1)**Creación de proyecto y archivos de componentes y servicio**
Para la creación del proyecto, archivos para componentes y archivo de servicio, ejecutamos primero el siguiente comando, ng new <nombre del proyecto>.
Con el comando 'ng generate component<nombre del componente>' ,generaremos los siguientes componentes: 'formulario-tareas'; 'lista-tareas'; 'barra-tareas'
Como son componentes pequeños, la plantilla HTML la generaremos 'in line', es decir dentro del propio componente, dentro del decorador @Component utilizando la propiedad 'template' para definir la plantilla directamente en línea, como una cadena de texto. Si utilizaremos el HTML externo 'app.component.html' para añadir los selectores de nuestros componentes y renderizarlos allí.
Utilizamos el comando 'ng generate service <nombre>' que creara nuestro archivo de servicios centralizado llamado TareasService.ts.

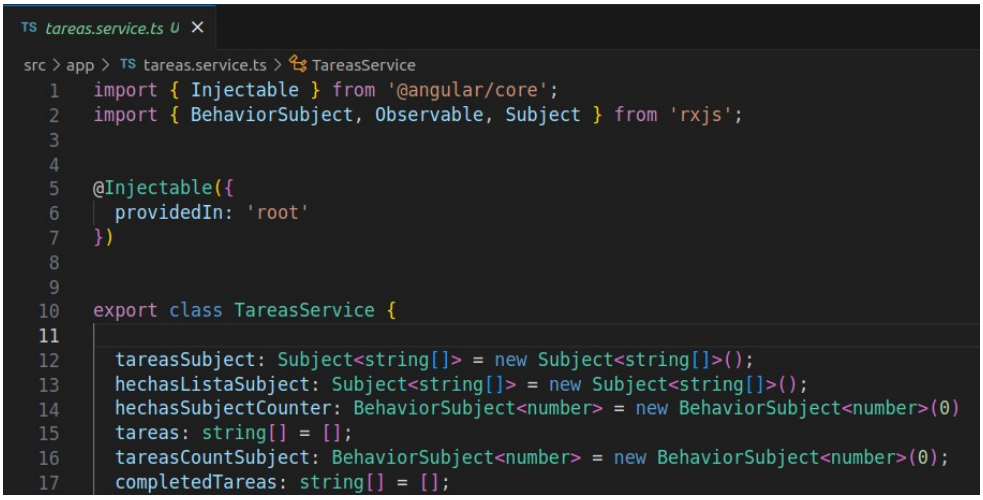
2)**Lógica del servicio e implementación de la librería RxJS**
Este servicio está diseñado para administrar las tareas de nuestra aplicación Angular. Utiliza RxJS para emitir eventos cuando cambian las tareas y proporciona 'observables' para que otros componentes de la aplicación puedan 'suscribirse' a estos cambios y reaccionar en consecuencia.
Hemos utilizado dos tipos de 'observable','Subject' y 'BehaviorSubject' .
Los 'Subject' son observables, pero son más activos y permiten controlar manualmente la emisión de valores. En contraste, los 'observables normales' son más pasivos y siguen un flujo de datos más predefinido.
Los 'Subject' pueden emitir múltiples valores a lo largo del tiempo, mientras que los 'observables normales' generalmente emiten un único valor o una secuencia finita de valores y luego se completan.
Los Subject son considerados "observables calientes" porque pueden emitir valores incluso si no hay 'observadores' suscritos en ese momento. Además, pueden suscribirse a otros 'observables' para recibir valores de datos y también consumen datos de forma directa, es decir, se les puede pasar datos como argumentos. En cambio, los 'observables normales',('observable') son "observables fríos" y solo comienzan a emitir valores cuando un observador se suscribe a ellos. Además, necesitan eventos externos que les provean de datos para emitir valores.

```
///Ejemplo Subject vs Oservable

export class SubjectVsObserver {
  data= 'hola'
getObservableData(){

  const myObs = new Observable <any>(observer=>{observer.next

  (console.log(this.data))})} //necesitamos añadir una funcion para la emision de datos

getSubject(){
const mySubject = new Subject()
mySubject.next(console.log(this.data)) //los datos van directos


}

}
```

