Una vez creado y lanzado Angular.

1º Creamos los componentes, vistas y servicios que necesitaremos.

2º Borramos el contenido de app.html y añadimos lo que queramos que se vea.

app.html

examenFebrero > src > app > app.html > router-outlet

```
<!-- Componente de cabecera de la aplicación. Se muestra de forma fija en todas las vistas -->
<app-header></app-header>

<!-- Contenedor donde Angular carga dinámicamente el componente asociado a la ruta activa. Permite la navegación entre vistas (SPA) -->
<router-outlet />
```

3º Importamos en app.ts el componente que queramos visualizar.

app.ts

examenFebrero > src > app > app.ts > ...

```
import { Component, signal } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Header } from './components/header/header';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, Header],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected readonly title = signal('examenFebrero');
}
```

4º En app.routes.ts importamos las vistas y componentes que va a tener el nav. Añadimos una por defecto.

app.routes.ts

examenFebrero > src > app > app.routes.ts > ...

```
import { Routes } from '@angular/router';
import { Header } from './components/header/header';
import { Ejercicio2 } from './views/ejercicio2/ejercicio2';
import { Ejercicio3 } from './views/ejercicio3/ejercicio3';
import { Ejercicio4 } from './views/ejercicio4/ejercicio4';

export const routes: Routes = [
    { path: '', redirectTo: 'header', pathMatch: 'full' },
    { path: 'header', component: Header },
    { path: 'ejercicio2', component: Ejercicio2 },
    { path: 'ejercicio3', component: Ejercicio3},
    { path: 'ejercicio4', component: Ejercicio4 }
];
```

5º En header.ts importamos RouterLInk y RouterLinkActive para poder crear las rutas.

header.ts

examenFebrero > src > app > components > header > header.ts > Header

```
import { Component } from '@angular/core';
import { RouterLink, RouterLinkActive } from '@angular/router';

@Component({
  selector: 'app-header',
  imports: [RouterLink, RouterLinkActive],
  templateUrl: './header.html',
  styleUrl: './header.css',
})
export class Header {

}
```

6º En el header.html creamos el nav (con la clase indicada en la plantilla) y las rutas (con sus respectivas clases).

header.html

examenFebrero > src > app > components > header > header.html > nav.header

```html
<nav class="header">
    <ul>
        <li><a class="question-link" routerLink="/ejercicio2"
        routerLinkActive="active">ejercicio2</a></li>
        <li><a class="question-link" routerLink="/ejercicio3"
        routerLinkActive="active">ejercicio3</a></li>
        <li><a class="question-link" routerLink="/ejercicio4"
        routerLinkActive="active">ejercicio4</a></li>
    </ul>
</nav>
```

1.- Creación e instanciación de los componentes *Pokemon* y *Photos*

Creación de los componentes.

Importación de componentes en Ejercicio2.

Para poder utilizar los componentes Pokemon y Photos dentro de la vista Ejercicio2, se importan en el decorador @Component del componente padre.

```typescript
import { Component } from '@angular/core';

// Importación de los componentes hijos
import { Pokemon } from '../../components/pokemon/pokemon';
import { Photos } from '../../components/photos/photos';

@Component({
  selector: 'app-ejercicio2',

  // Al ser componentes standalone, se importan directamente aquí
  imports: [Pokemon, Photos],

  templateUrl: './ejercicio2.html',
  styleUrl: './ejercicio2.css',
})
export class Ejercicio2 {

  // Este componente actuará como padre y controlador del flujo
  // La lógica de datos y estado se implementará en apartados posteriores

}
```

Instanciación de los componentes en la vista

Una vez importados, los componentes pueden instanciarse en el HTML de Ejercicio2 mediante sus selectores:

- <app-pokemon> para mostrar cada tarjeta de Pokémon.
- <app-photos> para mostrar la galería de imágenes.

Es importante destacar que **no se copian directamente los bloques .card de la plantilla**, sino que estos se sustituyen por el componente Pokemon.

