Clase 6 Ejemplo manejo State

1. Crear el Proyecto

```
ionic start EjemploManejoEstado blank --type=angular
cd EjemploManejoEstado
```

2. Crear un Servicio para el Manejo del Estado

Vamos a crear un servicio que se encargará de manejar el estado centralizado de la aplicación.

En Ionic, un servicio es una clase que se utiliza para organizar y reutilizar lógica compartida a lo largo de la aplicación. Los servicios son especialmente útiles para gestionar datos y lógica que necesitan ser accesibles desde diferentes componentes, sin duplicar código. Esto incluye operaciones como llamadas a APIs, manejo del estado de la aplicación, gestión de datos en memoria, entre otros.

En el contexto del ejemplo que estás revisando, el servicio `EstadoService` se utiliza para manejar el estado centralizado de la aplicación. Este servicio es responsable de gestionar un contador y proporciona métodos para incrementar, decrementar y reiniciar dicho contador. Al utilizar un servicio, puedes mantener un estado sincronizado entre diferentes partes de la aplicación, y cualquier cambio realizado en el estado desde un componente será reflejado automáticamente en todos los otros componentes que dependen de ese estado.

La razón por la que se utiliza un servicio en este ejemplo es precisamente para centralizar la lógica del manejo del estado del contador, evitando tener que manejar esta lógica en cada componente de manera independiente. Esto promueve una mejor organización del código y facilita el mantenimiento y la escalabilidad de la aplicación.

En resumen, los servicios en Ionic son fundamentales para manejar la lógica compartida y el estado de la aplicación de manera eficiente, permitiendo que diferentes componentes puedan interactuar con el mismo conjunto de datos sin duplicar esfuerzos

```
ionic generate service servicios/estado
```

Modifica el archivo `estado.service.ts` para manejar un contador y los métodos necesarios:.....por que modificamos este archivo.

Modificar el archivo `estado.service.ts` es necesario porque es el archivo donde definimos la lógica central para manejar el estado de la aplicación, en este caso, el contador. La edición de este archivo permite que otros componentes de la aplicación accedan y manipulen el estado de manera centralizada y coherente. Aquí te explico más en detalle:

Propósito: `Injectable` es un decorador que marca la clase `EstadoService` como un servicio que puede ser inyectado en otros componentes o servicios a través del mecanismo de inyección de dependencias de Angular. Al hacer esto, Angular puede crear una instancia de este servicio y proveerla a los componentes que la necesiten.

En resumen, estamos importando `Injectable` para hacer que el servicio sea inyectable en otros componentes y `BehaviorSubject` para manejar de manera eficiente el estado reactivo del contador en la aplicación.

Propósito: `BehaviorSubject` es una clase de la librería `rxjs` (Reactive Extensions for JavaScript) que se utiliza para manejar y emitir valores en flujos de datos de forma reactiva. Específicamente, un `BehaviorSubject`:

```
import { Injectable } from '@angular/core';//decorador
import { BehaviorSubject } from 'rxjs';//########

@Injectable({
  providedIn: 'root'
})
export class EstadoService {
  // BehaviorSubject que mantiene el estado actual del contad
or.##############
  // Se inicializa con un valor de 0.
  private contador = new BehaviorSubject<number>(0);

  // Observable al que otros componentes pueden suscribirse p
```

```
ara recibir el valor del contador.
  contadorActual = this.contador.asObservable();//#########

  constructor() {}

  // Método para incrementar el contador en 1.##########
  incrementar() {
    this.contador.next(this.contador.value + 1);
  }

  // Método para decrementar el contador en 1.
  decrementar() {
    this.contador.next(this.contador.value - 1);
  }

  // Método para reiniciar el contador a 0.
  reiniciarContador() {
    this.contador.next(0);
  }
}
```

3. Modificar la Página para Usar el Servicio y Ciclos de Vida

Integraremos el servicio en la página principal, implementando todos los hooks del ciclo de vida.

