(selectedActivity) //seteo con esta actividad el estado creado con useState en este componente
  }
}, [state.activeId])

```

Now, when selecting an activity to edit, its fields already appear filled in the form. However, when saving the changes, it gets added as a new activity. We need to change the logic of the save-activity action to fix this.

```

//TODAS ESTAS ACCIONES SERIAN EL DISPATCH
if(action.type==='save-activity'){

  let updatedActivities:Activity[]=[] //comienza como array vacio

  if(state.activeId){ //compruebo si hay un id activo
    updatedActivities=state.activities.map( ac => //recorro las actividades y si el id de una coincide con el idactivo,
    // reemplazo esa actividad por la nueva version editada
    ac.id===state.activeId ? action.payload.newActivity : ac // si no coincide, dejo la actividad tal cual
    )
  }else{ // si no hay activeId eso significa que estoy agregando una nueva actividad
    updatedActivities=[...state.activities, action.payload.newActivity]
  }

  return{
    ...state, //hago siempre una copia del estado
    activities: updatedActivities, //creo un nuevo array con todas las actividades anteriores + la nueva
    activeId:'' // reiniciar cada vez q haya una nueva actividad para q no reescriba
  }
}

```

## 5. DELETE ACTIVITIES

To delete activities in the application, a new action must be created in the reducer called delete-activity. I only need the ID to delete it, so that will be the payload:

```
{ type: 'delete-activity', payload: {id:Activity['id']}}
```

On the reducer function the action filters those activities that don't have the same id as the selected to delete:

```
if(action.type==='delete-activity'){
  return{
    ...state,
    activities: state.activities.filter( a => a.id !== action.payload.id)
  }
}
```

Now we work with the dispatch of this action. I go back to the ActivityList component and add a new button related to the newly created action.

To display a message when there are no activities, I use a ternary operator that shows either the message or everything that was previously rendered in the return.

To store the activities in LocalStorage, the easiest way is to do it in the main component of the application, synchronizing the activities state by using it as a dependency in a useEffect. The initialState configuration in the reducer must also be changed with a new function.

```
useEffect( () => {
  localStorage.setItem('activities',
JSON.stringify(state.activities))
}, [state.activities])
```

To be able to delete all activities with a single button, I create a new action called restart-app, which doesn't require any payload because it's just a button that clears everything — there's no need to identify any specific activity.

```
{ type: 'restart-app' } ;
```

```
if(action.type==='restart-app'){
  return{
    activities:[],
    activeId:''
  }
}
```

In App.tsx, I place the button and use the onClick to connect it with the dispatch that triggers the implemented action. I also created a small function, canRestartApp, to check if there are any activities so that the button is disabled when there aren't any.

## 6. CALORIE DIFFERENTIAL (CalorieTracker component)

To display the number of calories burned versus consumed, the CalorieTracker component is created. When declaring it in the main component, I only need to pass state.activities as a prop.

To show the calories consumed, the variable caloriesConsumed is created. Using useMemo combined with the reduce function and a ternary operator (to check the category), it returns the sum of consumed calories and recalculates whenever activities change. This is then rendered in a column layout, with its children arranged in a 1-column grid, along with a <span> to wrap small pieces of text inside other elements.

```
export default function CalorieTracker( {activities}: CalorieTrackerProps){  
  
  //contadores  
  const caloriesConsumed=useMemo( () =>  
    activities.reduce( (total, activity) => activity.category===1 ? total+activity.calories : total , 0) // el 0 es el valor inicial  
    , [activities]) // array de dependencias de useMemo  
  
  return(  
    <>  
      <h2 className="text-4xl font-black text-white text-center">Calories Overview</h2>  
  
