Laboratorio Angular 18 v2

- **Dependency Injection (DI) en Angular** — Angular usa un contenedor de dependencias jerárquico que resuelve y comparte instancias (servicios) entre componentes y proveedores. `providedIn: 'root'` crea un singleton global; también puedes proveer en `bootstrapApplication` o por ruta para ámbitos más limitados.

- **HttpClient / provideHttpClient()** — En las versiones modernas (Angular 18) se favorece `provideHttpClient()` (para apps standalone) en lugar de importar `HttpClientModule` en módulos; esto registra el servicio `HttpClient` para inyección y permite adaptar comportamientos (p. ej. `withFetch()` o interceptors).

- **rxResource / resource API** — `rxResource` es la variante orientada a RxJS del API de Resources de Angular: declara recursos reactivamente, maneja estados (loading/error/value), cancelación automática y refresco; es ideal cuando el loader devuelve Observables (p. ej. `HttpClient`). Estas APIs facilitan el control de race conditions y cancelaciones.

- **Lazy loading con standalone components** — Con componentes standalone se usa `loadComponent` o `loadChildren` (para módulos) en rutas; la carga puede crear un injector propio y proveer servicios a nivel de ruta. Esto reduce el bundle inicial y permite inyectar providers por ruta.

2) Laboratorio — Objetivos y estructura

Objetivos:

1. Mostrar varios modos de inyección (root, route-scoped, local providers).
2. Hacer peticiones HTTP con `HttpClient` y con `rxResource` (cancelación, refresh).
3. Usar `rxResource` para conectar `HttpClient` → signals.
4. Configurar routing con **lazy loading** de componentes standalone y providers por ruta.

Estructura final (files clave):

```
angular-lab/
  src/
    main.ts
    app/
      app.component.ts
      app.component.html
      app.routes.ts
      services/
        api.service.ts        <-- HttpClient tradicional + wrappers
        resource.service.ts   <-- rxResource usage
      features/
        users/
          users.component.ts   <-- lazy loaded component (standalone)
        dashboard/
          dashboard.component.ts
```

3) Preparar el proyecto (comandos)

```
ng new angular-lab --standalone --routing --style=scss
cd angular-lab
# Asegúrate de usar Angular 18 (si necesitas forzar)
# npm install @angular/core@^18 @angular/cli@^18
```

4) main.ts — bootstrap y registrar HttpClient

Usaremos `provideHttpClient()` (recomendado para apps standalone). También puedes añadir `withFetch()` si quieres que HttpClient use Fetch API internamente.

```
// src/main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient, withFetch } from '@angular/common/http';
import { provideRouter } from '@angular/router';
import { AppComponent } from './app/app.component';
import { routes } from './app/app.routes';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient(/* withFetch() optional */)
  ]
}).catch(err => console.error(err));
```

Nota: `provideHttpClient()` es la forma moderna para que `HttpClient` esté disponible a través de DI en apps standalone.

5) Servicios — HttpClient tradicional + rxResource

5.1 `api.service.ts` — servicio con HttpClient (sencillo, testable)

```
// src/app/services/api.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { firstValueFrom, map } from 'rxjs';

export interface User { id:number; name:string; email:string; }

@Injectable({ providedIn: 'root' })
export class ApiService {
  private base = 'https://jsonplaceholder.typicode.com';

  constructor(private http: HttpClient) {}

  // Observable-based
  getUsers$() {
    return this.http.get<User[]>(`${this.base}/users`);
  }

  // Promise-based helper (cuando lo necesites)
  async getUsersOnce(): Promise<User[]> {
    return firstValueFrom(this.getUsers$());
  }

  getUser$(id: number) {
    return this.http.get<User>(`${this.base}/users/${id}`);
  }
}
```

Puntos técnicos: preferimos exponer Observables para permitir composición RxJS, pero `firstValueFrom` sirve para interop con APIs async/await.