Los Subject son "multicasting", lo que significa que todos los subscriptores en los diferentes componentes de nuestra aplicación recibirán los mismos valores, pero los con los 'observable'existe el riesgo de que se les emitan valores diferentes a diferentes subscriptores. Esto se debe a que cada subscriptor realiza llamadas independientes. Por ejemplo, si un 'observable' emite valores procedentes de una función que crea números aleatorios, cada suscriptor recibirá un número diferente, ya que el 'observable' y la función que provee los valores se ejecutarán tantas veces como

suscriptores tengamos. En cambio, el ‘Subject’ emitirá el mismo valor a todos los suscriptores, ya que emiten los valores independientemente del suscriptor.
El ‘BehaviorSubject’ es una variante especializada de Subject en RxJS. A diferencia de un Subject normal, BehaviorSubject tiene un valor inicial y recuerda el último valor emitido. Cuando un observador se suscribe a un BehaviorSubject, recibirá el último valor emitido por el BehaviorSubject, o el valor inicial si aún no se ha emitido ninguno. Después de eso, el observador recibirá cualquier valor emitido por el BehaviorSubject, al igual que con un Subject normal. BehaviorSubject es útil cuando se necesita un valor inicial o cuando los observadores necesitan acceder al último valor emitido incluso si se suscriben después de que se haya emitido.

El primer paso será escribir las importaciones de cabecera import { BehaviorSubject, Observable, Subject } from 'rxjs';Aquí, se están importando algunas clases de RxJS que se usarán en el servicio, nos proporciona las herramientas para trabajar con flujos de datos .
export class TareasService {...}Definimos una clase llamada TareasService que será el servicio encargado de gestionar las tareas.
tareasSubject, hechasListaSubject: Son instancias de la clase ‘Subject’ de ‘RxJS’ que se utilizarán para emitir eventos cuando cambien las listas de tareas, por un lado las pendientes que almacenaremos en el array ‘tareas[]’ y por otro ,la lista de tareas hechas almacenadas en el array completedTareas[]:
hechasSubjectCounter, tareasCountSubject: Son instancias de la clase BehaviorSubject de RxJS. hechasSubjectCounter se usa para mantener un conteo de las tareas ya completadas y tareasCountSubject se usa para mantener un conteo de las tareas que aun están pendientes.



addTareas(tarea: string): void: Este método será ejecutado desde FormularioTareasComponent, recibe como argumento una tarea que recoge el formulario que crearemos en dicho componente, luego usamos .push() que inserta la nueva tarea en el array ‘tareas’, luego notifica a los suscriptores sobre el cambio usando tareasSubject.next(this.tareas), y luego llama a tareasCounter(), esta ultimo método es un ‘BehaviorSubject’ responsable de contar el número de tareas en la lista (tareas) y luego notificar a los suscriptores sobre este recuento a través de tareasCountSubject.next (this.tareas.length).
getTareas(): Observable<any>: El método devuelve un Observable lo que permite a los componentes suscritos escuchar y recibir actualizaciones sobre la lista de tareas (tareas[]). Esto se logra proporcionando acceso a tareasSubject como un Observable. Cuando se suscriben a este Observable, los componentes pueden recibir notificaciones cada vez que hay un cambio en la lista de tareas, lo que les permite mantenerse actualizados con los datos más recientes. Con el mismo

objetivo creamos getCounterPendientes():Observable<any>, con la diferencia que cuando se llame a este método se obtendrá acceso a los valores emitidos por 'tareasCountSubject'