      <div className="flex flex-col items-center md:flex-row md:justify-between gap-5 mt-5 px-20">  
        <p className="text-orange-500 font-bold rounded-full grid grid-cols-1 gap-3 text-center mt-3">  
          <span className="font-black text-6xl text-orange">{caloriesConsumed}</span>  
          cal consumed  
        </p>  
      </div>  
    </>  
  )  
}
```

For calories burned through exercise, it would be done in the same way. To finish, the calorie differential should be displayed using another function with useMemo.

Finally, the project is built with npm run build.





# CalTrack

Documentación del proyecto - Mario Sulé Domínguez

<https://github.com/mariosulee/CalTrack.git>

CalTrack

Calorie Tracker

an App by Mario Sulé

RESTART APP

Category:

Exercise

Activity:

Tennis match

Calories:

700

Record Exercise

Calories Overview

690

Cal consumed

650

Cal burned

40

Balance

CalTrack

Calorie Tracker

an App by Mario Sulé

RESTART APP

Category:

Exercise

Activity:

Tennis match

Calories:

700

Record Exercise

Calories Overview

690

Cal consumed

650

Cal burned

40

Balance

## **0.RESUMEN DE LA APLICACIÓN WEB DESARROLLADA**

La aplicación web desarrollada es un contador de calorías interactivo que permite a los usuarios registrar actividades en dos categorías distintas: comida y ejercicio. Cada entrada se almacena en un historial dinámico, mientras la aplicación calcula automáticamente las calorías consumidas, las calorías quemadas y el balance total, ofreciendo una experiencia de usuario intuitiva y visualmente clara.

La interfaz está diseñada con TailwindCSS, proporcionando un diseño responsive y moderno que se adapta a móviles y escritorios. Además, se utilizan React Hooks para optimizar el rendimiento, asegurando que los cálculos se actualicen cuando es necesario.

El proyecto integra un sistema de almacenamiento local para mantener el historial del usuario entre sesiones y un botón de reinicio que permite limpiar todas las actividades de manera rápida y segura. Este enfoque combina usabilidad y eficiencia, ofreciendo una herramienta práctica para el seguimiento diario de la dieta y el ejercicio físico.

## 1. CREACIÓN DEL FORMULARIO (componente Form)

Primeramente, se diseña el header usando TailWindCSS de la aplicación, en la cual se encuentra el nombre y el autor. Seguidamente se crea la carpeta components en la que se alojará el componente **Form.tsx**, en el que se diseña el formulario principal de la aplicación. El label del formulario contendrá categoría, y tan solo existen dos: una para las comidas, y otra para las actividades. Es decir, una categoría para sumar calorías ingestadas, y otra para las calorías quemadas.

Se crea la carpeta **data** para almacenar los dos tipos de datos de categoría, identificados por un id y un string, y se declara el tipo **Category** en la nueva carpeta types.

En el formulario, se crean 4 elementos distintos a través de <div>:

- El primero es un <select> para la **categoría**, comida o ejercicio.
- El segundo un <input> para el **nombre** de la actividad.
- El tercero es otro <input> para el **número** de calorías consumidas o quemadas.
- El último es un input de tipo submit para **guardar** los valores.

```
export default function Form(){
  <form className="space-y-5 ■ bg-white shadow-xl p-10 rounded-lg">

    {/ * CATEGORIA */}
    <div className="grid grid-cols-1 gap-3">
      <label htmlFor="category" className="font-[Inter] font-bold">Category:</label> {/ *htmlfor se usa para asociar la etiq
      <select className="border ■ border-slate-300 p-2 rounded-lg w-full ■ bg-white" id="category">

        {categories.map( category => (
          <option key={category.id} value={category.id}>
            {category.name}
          </option>
        ))}

      </select>
    </div>

    {/ * NOMBRE */}
    <div className="grid grid-cols-1 gap-3 mt-8">

      <label htmlFor="activity" className="font-[Inter] font-bold">Activity:</label>
      <input id="activity" type="text" className="border ■ border-slate-300 p-2 rounded-lg"
        placeholder="e.g. Tennis match, Dinner at a restaurant, 10km running..."/>
    </div>


```

```
    {/ * NUMERO DE CALORIAS */}
    <div className="grid grid-cols-1 gap-3 mt-8">
      <label htmlFor="calories" className="font-[Inter] font-bold">Calories:</label>
      <input id="calories" type="number" className="border ■ border-slate-300 p-2 rounded-lg"
        placeholder="Calories consumed or burned"/>
    </div>

    {/ * SUBMIT */}
    <input type="submit" className="mt-5 ■ bg-gray-900 ■ hover:bg-lime-700 w-full p-2 font-bold ■ text-white font-[Inter] curs
      value="Save"/>

  </form>
</>
```

A continuación, se va a crear un state para ir almacenando lo que el usuario va escribiendo en el formulario. Como cada campo del formulario depende del otro, será mejor hacer un estado que sea un objeto, llamado **activity**.