Respeto de la estructura de la plantilla original

Para que el CSS proporcionado en la plantilla funcione correctamente, se mantiene la estructura HTML original:

- #page
- section
- .grid

Esta estructura actúa como contenedor del layout, mientras que los componentes hijos se encargan únicamente de renderizar el contenido dinámico.

```
<div id="page">
  <section>

    <!-- Contenedor grid proporcionado por la plantilla -->
    <div class="grid">

      <!-- En este punto se instanciarán los componentes Pokemon -->
      <!-- La repetición se realizará posteriormente mediante un bucle -->
      <app-pokemon></app-pokemon>

    </div>

  </section>
</div>
```

2.- Obtención de datos mediante HTTP (PokeAPI)

Objetivo de esta parte

El enunciado exige que, **al cargarse la vista Ejercicio2**, se muestren los datos de los Pokémon **Pikachu, Bulbasaur y Charmander**, obtenidos mediante una **petición GET** a la API REST pública **PokeAPI**.

Para ello, se implementa un **servicio Angular** encargado de realizar las peticiones HTTP.

Este enfoque sigue la buena práctica de Angular de separar la lógica de acceso a datos (servicios) de la lógica de presentación (componentes).

Creación del servicio de acceso a PokeAPI

Se crea un servicio (por ejemplo, PokeApi) cuya responsabilidad es:

- Centralizar las peticiones HTTP.
- Facilitar la reutilización.
- Evitar repetir lógica en componentes.
- Mantener el código más limpio y mantenible.

Configuración del cliente HTTP en Angular

Para poder utilizar HttpClient, el proyecto debe tener habilitado el proveedor HTTP en la configuración global (app.config.ts).

```ts
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient, withFetch } from '@angular/common/http';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    // Manejo global de errores del navegador
    provideBrowserGlobalErrorListeners(),

    // Configuración de rutas
    provideRouter(routes),

    // Habilita HttpClient en toda la aplicación
    // withFetch() permite usar fetch internamente (Angular moderno)
    provideHttpClient(withFetch())
  ]
};
```

Implementación del método GET en el servicio

La PokeAPI permite obtener un Pokémon individual usando el endpoint:

- https://pokeapi.co/api/v2/pokemon/{nombre}

Por tanto, el servicio expone un método que recibe el nombre del Pokémon (string) y devuelve un Observable con la respuesta. En poke-api.ts:

```ts
import { inject, Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

// En esta parte se puede utilizar any temporalmente,
// aunque posteriormente se sustituirá por la interfaz (Parte 3)
import { PokemonResponse } from '../interfaces/pokemon-response';

@Injectable({
  providedIn: 'root',
})
export class PokeApi {

  // Inyección de HttpClient mediante inject() (Angular moderno)
  private http = inject(HttpClient);

  /**
   * Realiza una petición GET a la PokeAPI para obtener un Pokémon concreto.
   * @param nombre Nombre del Pokémon (ej: 'pikachu')
   */
  public getPokemon(nombre: string): Observable<PokemonResponse> {
    return this.http.get<PokemonResponse>(
      `https://pokeapi.co/api/v2/pokemon/${nombre}`
    );
  }
}
```

Carga de datos al iniciar la vista

Para que los datos se obtengan **automáticamente al cargarse Ejercicio2**, se ejecuta la lógica de carga en el ciclo de vida del componente (por ejemplo, en ngOnInit).

Se define:

- Un array de nombres con los Pokémon requeridos: ['pikachu', 'bulbasaur', 'charmander']
- Un método loadPokemon() que realiza las peticiones a la API
- Una estructura donde almacenar los datos ya procesados para la vista