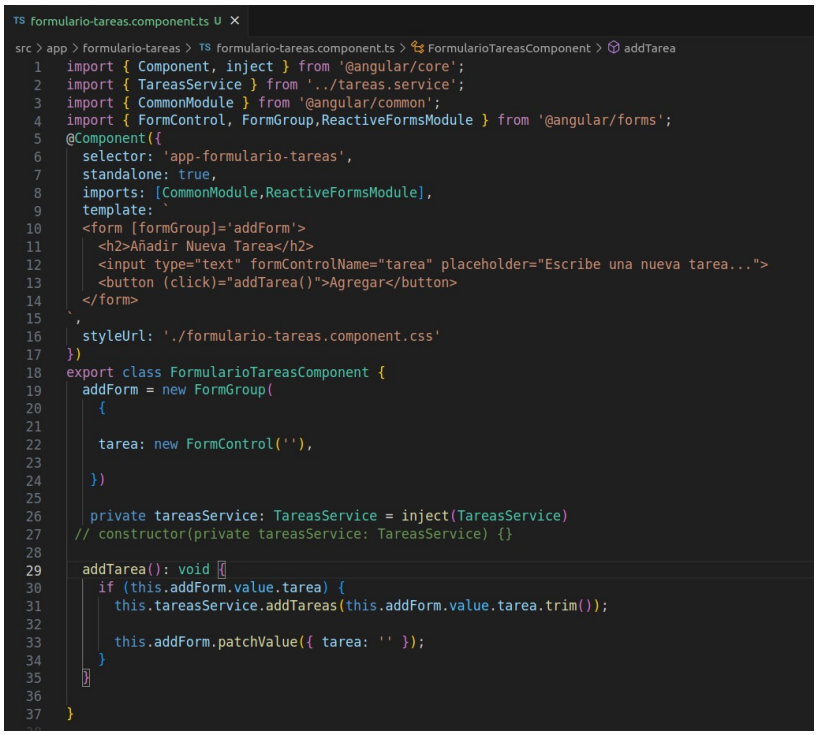
```
19
20      addTareas(tarea: string): void {
21      this.tareas.push(tarea);
22      this.tareasSubject.next(this.tareas);
23      this.tareasCounter()
24
25    }
26    getTareas():Observable<any>{
27      return this.tareasSubject.asObservable()
28       }
29
30    ////////////////////////////////////////////////
31
32      getCounterPendientes():Observable<any>{
33
34       return this.tareasCountSubject.asObservable()
35
36      }
37      tareasCounter(){
38
39       this.tareasCountSubject.next(this.tareas.length);
40      }
41    ////////////////////////////////////////////////////
```

El método getHechasLista() proporciona un medio para obtener como observable hechasListaSubject, para que los componentes suscritos estén al tanto de cualquier cambio en la lista de completedTareas[].
Por otro lado, el método markTareaskAsDone(taskIndex: number) se encarga de marcar una tarea específica como completada, según su índice en la lista de tareas pendientes, este index se recoge en ListaTareasComponent, , se verifica que el índice proporcionado esté dentro de los límites válidos, if (taskIndex >= 0 && taskIndex < this.tareas.length),luego, la tarea se a eliminar se guarda en una variable const tareaEliminada = this.tareas[taskIndex]; para luego eliminar la tarea de tareas[] mediante this.tareas.splice(taskIndex, 1); se añade dicha tarea a la lista de tareas completadas, this.completedTareas.push(tareaEliminada) para después emitir los cambios de las dos array, tareas[] y completedTareas[], a los suscriptores this.hechasListaSubject.next(this.completedTareas); this.tareasSubject.next(this.tareas); por último ejecutar los contadores tareasCounter(), ya explicado en el párrafo anterior y this.counterTareasdone() que ejecuta otro BehaviorSubject , con la diferencia que este lleva la cuenta de las tareas que ya han sido realizadas, this.hechasSubjectCounter.next(this.hechasSubjectCounter.value + 1), y que es pasado como observable a sus subscriptores a través de getCounterHechas():Observable<any> {…...}

```
TS tareas.service.ts U ×
src > app > TS tareas.service.ts > TareasService > getCounterHechas
10  export class TareasService {
42    counterTareasdone(){
43
44        this.hechasSubjectCounter.next(this.hechasSubjectCounter.value +
45    }
46
47    getCounterHechas():Observable<any> {
48
49    return this.hechasSubjectCounter.asObservable()
50
51    }
52    ////////////////////////////////////////////////////////7
53
54    getHechasLista():Observable<any>{
55      return this.hechasListaSubject.asObservable()
56    }
57
58
59    markTareaskAsDone(taskIndex: number): void {
60
61      if (taskIndex >= 0 && taskIndex < this.tareas.length) {
62
63        const tareaEliminada = this.tareas[taskIndex];
64
65        this.tareas.splice(taskIndex, 1);
66        this.completedTareas.push(tareaEliminada)
67        this.hechasListaSubject.next(this.completedTareas)
68        this.tareasSubject.next(this.tareas );
69        this.tareasCounter()
70        this.counterTareasdone()
71
72      }
73    }
```