```
export default function Form(){  
  
  const [activity, setActivity]=useState({  
    category:'',  
    name:'',  
    calories:0,  
  })  
  
  return(  

```

Para gestionar los cambios en las entradas, hay que colocar un onChange en cada uno de los elementos del formulario. El onChange se parece a un addEventListener. El onChange lo utilizaré con la función **handleChange** que leerá lo que el usuario ha ingresado:

```
const handleChange= (e)=>{  
  setActivity({  
    ...activity,  
    [e.target.id]:e.target.value  
    // es el id del elemento q ha cambiado (el input con id name, category, o calories)  
    // : es el valor nuevo q escribe el usuario  
  })  
}
```

Posteriormente se debe especificar también el tipo del evento e. Básicamente lo que se hace aquí es actualizar el estado de activity detectando que input ha cambiado con el id y guarda el nuevo valor con value, actualizando solo esa propiedad del objeto.

Ya que al escribir en el form los números de la categoría y calorías se pasan a string, procedo a crear un nuevo tipo **Activity** en types.ts y fuerzo ese tipo en la declaración del estado. También en la función handleChange compruebo con isNumberField si se está escribiendo en la categoría o calorías y en caso de que si convierto el parámetro a un número con el + antes del evento.

Para validar que todos los elementos del formulario estén rellenos para pulsar el botón, se crea la función **isValidActivity**, en la que se extraen los parámetros del estado y se devuelva true si no estan vacios, es decir, si ya han sido rellenos. También se debe controlar en el botón de submit el parámetro *disabled*.

```
const isValidActivity=() =>{  
  const{name, calories}=activity //SE EXTRAEN LAS PROPIEDADES DEL ESTADO activity  
  return name.trim() !=='' && calories >0  
}  
  
return(  

```

A continuación se maneja el botón submit para almacenar lo introducido usando la función **handleSubmit**, la cual se usa en el parámetro onSubmit del <form>.



## 2. REDUCER

Hasta ahora, se había implementado un custom hook para guardar todas estas funciones. Eso estaba bien pero para estados más complejos podemos utilizar un **Reducer**, que es una forma de almacenar un state, siendo un archivo aparte y usando un hook UseReducer.

Para incorporar el Reducer a nuestra aplicación, creo la carpeta **reducers** y en ella el archivo **activityReducer.ts** (es como si fuese un custom hook). Antes de implementar las acciones, el esqueleto de este archivo sería así.

```
import type { Activity } from "../types/types"

export type ActivityActions={ //type q va a describir lo que pasa en activityReducer
}

type ActivityState={
  activities:Activity[] //array del tipo Activity definido en types.ts
}

export const initialState: ActivityState={ // el estado inicial es un objeto
  activities: [] //el estado inicial no tiene actividades y a lo largo del día el usuario va metiendo y se van almacenando
}

export const activityReducer= ( // conecta el state con las acciones
  state: ActivityState =initialState,
  action: ActivityActions
) => {
}
```

- Se importa el tipo *Actividad* (category,name,calories).
- Se pone **ActivityActions** que definirá los tipos de acciones que el reducer puede manejar.
- Se define el tipo *ActivityState* que será el tipo del estado y que manejará el reducer. En mi caso es un objeto que tiene un array del tipo Activity.
- Se inicializa el estado **initialState** del tipo definido arriba. Cada vez que el usuario agregue algo, se irá llenando en el array **actividades**.
- La función principal del reducer es **activityReducer**, que recibe el estado actual y la acción COMO PARÁMETROS, y finalmente devuelve un nuevo estado actualizado.

Defino entonces la acción save-activity, que tendrá un objeto payload llamado newActivity del tipo Actividad. Uso esta acción en la función principal del reducer de la siguiente forma:

```
export type ActivityActions={ //type q va a describir lo que pasa en activityReducer
  type: 'save-activity', payload: {newActivity: Activity} // el payload va a ser un objeto que lo llamo newActivity
}
```