// Convertimos el BehaviorSubject en un Observable, lo que permite que otros componentes se suscriban para recibir actualizaciones.
// Un Observable es una fuente de datos a la que se pueden suscribir otros componentes,
// permitiéndoles reaccionar automáticamente a los cambios en el estado.

// Creamos un BehaviorSubject que mantiene el estado actual del contador.
// Un BehaviorSubject siempre almacena el valor más reciente, en este caso, se inicializa con 0.

`home.page.ts`:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { EstadoService } from '../servicios/estado.servic
e';//#######

@Component({
  selector: 'app-home',
  templateUrl: './home.page.html',
  styleUrls: ['./home.page.scss'],
})
export class HomePage implements OnInit, OnDestroy {
  // Variable para almacenar el valor actual del contador.
  contador: number=0;//##########

  constructor(private estadoService: EstadoService) {}//####

  // Se ejecuta cuando se inicializa el componente.//########
#
  ngOnInit() {
    console.log('ngOnInit');
    // Nos suscribimos al observable del servicio para recibi
r actualizaciones del contador.
    this.estadoService.contadorActual.subscribe(valor => {
      this.contador = valor;
    });
  }

  // Se ejecuta justo antes de que la vista se muestre.
  ionViewWillEnter() {
    console.log('ionViewWillEnter');
  }

  // Se ejecuta justo después de que la vista se haya mostrad
o.
  ionViewDidEnter() {
    console.log('ionViewDidEnter');
```

```
  }

  // Se ejecuta justo antes de que la vista deje de mostrars
e.
  ionViewWillLeave() {
    console.log('ionViewWillLeave');
  }

  // Se ejecuta justo después de que la vista haya dejado de
mostrarse.
  ionViewDidLeave() {
    console.log('ionViewDidLeave');
  }

  // Se ejecuta cuando el componente se destruye.
  ngOnDestroy() {
    console.log('ngOnDestroy');
  }

  // Método para incrementar el contador llamando al servici
o.
  incrementarContador() {
    this.estadoService.incrementar();
  }

  // Método para decrementar el contador llamando al servici
o.
  decrementarContador() {
    this.estadoService.decrementar();
  }
}
```

`home.page.html`:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Ejemplo de Manejo de Estado
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <!-- Mostrar el valor actual del contador -->
  <h1>Contador: {{ contador }}</h1>
  <!-- Botón para incrementar el contador -->
  <ion-button shape="round" (click)="incrementarContador()">I
ncrementar</ion-button>
  <!-- Botón para decrementar el contador -->
  <ion-button color="tertiary" (click)="decrementarContador
()">Decrementar</ion-button>
</ion-content>
```

4. Ejecutar la Aplicación

Finalmente, ejecuta la aplicación para probarla:

```
ionic serve
```

Explicación de cada uno de los estados:

- **ngOnInit():** Este método se ejecuta cuando se inicializa el componente. Aquí, nos suscribimos al observable del servicio `EstadoService` para recibir y actualizar el valor del contador en la variable `contador`.
- **ionViewWillEnter():** Este método se ejecuta justo antes de que la vista se muestre al usuario. Es útil para realizar configuraciones antes de que la vista sea visible.

- **ionViewDidEnter():** Se ejecuta inmediatamente después de que la vista se ha mostrado completamente. Puede ser usado para iniciar animaciones o cargar datos adicionales.
- **ionViewWillLeave():** Este método se ejecuta justo antes de que la vista comience a desaparecer. Puede ser útil para guardar datos o cancelar operaciones pendientes.
- **ionViewDidLeave():** Este método se ejecuta justo después de que la vista ha dejado de mostrarse. Es un buen lugar para limpiar recursos o cancelar suscripciones.
- **ngOnDestroy():** Se ejecuta cuando el componente está a punto de ser destruido. Es el lugar adecuado para realizar la limpieza de recursos y evitar fugas de memoria.