En conjunto, estos métodos permiten una gestión fluida de las tareas completadas y pendientes dentro de la aplicación, manteniendo a los componentes informados sobre cualquier cambio relevante en estas listas.

3)L**ógica de componentes**
Comenzaremos con el ‘formulario-tareas.components.ts’ para añadir nuevas tareas, en la cabecera debemos importar varios módulos y servicios necesarios para el componente, ‘TareasService’ para acceder al servicio que hemos creado, ‘CommonModule’ para la utilización de las directivas en el html; FormControl, FormGroup y ReactiveFormsModule para crear el formulario.



Clase del Componente ‘formularioTareasComponent {}’:
Establecemos un formulario simple en el que los usuarios pueden escribir nuevas tareas. La conexión entre los elementos del formulario y la lógica del componente se realiza a través de FormGroup y FormControl, lo que permite la manipulación y validación de los datos del formulario en la clase del componente.
addForm: Se define un nuevo FormGroup llamado addForm. Este FormGroup actúa como un contenedor para los controles del formulario. En este caso, solo contiene un FormControl llamado "tarea".
tarea: new FormControl(''): Se define un FormControl llamado "tarea". Este FormControl representa el campo de entrada de texto en el formulario. El valor inicial del campo se establece en una cadena vacía.
addTarea(): Este método se invoca cuando se hace clic en el botón "Agregar" en el formulario. Primero, verifica si el campo de tarea no está vacío (if (this.addForm.value.tarea)). Si no está vacío, se llama al método addTareas(),(ver punto 2) del servicio TareasService para agregar la tarea. El .trim() lo usamos para eliminar espacios vacíos al principio y final de la cadena de texto, luego, se limpia el campo de tarea estableciendo su valor en una cadena vacía usando this.addForm.patchValue({ tarea: '' }).

Plantilla HTML:
El siguiente ‘template’ HTML define la visualización del formulario donde se recojeran los datos para nuestra lista de tareas

<form [formGroup]='addForm'>: sirve como contenedor para el formulario. El atributo formGroup establece la relación entre este ‘div’ y el FormGroup definido en la clase del componente. En este caso, addForm es el FormGroup que contiene los controles del formulario.

<input type="text" formControlName="tarea" placeholder="Escribe una nueva tarea...">: Este es un campo de entrada de texto que permite al usuario escribir una nueva tarea. El atributo formControlName establece la conexión entre este campo de entrada y el FormControl llamado "tarea" dentro del FormGroup addForm.

<button (click)="addTarea()">Agregar</button>: Este es un botón que, al hacer clic en él, invocará el método addTarea() definido en la clase del componente. Su función es agregar la tarea escrita por el usuario al formulario.

El siguiente componete para crear sera el que rederize la lista de tareas, abrimos ‘lista-tareas.components.ts’, importamos en la bacera ‘CommonModule’ y ‘TareasService’.
El componente ListaTareasComponent sera responsable de mostrar una lista de tareas pendientes y una lista de tareas realizadas. ListaTareasComponent muestra de manera dinámica las tareas pendientes y realizadas en la interfaz de usuario. Utiliza el patrón de suscripción a observables para mantener sincronizados los datos mostrados con los datos obtenidos del servicio TareasService. Además, proporciona la capacidad de marcar las tareas pendientes como realizadas mediante la interacción del usuario.