```
export const activityReducer= ( // conecta el state con las acciones
  state: ActivityState =initialState,
  action: ActivityActions
) => {
  if(action.type==='save-activity'){
    //aquí se maneja la logica para actualizar el state
  }
}
```

Para ejecutar estas acciones con el dispatch, utilizo el hook de useReducer en App.tsx. Declarado en este componente principal el useReducer se vería así:

```
const [state, dispatch] = useReducer ( activityReducer, initialState )
```

Entonces paso por props del componente Form el dispatch, y volviendo a Form.tsx creo un nuevo tipo para los props de este componente, *FormProps*. Ya tenemos acceso entonces a ese dispatch para disparar esa acción cuando se llame al handleSubmit, el cual queda así:

```
const handleSubmit= (e:React.FormEvent<HTMLFormElement>) =>{
  e.preventDefault() //se previene la accion por defecto

  //dispatch envia una accion al reducer, diciendole q accion se debe realizar,
  // y los datos q quiere guardar (payload), en este caso el estado activity
  dispatch( {type:"save-activity", payload:{newActivity:activity} })
}
```

De esta forma al pulsar submit del formulario nuestros componentes ya quedan definidos. Para escribir en el state del reducer , retorno en la función activityReducer una copia del state y creo un nuevo array con todas las actividades de antes +la nueva. La función queda:

```
//3. LA FUNCION QUE CONECTA EL ESTADO ACTUAL Y LAS ACCIONES
export const activityReducer= (
  state: ActivityState=initialState,
  action: ActivityActions
) => {

  //TODAS ESTAS ACCIONES SERIAN EL DISPATCH
  if(action.type==='save-activity'){
    return{
      ...state, //hago siempre una copia del estado
      activities: [...state.activities, action.payload.newActivity ]
      //creo un nuevo array con todas las actividades anteriores + la nueva
    }
  }

  return state;
}
```

Hecho eso, queremos que cuando el formulario se guarde y se pulse el botón, se borre cada uno de los parámetros ya introducidos. Para ello, vuelvo al handleSubmit del componente Form para llamar a setActivity y modificar el estado a lo inicial otra vez:

```
//q se borre todo al pulsar enviar
setActivity({ category:1, name:'', calories:0})
```

Para que cada actividad tenga un id único, cambio en el componente Form el initialState.

### 3. CREACIÓN DE LISTA DE ACTIVIDADES (componente **ActivityList**)

Ahora que cada actividad tiene un id único, toca mostrarlas en pantalla. Estas actividades están guardadas entonces en el state declarado en App.tsx. Creo una nueva sección en la que meteré un nuevo componente, llamado **ActivityList**. Para pasarle el state a este componente desde App.tsx se hace por medio de props escribiendo state.activities (recordar que state es el estado que maneja el reducer, y activities es una propiedad dentro de este state, y es el array donde se guardan todas las actividades).

En el componente **ActivityList**, se recorre el array activities para renderizar la categoría, nombre, y calorías de cada una de las actividades, de la siguiente forma:

```
export type ActivityListProps={
  activities:Activity[]
}

export default function ActivityList( {activities}: ActivityListProps){
  return(
    <>
      <h2 className="text-4xl font-bold text-slate-600 text-center">Food & Exercise</h2>

      {activities.map( (act) => (
        <div key={act.id} className="px-5 py-10 bg-white mt-5 flex justify-between">

          <div className="space-y-2 relative"> {/*ESTE AL LADO IZQ POR EL JUSTIFY-BETWEEN */}

            <p>Categoría {act.category}</p>

            <p className="text-2xl font-bold pt-5">{act.name}</p>

            <p className="font-black text-4xl text-lime-600">{act.calories}</p>
          </div>

          <div> {/*ESTE AL LADO DCHO POR EL JUSTIFY-BETWEEN */}

            </div>
        </div>
      ))}
    </>
  )
}
```

Y se vería así:



Ahora se debe cambiar la forma en la que se ve la categoría, para que muestre Food o Exercise y en distintos colores al registrar una actividad. Para ello creo la función categoryName() que usa el hook useMemo para que no se ejecute a menos que cambie el array de dependencias activities. Dentro de ese hook se devuelve otra función que recorre

las dos categorías distintas declaradas en la carpeta data del proyecto. Como ejemplo, `categoryName(1)` devolvería 'Food'.