En ejercicio2.ts:

```ts
import { Component, inject, OnInit } from '@angular/core';
import { PokeApi } from '../../services/poke-api';
import { PokemonResponse } from '../../interfaces/pokemon-response';

type PokemonVM = {
  name: string;
  photo: string;
  sprites4: string[];
};

@Component({
  selector: 'app-ejercicio2',
  imports: [],
  templateUrl: './ejercicio2.html',
  styleUrl: './ejercicio2.css',
})
export class Ejercicio2 implements OnInit {

  // Inyectamos el servicio que gestiona las peticiones HTTP
  private pokemonService = inject(PokeApi);

  // Lista de Pokémon que exige el enunciado
  public names: string[] = ['pikachu', 'bulbasaur', 'charmander'];

  // Array donde se almacenarán los Pokémon ya listos para mostrar en la vista
  public pokemon: PokemonVM[] = [
    { name: '', photo: '', sprites4: [] },
    { name: '', photo: '', sprites4: [] },
    { name: '', photo: '', sprites4: [] }
  ];

  /**
   * Método que se ejecuta al cargar el componente.
   * Se utiliza para iniciar la carga HTTP.
   */
  ngOnInit(): void {
    this.loadPokemon();
  }
```

```
  /**
   * Carga los datos de cada Pokémon realizando una petición GET por nombre.
   * Cada respuesta se transforma al formato que necesita la vista (PokemonVM).
   */
  public loadPokemon(): void {

    this.names.forEach((nombre, i) => {

      this.pokemonService.getPokemon(nombre).subscribe((response: PokemonResponse) => {

        // Transformación de la respuesta de la API al modelo de vista
        this.pokemon[i] = {
          name: response.name,
          photo: response.sprites.front_default,
          sprites4: [
            response.sprites.front_default,
            response.sprites.back_default,
            response.sprites.front_shiny,
            response.sprites.back_shiny,
          ].filter(Boolean) as string[]
        };

      });
    });
  }
}
```

Resultado de esta fase

Con esta implementación:

- Al entrar en Ejercicio2, se lanza automáticamente la carga de datos.
- Se realizan 3 peticiones GET a PokeAPI, una por cada Pokémon requerido.
- Los datos obtenidos se almacenan en un array listo para ser usado por los componentes (Pokemon y Photos).

3.- Modelado de datos con interfaces (TypeScript)

Objetivo de esta parte

El enunciado exige “implementar las interfaces necesarias para gestionar la respuesta a la petición GET”. Para cumplir este punto, se define un conjunto de **interfaces TypeScript** que representan la estructura de la respuesta devuelta por PokeAPI.

Identificación de los datos necesarios

Aunque la respuesta de PokeAPI contiene muchas propiedades, para este ejercicio solo se necesitan:

- name: nombre del Pokémon
- sprites: conjunto de URLs de imágenes (sprites)

Dentro de sprites, para cumplir los requisitos del ejercicio se utilizan:

- front_default (imagen principal del Pokémon en la card)
- back_default, front_shiny, back_shiny (para las 4 miniaturas)

Por tanto, no es necesario modelar toda la respuesta completa, sino únicamente la parte relevante para el enunciado.

Creación del archivo de interfaces

Se crea un archivo de interfaces (pokemon-response.ts) dentro de una carpeta interfaces/ para centralizar el tipado de la API.

Esta organización permite que el servicio y los componentes importen el mismo modelo sin duplicación.

```ts
/**
 * Representa la estructura mínima de respuesta de PokeAPI
 * necesaria para este ejercicio.
 *
 * Solo se incluyen los campos que vamos a usar en la aplicación.
 */
export interface PokemonResponse {

  /**
   * Nombre del Pokémon (ej: 'pikachu')
   */
  name: string;

  /**
   * Contiene las URLs de imágenes del Pokémon
   */
  sprites: Sprites;
}

/**
 * Representa el bloque 'sprites' de la respuesta de PokeAPI.
 * Aquí se incluyen las propiedades necesarias para:
 * - mostrar una imagen en el componente Pokemon
 * - mostrar 4 miniaturas en el componente Photos
 */
export interface Sprites {

  /** Imagen frontal por defecto */
  front_default: string;

  /** Imagen trasera por defecto */
  back_default: string;

  /** Imagen frontal shiny (variación brillante) */
  front_shiny: string;

  /** Imagen trasera shiny */
  back_shiny: string;
}
```

Uso de la interfaz en el servicio y componentes

Una vez definidas las interfaces:

- El servicio PokeApi tipa la respuesta del GET con PokemonResponse.
- El componente padre Ejercicio2 tipa la variable response al hacer subscribe.
- Se mejora la legibilidad y se eliminan errores de acceso a propiedades inexistentes.

```ts
public getPokemon(nombre: string): Observable<PokemonResponse> {
  return this.http.get<PokemonResponse>(`https://pokeapi.co/api/v2/pokemon/${nombre}`);
}
```

4.- Comunicación Padre → Hijo (Ejercicio2 → Pokemon) con @Input

Objetivo de esta parte

Una vez que la vista Ejercicio2 ya obtiene los datos desde la API (servicio HTTP) y los prepara en un array con la información necesaria, el siguiente paso es **enviar esos datos al componente hijo** Pokemon para que los muestre.