Clase del Componente ListaTareasCompnent{}:
La clase del componente contiene la lógica que controla el comportamiento y la funcionalidad de la vista.
Se declaran dos array, nuevastareas[] y tareasHechas[], para almacenar las tareas pendientes y realizadas respectivamente.
Método ngOnInit(){...}: Este método se ejecuta después de que Angular ha inicializado el componente. Aquí se suscribe a los observables proporcionados por el servicio TareasService para obtener las listas de tareas pendientes y realizadas, this.tareasService.getTareas().subscribe({..}) y this.tareasService.getHechasLista().subscribe({…}), como vemos se hace referencia al servicio, este se encadena con al método del servicio que nos devuelve el ‘subject’ como obserbale y y este último con el método subscribe() que recibirá los datos cada vez que el ‘subject’ del servicio emita un cambio, los datos recibidos se manipulan mediante ‘next: =>’ .Cuando se emiten nuevos valores, los arreglos nuevastareas[] y tareasHechas[] se actualizan con las últimas tareas recibidas.
Método markTareaAsDone(): Este método se invoca cuando se hace clic en el botón "Marcar como realizada" en la lista de tareas pendientes. Recibe el índice de la tarea como argumento y llama a la función this.tareasService.markTareaskAsDone(taskIndex) del servicio TareasService , que ejecuta una serie de operaciones para actualizar los datos, (ver punto 2) . Este indice se recoge en la plantilla del componente <li *ngFor="let tarea of nuevastareas,let i = index"> ; <button (click)= "markTareaAsDone(i)">Marcar como realizada</button>; ‘index’ es una variable global que nos da la posición dentro de un array de un valor en concreto cuando iteramos sobre ella.

```ts
import { Component, inject } from '@angular/core';
import { TareasService } from '../tareas.service';
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-lista-tareas',
  standalone: true,
  imports: [CommonModule],
  template: `
  <div>
    <h2>Lista de Tareas Pendientes</h2>
    <ul>
      <li *ngFor="let tarea of nuevastareas,let i = index">
      {{ tarea }}
      <button (click)="markTareaAsDone(i)">Marcar como realizada</button>

      </li>
    </ul>
  </div>
  <div *ngIf="nuevastareas.length === 0">
    <p>No hay tareas pendientes</p>
  </div>
  <div >
  <h2>Tareas Realizadas</h2>
  <ul>
    <li *ngFor="let task of tareasHechas">
      {{ task }}
    </li>
  </ul>
</div>
  `,
  styleUrl: './lista-tareas.component.css'
})
```

```ts
  styleUrl: './lista-tareas.component.css'
})
export class ListaTareasComponent {
  nuevastareas: string[] = [];
  tareasHechas: string[] = [];
  private tareasService: TareasService = inject(TareasService)
 // constructor(private tareasService: TareasService) {}

  ngOnInit(): void {
    this.tareasService.getTareas()
    .subscribe({
    next: tarea => this.nuevastareas = tarea
    })
    this.tareasService.getHechasLista()
    .subscribe({
      next: tarea => this.tareasHechas = tarea
    })
    }

   markTareaAsDone(taskIndex: number): void {

    this.tareasService.markTareaskAsDone(taskIndex);

  }

}
```

Plantilla HTML:

El siguiente ‘template’ HTML define cómo se visualizarán las tareas en la interfaz de usuario.

<h2>Lista de Tareas Pendientes</h2> Se muestra un título y una lista de tareas pendientes usando *ngFor para iterar sobre el array ‘nuevastareas[]’. Cada tarea se muestra en un <li> y se proporciona un <boton> "Marcar como realizada" que llama a la función ‘markTareaAsDone(i)’ cuando se hace clic en él. Aquí pasamos el index de la tarea que debemos pasar a la función para que el servicio elimine la tarea de la lista de pendientes y al mismo tiempo las añada en tareas realizadas, como explicamos antes,<li *ngFor="let tarea of nuevastareas,let i = index"> ; <button (click)= "markTareaAsDone(i)">Marcar como realizada</button>

No hay tareas pendientes : Si el array ‘nuevastareas[]’ está vacío, <div *ngIf="nuevastareas.length === 0"> se muestra un mensaje indicando que no hay tareas pendientes.