5.2 `resource.service.ts` — usar `rxResource` para exponer recursos reactivos

```
// src/app/services/resource.service.ts
import { Injectable, rxResource } from '@angular/core';
import { ApiService } from './api.service';
import { switchMap, tap } from 'rxjs/operators';
import { signal } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class ResourceService {
  // controlamos el id activo como signal, para que la resource sea reactiva a cambios
  currentUserId = signal<number | null>(null);

  // rxResource: cuando currentUserId cambia, el loader (stream) va a emitir el Observable
de HttpClient
  user = rxResource({
    stream: () =>
      this.currentUserId() == null
        ? of(undefined) // or empty observable
        : this.api.getUser$(this.currentUserId()).pipe(
            // puedes mapear, cachear, etc.
            tap(() => console.log('http user fetched via rxResource'))
          )
  });

  constructor(private api: ApiService) {}

  // util para cambiar id (trigger reload automático)
  setUserId(id: number | null) {
    this.currentUserId.set(id);
  }

  // refresh manual
  refresh() {
    this.user.set(); // según API, set() reinicia/cancela y re-ejecuta
  }
}
```

Notas técnicas sobre `rxResource`:

- `rxResource` convierte un Observable (loader) en un `ResourceRef` que expone estado (loading/error/value) y maneja cancelación automática cuando la dependencia cambia. Ideal para integrar `HttpClient` (observable) con el mundo de signals.

6) Componentes y uso de servicios

6.1 `app.component.ts` — root y nav

```
// src/app/app.component.ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html'
})
export class AppComponent {}
```

`app.component.html` (navegación simple):

```
<nav>
  <a routerLink="/dashboard">Dashboard</a> |
  <a routerLink="/users">Users</a>
</nav>
<main>
  <router-outlet></router-outlet>
</main>
```

6.2 `dashboard.component.ts` — componente no-lazy (ejemplo simple)

```
// src/app/features/dashboard/dashboard.component.ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-dashboard',
  standalone: true,
  imports: [CommonModule],
  template: `
    <h2>Dashboard</h2>
    <p>Demo: metrics + local provider</p>
  `
})
export class DashboardComponent {}
```

6.3 `users.component.ts` — componente lazy-loaded que usa ApiService y ResourceService

```
// src/app/features/users/users.component.ts
import { Component, OnDestroy } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ApiService, User } from '../../services/api.service';
import { ResourceService } from '../../services/resource.service';
import { effect } from '@angular/core';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-users',
  standalone: true,
  imports: [CommonModule],
  template: `
    <h2>Users (lazy loaded)</h2>

    <button (click)="reloadUsers()">Reload (ApiService)</button>

    <section>
      <h3>Observable based (ApiService)</h3>
      <ul>
        <li *ngFor="let u of usersObsSnapshot">{{u.name}} ({{u.email}})</li>
      </ul>
    </section>

    <section>
      <h3>Resource-based user detail (rxResource)</h3>
      <div *ngIf="resourceService.user().state.loading">Loading user…</div>
      <div *ngIf="resourceService.user().value">
        <p>Name: {{ resourceService.user().value.name }}</p>
        <p>Email: {{ resourceService.user().value.email }}</p>
      </div>
      <button (click)="loadUser(1)">Load user #1</button>
      <button (click)="loadUser(2)">Load user #2</button>
      <button (click)="resourceService.refresh()">Refresh</button>
    </section>
  `
})
export class UsersComponent implements OnDestroy {
  usersObsSnapshot: User[] = [];
  private sub?: Subscription;

  constructor(
    private api: ApiService,
    public resourceService: ResourceService
```

```
  ) {
    // ejemplo simple: suscribir para snapshot (demo)
    this.sub = this.api.getUsers$().subscribe(list => (this.usersObsSnapshot = list));
    // ejemplo: effect para reacción cuando resource cambia (opcional)
    effect(() => {
      const v = resourceService.user().value;
      if (v) console.log('resource user changed', v.id);
    });
  }

  reloadUsers() {
    // re-solicitar desde api.service
    this.api.getUsers$().subscribe(list => (this.usersObsSnapshot = list));
  }

  loadUser(id: number) {
    this.resourceService.setUserId(id); // rxResource detecta el cambio y lanza la petición
  }

  ngOnDestroy() { this.sub?.unsubscribe(); }
}
```

Técnicas mostradas:

- Combinación de Observable-based lists y Resource-based single item.
- `resourceService.setUserId(id)` cambia la dependencia (signal) y hace que `rxResource` emita la nueva petición; si el id cambia antes de terminar, la petición anterior se cancela automáticamente (comportamiento de resource/rxResource).