En Angular, la comunicación **del padre al hijo** se realiza mediante **@Input**, de forma que el padre **"inyecta" datos** al hijo a través de propiedades públicas. Esto encaja con la filosofía de componentes: el padre controla los datos y el hijo se centra en renderizarlos.

Qué datos se pasan al componente Pokemon

Según el enunciado, en la vista de lista (estado inicial) cada tarjeta debe mostrar:

- **Nombre del Pokémon**
- **Una imagen (sprite) del Pokémon**

Por tanto, el componente Pokemon debe recibir como @Input:

- name: string
- photo: string

En esta parte **no se incluye todavía @Output**, porque eso pertenece a la comunicación Hijo → Padre

Definición de @Input en el componente hijo (pokemon.ts)

En el componente Pokemon se declaran dos propiedades públicas decoradas con @Input.

Esto permite que el padre las pueda establecer desde el HTML usando **property binding** ([name]="...").

```ts
import { Component, Input } from '@angular/core';
import { NgStyle } from '@angular/common';

@Component({
  selector: 'app-pokemon',
  // NgStyle se usa para aplicar estilos dinámicos desde el template (foto)
  imports: [NgStyle],
  templateUrl: './pokemon.html',
  styleUrl: './pokemon.css',
})
export class Pokemon {

  // Recibe desde el padre el nombre del Pokémon (ej: 'pikachu')
  @Input() name: string = '';

  // Recibe desde el padre la URL de la imagen principal del Pokémon (sprite)
  @Input() photo: string = '';
}
```

Renderizado de los @Input en el template del hijo (pokemon.html)

En el HTML del componente Pokemon se utiliza:

- {{ name }} para mostrar el nombre.
- [ngStyle] para aplicar dinámicamente la imagen como fondo del div .photo.

El **layout** (posicionamiento y distribución en columnas) lo proporciona la estructura del padre (#page y .grid) en Ejercicio2, tal como viene en la plantilla.

```html
<div class="card">

  <!-- Se asigna la imagen del Pokémon de forma dinámica usando ngStyle -->
  <div class="photo"
       [ngStyle]="{ 'background-image': 'url(' + photo + ')' }">
  </div>

  <div class="footer">
    <div class="titles">

      <!-- Se muestra el nombre del Pokémon recibido desde el padre -->
      <h2 class="card-title">
        {{ name }}
      </h2>

    </div>
  </div>

</div>
```

Envío de datos desde el padre usando @for y property binding (ejercicio2.html)

En el componente padre (Ejercicio2) se dispone de un array pokemon con objetos que contienen name y photo. Para pintar **una tarjeta por cada Pokémon**, se utiliza Control Flow @for en el HTML del padre, que permite iterar un array y renderizar un bloque HTML por cada elemento.

Dentro del bucle, se instancia el componente hijo:

- [name]="p.name" → se envía el nombre del objeto actual.
- [photo]="p.photo" → se envía la URL de la imagen.

```html
<div id="page">
  <section>

    <div class="grid">
      <!-- Se recorre el array de pokémon -->
      @for (p of pokemon; track p.name) {

        <!-- Se instancia el hijo y se le pasan datos mediante Inputs -->
        <app-pokemon
          [name]="p.name"
          [photo]="p.photo">
        </app-pokemon>

      }
    </div>

  </section>
</div>
```

5.- Comunicación Hijo → Padre (Ejercicio2 → Pokemon) con @Output + EventEmitter y control del estado de la vista

Objetivo de esta parte

El enunciado indica que, **al pulsar sobre el nombre del Pokémon en el componente Pokemon**, se debe:

- ocultar la vista de lista (componentes Pokemon)

- mostrar el componente Photos con las imágenes del Pokémon seleccionado

Para conseguir esto, el componente hijo (Pokemon) debe **notificar al componente padre (Ejercicio2)** qué Pokémon ha sido seleccionado. En Angular, la comunicación **Hijo → Padre** se realiza mediante:

- @Output
- EventEmitter

El padre recibe el evento y actualiza su estado (por ejemplo mediante una variable mode) para cambiar la vista.