<h2>Tareas Realizadas</h2>: Se muestra un título y una lista de tareas realizadas usando *ngFor para iterar sobre el array ‘tareasHechas[]’,<li *ngFor="let task of tareasHechas">{{ task }}

El último componente a añadir es el que mediante la utilización del patrón de suscripción a observables nos muestre un contador con el número de tareas realizadas y pendientes. Abrimos ‘barra-tareas.componets.ts’

Clase del Componente BarraTareasComponent {}:

La clase ‘BarraTareasComponent’ contiene la lógica que controla el comportamiento y la funcionalidad del componente. En la importación de cabecera importamos ‘TareasService’
Se declaran dos propiedades tareasPendientes !: number y tareasHechas!:number para almacenar el número de tareas pendientes y tareas realizadas respectivamente.

Método ngOnInit(){...}:Este método se ejecuta después de que Angular ha inicializado el componente. Dentro de este método, se suscriben a los observables proporcionados por el servicio ‘TareasService’ para obtener la cantidad de tareas pendientes y tareas realizadas.
Cuando se emiten nuevos valores, las propiedades ‘tareasPendientes’ y ‘tareasHechas’ se actualizan con las últimas cantidades recibidas, esto se consigue utilizando la misma lógica utilizada en ‘ListaTareasCompnents’,es decir se suscribe a los observables proporcionados por el servicio TareasService,pero esta vez llamamos a los metodos getCounterPendientes() y getCounterHechas(), que nos devuelven como observables los ‘BehaviorSubject’ que llevan este conteo , ellos son ‘hechasSubjectCounter’ y ‘hechasListaSubject’, (ver punto 2).

```ts
import { Component } from '@angular/core';
import { TareasService } from '../tareas.service';

@Component({
  selector: 'app-barra-tareas',
  standalone: true,
  imports: [],
  template:
  <div>
    <p>Tareas Pendientes: {{ tareasPendientes }}</p>
  </div>
  <div>
    <p>Tareas hechas: {{ tareasHechas }}</p>
  </div>
,
  styleUrl: './barra-tareas.component.css'
})
export class BarraTareasComponent {
  tareasPendientes !: number
  tareasHechas!:number

  constructor(private tareasService: TareasService) {}

ngOnInit(): void
  {
    this.tareasService.getCounterPendientes()
    .subscribe({
    next: tarea => this.tareasPendientes = tarea,
      }),

      this.tareasService.getCounterHechas()
      .subscribe({next: tarea => this.tareasHechas = tarea})
    }

}
```

Plantilla HTML:

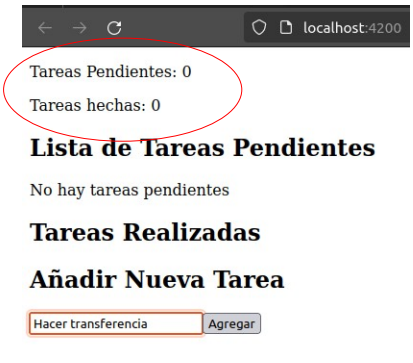
El siguiente ‘template’ HTML define dos secciones donde se mostrarán las cantidades de tareas pendientes y tareas realizadas.

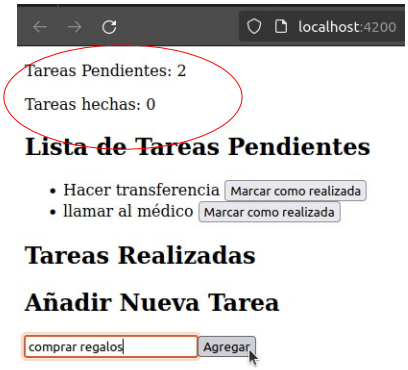
Tareas Pendientes :