7) Routing + lazy-loading (standalone components)

Creamos `app.routes.ts` con lazy load de `users`:

```
// src/app/app.routes.ts
import { Routes } from '@angular/router';

export const routes: Routes = [
  { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
  {
    path: 'dashboard',
    loadComponent: () => import('./features/dashboard/dashboard.component').then(m =>
m.DashboardComponent)
  },
  {
    path: 'users',
    // Lazy-load standalone component
    loadComponent: () => import('./features/users/users.component').then(m =>
m.UsersComponent),
    // puedes añadir providers específicos a la ruta (scope)
    // providers: [ { provide: SomeService, useClass: RouteScopedService } ]
  }
];
```

Puntos técnicos:

- `loadComponent` descarga el chunk solo cuando el usuario navega a `/users`, reduciendo el bundle inicial.

- Puedes declarar `providers` en la ruta para crear servicios con ámbito de la ruta (route-scoped singletons).

8) Cómo medir y pruebas a ejecutar (benchmark + validación)

Construye en prod:

ng build --configuration production

1. Compara `dist/` sizes y chunks para ver lazy chunks generados.

2. **Servir y abrir /users** ⇒ comprobar en Network los chunks que se bajan al navegar (ver chunk users.*.js).

3. **Medir latencias y cancelaciones**:

    - En `users.component` pulsa Load user 1 y rápidamente Load user 2; observa en Network que la petición 1 es abortada (o verás que rxResource cancela el observable según implementación). Comprueba console logs de `tap()` en el `resource` stream.

4. **Lighthouse** o Web Vitals para comparar inicial load vs post-lazy-navigation.

9) Buenas prácticas

- **Preferir `provideHttpClient()`** en apps standalone; `HttpClientModule` está deprecado/alternativa.

- **Usa `rxResource`** cuando trabajas con Observables (HttpClient): simplifica cancelación y sincronización con signals; para fetch eager / signal-driven prefieres `httpResource` o `resource` según el caso.

- **Inyectar vs proveer por ruta**: usar providers por ruta cuando el servicio debe vivir mientras la ruta existe (scope asociado) — útil para caches o stores por feature.

- **Evitar memory leaks**: `rxResource` y HttpClient observables están bien para evitar unsubscribe manual en muchos casos, pero cuando subscribes manualmente recuerda limpiar.

- **Testing**: mocks de `HttpClient` con `HttpTestingController` y pruebas de integración que verifiquen la cancelación de peticiones cuando la dependencia sinal cambia.

10) Extras (snippets útiles)

a) Proveer HttpClient con `withFetch()`

```
import { provideHttpClient, withFetch } from '@angular/common/http';
provideHttpClient(withFetch());
```

Esto deja a HttpClient usar `fetch` internamente para mejor interoperabilidad en entornos modernos.

b) Route-scoped provider example

```
{
  path: 'users',
  loadComponent: () => import('./features/users/users.component').then(m =>
m.UsersComponent),
  providers: [
    { provide: 'USERS_STORE', useValue: {} }
  ]
}
```

Glosario Angular 18 — Laboratorio 2

Dependency Injection (DI)

Mecanismo central de Angular para gestionar dependencias entre clases.
Permite que los servicios, componentes y pipes declaren en sus constructores qué objetos necesitan, y Angular se encarga de instanciarlos y mantenerlos según un *scope* (global, de módulo, de componente o de ruta).