Emisión del evento desde el componente Pokemon

El componente Pokemon emite un evento cuando el usuario hace click en el nombre. En la implementación realizada, el evento emite un dato simple: la **URL de la imagen principal (photo)**, que el padre utiliza como referencia para localizar el Pokémon seleccionado dentro del array.

Código comentado (pokemon.ts)

```
import { Component, EventEmitter, Input, Output } from '@angular/core';
import { NgStyle } from '@angular/common';

@Component({
  selector: 'app-pokemon',
  imports: [NgStyle],
  templateUrl: './pokemon.html',
  styleUrl: './pokemon.css',
})
export class Pokemon {

  // Datos recibidos desde el padre
  @Input() name: string = '';
  @Input() photo: string = '';

  // Evento que se emite hacia el padre cuando se selecciona el Pokémon
  @Output() pokemonSeleccionado = new EventEmitter<string>();

  /**
   * Método que se ejecuta al hacer click en el nombre del Pokémon.
   * Emite hacia el padre un dato identificador del Pokémon seleccionado.
   */
  public seleccionar(): void {
    this.pokemonSeleccionado.emit(this.photo);
  }
}
```

Captura del evento en el padre (Ejercicio2)

En el componente padre (Ejercicio2), se escucha el evento del hijo en el HTML usando:

- (pokemonSeleccionado)="seleccionar($event)"

Donde $event contiene el dato emitido por el hijo (en este caso, la photo).

Código comentado (ejercicio2.html — vista lista)

```
@for (p of pokemon; track p.name) {

  <app-pokemon
    [name]="p.name"
    [photo]="p.photo"

    <!-- Se escucha el evento emitido por el hijo -->
    (pokemonSeleccionado)="seleccionar($event)">
  </app-pokemon>

}
```

Control del estado de la vista (mostrar lista o photos)

Para alternar entre “lista” y “galería”, el padre mantiene una variable de estado:

- mode: 'list' | 'photos'

Este estado se utiliza en el template para mostrar un bloque u otro mediante @if / @else.

Código comentado (ejercicio2.ts — estado de vista)

```
// Estado de la vista:
// - 'list' muestra los 3 Pokémon
// - 'photos' muestra el componente Photos
public mode: 'list' | 'photos' = 'list';
```

Código comentado (ejercicio2.html — control de renderizado)

```
@if (mode === 'list') {

  <!-- lista de Pokémon -->

} @else {

  <!-- componente Photos -->

}
```

Resolución del Pokémon seleccionado y cambio de vista

Como el hijo emite una photo (string), el padre la utiliza para localizar el objeto correspondiente dentro del array pokemon.
Una vez encontrado:

- se guarda en pokemonSeleccionado
- se cambia el estado mode a 'photos'

Código comentado (ejercicio2.ts — método seleccionar)

```ts
public pokemonSeleccionado?: PokemonVM;

/**
 * Recibe la photo emitida por el componente Pokemon.
 * Busca en el array el Pokémon correspondiente y cambia la vista a Photos.
 */
public seleccionar(photo: string): void {

  // Se localiza el Pokémon seleccionado a partir de la photo recibida
  const encontrado = this.pokemon.find(p => p.photo === photo);

  // Si no se encuentra, no se cambia de vista
  if (!encontrado) return;

  // Se guarda el objeto completo (necesario para sprites4)
  this.pokemonSeleccionado = encontrado;

  // Se cambia al modo fotos para mostrar el componente Photos
  this.mode = 'photos';
}
```

Activación del evento en el template del hijo (pokemon.html)

Para disparar el evento cuando el usuario pulsa el nombre del Pokémon, se añade el evento (click) en el h2.