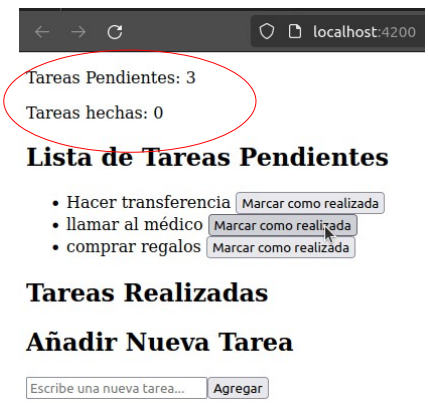
Se muestra un mensaje que indica "Tareas Pendientes:", seguido del número de tareas pendientes. La cantidad de tareas pendientes se representa mediante la interpolación de datos {{ tareasPendientes }}, que se actualiza dinámicamente en función de los datos recibidos del servicio.

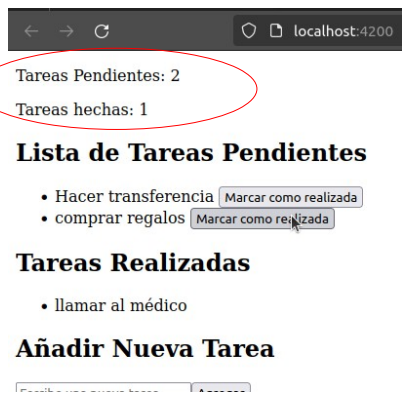
Tareas Realizadas :

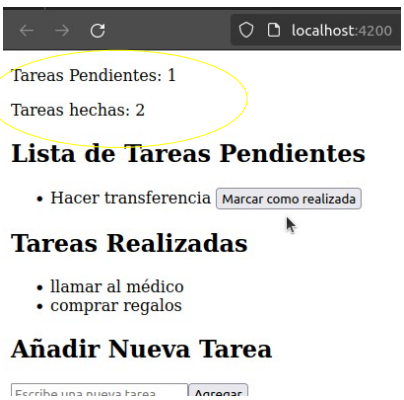
Similar al anterior, se muestra un mensaje que indica "Tareas hechas:", seguido del número de tareas realizadas. La cantidad de tareas realizadas se representa mediante la interpolación de datos {{ tareasHechas }}, que también se actualiza dinámicamente.









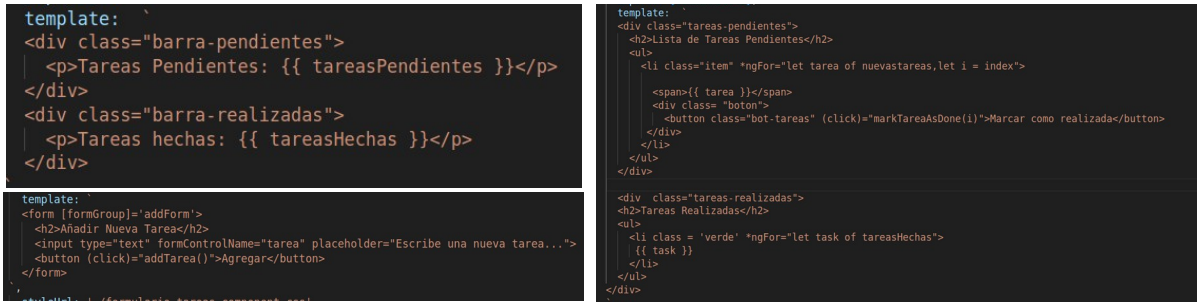


Creamos una clase ‘container’ la cual aplicaremos sobre <app-barra-tareas class="container"></app-barra-tareas> y sobre <app-lista-tareas class="container"></app-lista-tareas> en ‘app.component.html’para activar el flexbox y permitir el diseño en columnas, Con este CSS y HTML, las "Tareas Pendientes" estarán en una columna a la izquierda y las "Tareas Realizadas" estarán en una columna a la derecha, de la misma forma se ubicaran los contadores y el formulario quedará entre contadores y listas.


```
.container {
    display: flex;
    text-align: center;
  }
```

Tendremos un ‘color-schema: ligth dark’ para obtener letras blancas que contrasten con una paleta de colores oscura. El ‘body’ será de color negro y utilizaremos verde oscuro para el contador ‘tareas hechas’ y la lista de ‘tareas realizadas’, y un color rojo oscuro para el contador de ‘tareas pendientes ‘ y la lista ‘tareas pendientes ‘ . Utilizamos ‘flex: 1’ ; tanto para las listas como para los contadores ya que dos elementos dentro de un contenedor flexible, ambos elementos ocuparán la misma cantidad de espacio disponible, dividiéndose equitativamente el espacio disponible. A las listas le hemos añadido también un largo de 750px.