```
export default function ActivityList({activities}: ActivityListProps){

  const categoryName=useMemo( () =>
    (cate:Activity['category']) =>
      categories.map(c => c.id===cate ? c.name : '')
    , [activities]) //activities es el array de dependencias

  return(
```

En la parte del return, especifico con JS y llamando a esta función que si la categoría del objeto que se recorre con map es 1, se devuelva su nombre en rojo, y sino en verde.

```
{activities.map( (act) => (
  <div key={act.id} className="px-5 py-10 bg-white mt-5 flex justify-between">

    <div className="space-y-2 relative"> /*ESTE AL LADO IZQ POR EL JUSTIFY-BETWEEN */

      <p className={`absolute -top-8 -left-8 px-10 py-2 text-white font-bold
        ${act.category===1 ? 'bg-red-500' : 'bg-lime-500'}`}>
        {categoryName(act.category)}
      </p>
    )
  )
}
```

#### 4. EDICIÓN DE ACTIVIDADES

Hecho eso, se necesita un botón que permita editar las actividades registradas. Para ello se instala **HeroIcons** que es una librería de iconos SVG para integrar con TailWindCSS y React. Se deberá crear un nuevo atributo en mi reducer que será el id activo de la actividad que se presiona en editar. De ahí se deben rellenar todos los campos (que ya estaban rellenos) al formulario.

Para identificar a qué actividad estoy dando click con el editar, vuelvo al reducer para modificar el estado, que ahora tendrá otra propiedad llamada `activeId` y se crea una nueva actividad **set-ActiveId** y como lo único que requiero es el id, el payload ahora será un id.

```
export type ActivityActions=
  { type: 'save-activity', payload: {newActivity: Activity} } |
  { type: 'set-activeId', payload: {id:Activity['id']}
}
```

Dentro de la función `activityReducer` se declarará la acción:

```
if(action.type==='set-activeId') {
  return{
```

```

        ...state,
        activeId: action.payload.id
    // se actualiza en activeId lo que le estoy pasando como payload
    }
}

```

Ahora se debe lanzar esa acción con el dispatch. Se lo paso por props al componente ActivityList y en este componente cambio la lista de props y hago destructuring de dispatch para poder utilizarlo en el componente (lo que llevo haciendo ya tiempo).

Entonces ya lo puedo incluir en el parámetro onClick del <button> de editar de la siguiente forma:

```

<div className="flex gap-5 items-center"> {/*ESTE AL LADO DCHO POR EL JUSTIFY-BETWEEN */}
  <button onClick={ () => dispatch( {type: "set-activeId", payload: {id:act.id} } )}>
    <PencilSquareIcon className="h-8 w-8 text-gray-800"/>
  </button>
</div>

```

Para hacer rellenar todos los campos del formulario para editar una acción, primero se le pasa como prop el state al componente Form (recordar que el state tiene el array de actividades y el activeId). Para saber cuando nuestro state tiene un activeId (es decir, cuando se pulsa el boton de editar), podemos hacerlo con un useEffect. Se crea entonces una variable selectedActivity que recorre el estado de actividades e identifica el id de la seleccionada a editar. Luego se cambia el estado del componente Form con setActivity:

```

// detectar el activeId cuando se pulsa el boton de edicion de una actividad
useEffect( () => {
  if(state.activeId){ //si hay algo
    const selectedActivity=state.activities.filter( a => a.id === state.activeId ) [0] //recorro las actividades y meto
    // en selectedActivity la actividad q tenga el mismo id q la que yo presione en editar
    setActivity(selectedActivity) //seteo con esta actividad el estado creado con useState en este componente
  }
}, [state.activeId])

```

Ahora ya al seleccionar una actividad a editar salen sus campos rellenos en el formulario, pero al guardar el cambio se añade como una actividad nueva y hay que cambiar la lógica de la acción save-activity para solucionar esto:

```

//TODAS ESTAS ACCIONES SERIAN EL DISPATCH
if(action.type==='save-activity'){

  let updatedActivities:Activity[]=[] //comienza como array vacio

  if(state.activeId){ //compruebo si hay un id activo
    updatedActivities=state.activities.map( ac => //recorro las actividades y si el id de una coincide con el idactivo,
    // reemplazo esa actividad por la nueva version editada
    ac.id===state.activeId ? action.payload.newActivity : ac // si no coincide, dejo la actividad tal cual
    )
  }else{ // si no hay activeId eso significa que estoy agregando una nueva actividad
    updatedActivities=[...state.activities, action.payload.newActivity]
  }

  return{
    ...state, //hago siempre una copia del estado
    activities: updatedActivities, //creo un nuevo array con todas las actividades anteriores + la nueva
    activeId:'' // reiniciar cada vez q haya una nueva actividad para q no reescriba
  }
}

```

## 5. ELIMINAR ACTIVIDADES

Para eliminar actividades en la aplicación, se debe hacer una nueva acción en el Reducer llamada **delete-activity**. Solo necesito el id para eliminarla así que eso será el payload:

```
{ type: 'delete-activity', payload: {id:Activity['id']}}
```

Y en la función del reducer la utilizo filtrando todas las actividades que no tengan el id seleccionado:

```
if(action.type==='delete-activity'){
  return{
    ...state,
    activities: state.activities.filter( a => a.id !== action.payload.id)
  }
}
```

Ahora se trabaja con el dispatch de esta acción. Vuelvo al componente ActivityList y pongo un nuevo botón relacionado a la acción recién creada.

Para mostrar un mensaje si no hay actividades, utilizo un operador ternario que muestre un mensaje o todo lo que había antes a renderizar en el return.

Para almacenar las actividades en el LocalStorage, para ello la forma más fácil es en el componente principal de la aplicación y sincronizando el state de actividades usándolo como dependencia en un useEffect. Se debe cambiar también la configuración del initialState del reducer con una nueva función.

```
useEffect( () => {
  localStorage.setItem('activities',
JSON.stringify(state.activities))
}, [state.activities])
```

Para poder eliminar todas las actividades con un solo botón, creo una nueva acción llamada restart-app que no requiere ningún payload porque solo va a ser un botón que borre todo, no hay que identificar ninguna actividad.

```
{ type: 'restart-app' } ;
```

```
if(action.type==='restart-app'){
  return{
    activities:[],
    activeId:''
  }
}
```

En el App.tsx pongo el botón y con el onClick lo relaciono con el dispatch que lance esa acción implementada. También creo una pequeña función canRestartApp que compruebe si hay actividades para que si no hay, no esté habilitado este botón.

## 6. DIFERENCIAL DE CALORÍAS (componente CalorieTracker)

Para mostrar el número de calorías quemadas frente a las consumidas, se crea el componente **CalorieTracker** y al declararlo en el componente principal solo necesito pasarle como prop el state.activities.

Para mostrar las calorías consumidas, se crea la variable caloriesConsumed que con useMemo y combinando la función reducir y un operador ternario (para comprobar la categoría), devuelve la suma de las calorías consumidas, y se ejecuta siempre que activities cambie. Después esto se renderiza con disposición de columna y sus hijos en grid de 1 columna junto con un <span>, que envuelve textos pequeños dentro de otros elementos.

```
export default function CalorieTracker( {activities}: CalorieTrackerProps){

  //contadores
  const caloriesConsumed=useMemo( () =>
    activities.reduce( (total, activity) => activity.category===1 ? total+activity.calories : total , 0) // el 0 es el valor inicial
    , [activities]) // array de dependencias de useMemo

  return(
    <>
      <h2 className="text-4xl font-black text-white text-center">Calories Overview</h2>

      <div className="flex flex-col items-center md:flex-row md:justify-between gap-5 mt-5 px-20">
        <p className="text-orange-500 font-bold rounded-full grid grid-cols-1 gap-3 text-center mt-3">
          <span className="font-black text-6xl text-orange">{caloriesConsumed}</span>
          cal consumed
        </p>
      </div>
    </>
  )
}
```

Para las calorías quemadas con ejercicio, se haría de la misma forma. Para acabar, se debe mostrar el diferencial de calorías con otra función que use useMemo.

Finalmente se construye el proyecto con npm run build.