Código comentado (pokemon.html)

```html
<div class="card">
  <div class="photo"
       [ngStyle]="{ 'background-image': 'url(' + photo + ')' }">
  </div>

  <div class="footer">
    <div class="titles">

      <!-- Click en el nombre: se emite el evento al padre -->
      <h2 class="card-title" (click)="seleccionar()">
        {{ name }}
      </h2>

    </div>
  </div>
</div>
```

6.- Componente Photos e interacción del usuario (miniaturas, ampliación y volver)

Objetivo de esta parte

El enunciado exige que, al seleccionar un Pokémon:

1. Se oculte el listado de componentes Pokemon.
2. Se muestre el componente Photos.

3. En Photos se muestren inicialmente **cuatro imágenes en miniatura** (sprites) del Pokémon seleccionado.
4. Al hacer click en una miniatura:
    - se oculten las cuatro miniaturas
    - y se muestre **solo** la imagen seleccionada en grande
5. Si se vuelve a pulsar sobre la imagen grande:
    - se vuelvan a mostrar las cuatro miniaturas
6. Si se pulsa el botón **"volver"**:
    - se regrese al estado inicial (lista de 3 Pokémon)

Para cumplirlo, Photos debe:

- recibir datos desde el padre (@Input)
- emitir el evento de vuelta (@Output)
- gestionar internamente qué imagen está seleccionada

Paso de datos desde el padre al componente Photos (@Input)

El padre (Ejercicio2) ya dispone de un objeto pokemonSeleccionado que contiene sprites4 (array de 4 URLs). Por tanto, al instanciar Photos, se le pasa ese array por @Input.

Además, para evitar errores si pokemonSeleccionado aún no existe, se utiliza un valor por defecto [].

Código comentado (ejercicio2.html)

```
<app-photos
  [sprites4]="pokemonSeleccionado?.sprites4 ?? []"
  (volverEstado)="volverAListe()">
</app-photos>
```

Definición de @Input y @Output en Photos

En el componente Photos se declaran:

- @Input() sprites4: string[] → recibe las 4 imágenes
- @Output() volverEstado = new EventEmitter<void>() → notifica al padre que se quiere volver

Además, se crea un estado interno selectPhoto para saber si hay una imagen seleccionada.

- selectPhoto = null → se muestran miniaturas
- selectPhoto = url → se muestra la imagen grande

Código comentado (photos.ts)

```ts
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-photos',
  imports: [],
  templateUrl: './photos.html',
  styleUrl: './photos.css',
})
export class Photos {

  // Recibe las 4 URLs de sprites desde el componente padre
  @Input() sprites4: string[] = [];

  // Evento para notificar al padre que el usuario quiere volver
  @Output() volverEstado = new EventEmitter<void>();

  // Estado interno: guarda la URL seleccionada o null si se muestran miniaturas
  public selectPhoto: string | null = null;

  /**
   * Notifica al componente padre que se ha pulsado el botón "volver".
   */
  public notificarRegreso(): void {
    this.volverEstado.emit();
  }
}
```

Template de Photos: miniaturas vs imagen ampliada

El template implementa la lógica visual usando Control Flow:

- @if (selectPhoto) → se muestra la imagen seleccionada a tamaño grande
- @else → se muestran las 4 imágenes miniatura recorriendo sprites4 con @for

Además:

- click en miniatura → selectPhoto = i
- click en imagen grande → selectPhoto = null (volver a miniaturas)

Código comentado (photos.html)

```
<!-- PHOTO -->
<div class="container-view-three">
  <div class="photos-view-three">
    <div>

      <!-- Botón para volver al estado inicial -->
      <div id="render-more">
        <button (click)="notificarRegreso()">Atrás</button>
      </div>

      <!-- Si hay una imagen seleccionada, se muestra en grande -->
      @if (selectPhoto) {

        <img [src]="selectPhoto" style="width: 100%;" (click)="selectPhoto = null">

      } @else {

        <!-- Si no hay selección, se muestran las 4 miniaturas -->
        @for (i of sprites4; track i) {

          <img [src]="i" style="width: 24%" (click)="selectPhoto = i">

        }
      }

    </div>
  </div>
</div>
```

Volver al estado inicial (padre)

Cuando Photos emite el evento volverEstado, el componente padre ejecuta un método (volverAListа) que:

- cambia el modo a 'list'
- limpia el Pokémon seleccionado (opcional pero recomendable)

Código comentado (ejercicio2.ts)

```
/**
 * Vuelve al estado inicial: lista de Pokémon.
 */
public volverAListа(): void {
  this.mode = 'list';
  this.pokemonSeleccionado = undefined;
}
```