En cuanto al formulario, (’form’) esta alineado en el centro y con los estilos aplicados sobre el ‘input’ y ‘button’ para que solo veamos los bordes en blanco.



Dentro de la ‘lista de tareas pendientes’ establecemos un diseño de cuadrícula para elementos con la clase ‘.item’. Cada elemento .item es un <li> que se divide en dos columnas, donde la primera columna se ajusta automáticamente al contenido y la segunda columna tiene un ancho mínimo.
El contenido de texto dentro de un elemento se coloca en la primera columna utilizando la propiedad grid-column: 1, y se alinea verticalmente en el centro con align-self: center. Se establece un ancho máximo de 300px para el texto y se permite que las palabras se pasen a una nueva linea si

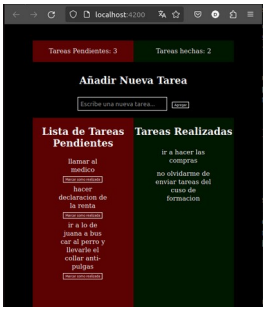
no caben en la primera con word-wrap: break-word. Además, se centra horizontalmente el texto utilizando margin: 0 auto.
El botón para ‘marcar tarea como realizada’, identificado con la clase .boton, se coloca en la segunda columna utilizando grid-column: 2. También se alinea verticalmente en el centro con align-self: center y se agrega un margen derecho de 20px para separarlo del contenido de texto.
Para la ‘lista de tareas realizadas’ marcada con la clase .verde incluye un ancho máximo de 300px para los elementos, permitiendo que el contenido se ajuste a este ancho máximo y pasen a una nueva linea de texto si es necesario (word-wrap: break-word). Además, el contenido se centra horizontalmente dentro de su contenedor utilizando margin: 0 auto;.

Finalmente el código CSS contiene tres consultas de medios para manejar la respondividad (@media):
La primera consulta se aplica cuando el ancho de la ventana del navegador es máximo de 1250px. En este caso, el tamaño de fuente del cuerpo (body) se reduce (font-size: smaller;). Además, los elementos con las clases .tareas-pendientes y .tareas-realizadas tienen un ancho de 200px, y el botón para ‘marcar como tareas realizadas’, button ,tienen un tamaño de fuente de 7px.

```
@media (max-width: 1250px) {
  body {
    font-size: smaller;
  }
  .tareas-pendientes, .tareas-realizadas{ width: 200px;}
  button{
    font-size: 7px;
  }
}
```


```
@media (max-width: 950px) {


  .tareas-pendientes, .tareas-realizadas{ width: 100px;}
 span{max-width: 200px;
}
}
```


```
@media (max-width: 800px) {

 span{max-width: 100px;}
 .item {
   grid-template-columns: 1fr;
}

.boton {
  grid-column: auto;
  margin-right: 0;
}

ul{margin-right: 20%;}

}
```



La segunda consulta se aplica cuando el ancho de la ventana del navegador es máximo de 950px. En este caso, los elementos con las clases .tareas-pendientes y .tareas-realizadas tienen un ancho de 100px. Además, todos los elementos tienen un ancho máximo de 200px.

La tercera consulta se aplica cuando el ancho de la ventana del navegador es máximo de 800px. En este caso, el ancho de los elementos se reduce a 100px. Además, la disposición de los elementos con la clase .item cambia para tener una sola columna (grid-template-columns: 1fr;).Los botones dentro de .boton tienen un ancho de columna automático y el margen derecho se elimina. Además, se ajusta el margen derecho de las listas desordenadas (ul) al 20% del contenedor.

4) **Demostración del funcionamiento**(ver video)