- `@Injectable()`: marca una clase como inyectable.
- `providedIn: 'root'`: crea una única instancia global (singleton).
- **Route-scoped providers**: definidos en las rutas (`providers: [...]` dentro de `loadComponent`), viven mientras la ruta está activa.
- **Local providers**: declarados en el `@Component`, solo accesibles en esa jerarquía.

En Angular 18, el DI es compatible con los nuevos standalone components y los route injectors, sin necesidad de NgModules.

HttpClient / provideHttpClient()

API de Angular para realizar peticiones HTTP basadas en Observables (`RxJS`).
En Angular 18, la configuración recomendada es a través de:

provideHttpClient(withFetch())

Esto reemplaza al obsoleto `HttpClientModule` y habilita la opción de usar la **Fetch API** nativa del navegador en lugar de XHR.

Ventajas técnicas:

- Mejor integración con entornos SSR y zoneless.
- Cancelación nativa con `AbortController`.
- Menor overhead de bundles.

withFetch()

Función opcional de configuración para `provideHttpClient()` que permite a `HttpClient` internamente usar `fetch()` en lugar de `XMLHttpRequest`.
Aporta compatibilidad con:

- Streaming de respuestas.
- AbortController y cancelación nativa.
- Integración directa con `resource` y `rxResource`.

Observable

Entidad central en RxJS y Angular para manejar flujos asíncronos.
Un `Observable` emite valores en el tiempo (p. ej. respuesta HTTP, cambios de input).
Se gestiona con operadores (`map`, `switchMap`, `tap`, etc.) y se suscribe para obtener los valores.
Angular lo usa internamente para `HttpClient`, `Forms`, `Events`, `Router`, etc.

Signals

Introducidos en Angular 16 y consolidados en Angular 18.
Son **variables reactivas nativas** del framework que notifican automáticamente a la vista cuando cambian.

Ejemplo:

count = signal(0);
increment() { this.count.update(c => c + 1); }

- Sustituyen la necesidad de `ChangeDetectionStrategy.OnPush`.
- Son *zone-less ready*.
- Se integran directamente con `computed` y `effect`.

computed()

Función derivada de signals.
Crea un valor calculado en base a otros signals.
Se recalcula automáticamente cuando cualquiera de sus dependencias cambia.

Ejemplo:

```
total = computed(() => this.price() * this.quantity());
```

effect()

Función reactiva que ejecuta una acción cuando cambia el valor de uno o más signals.
Ideal para side effects (por ejemplo, logs, peticiones, sincronización externa).

Ejemplo:

```
effect(() => console.log('Nuevo total:', this.total()));
```

rxResource

API introducida en Angular 18 que combina **Signals** con **RxJS Observables**.
Permite declarar un recurso que se carga reactivamente (por ejemplo, datos desde `HttpClient`) y gestiona automáticamente:

- Estado (`loading`, `error`, `value`).
- Cancelación de peticiones anteriores al cambiar dependencias.
- Refresh manual.

Ejemplo:

```
userId = signal(1);

user = rxResource({
  stream: () => this.api.getUser$(this.userId())
});
```

Cada vez que `userId` cambia, el recurso recarga y cancela el anterior.

Es la versión basada en Observables del API `resource()`, que usa `async/await`.

resource()

API “hermana” de `rxResource`, orientada a `Promises` o funciones asíncronas.
Ideal para integrarse con `fetch()` o funciones `async`.

```
user = resource({
  loader: async () => await fetch('/api/user').then(r => r.json())
});
```

Lazy Loading

Técnica de división del código (*code splitting*) en Angular.
Permite que solo se descarguen los componentes o rutas cuando son necesarios, reduciendo el **bundle inicial** y mejorando el **LCP (Largest Contentful Paint)**.

En Angular 18:

```
{
  path: 'users',
  loadComponent: () => import('./users.component').then(m => m.UsersComponent)
}
```

Ventajas:

- Menor tiempo de carga inicial.
- Mejora del SEO (en SSR).
- Separación funcional del código por features.

Route-Scoped Providers

Novedad clave en Angular 17+ consolidada en 18.
Permiten declarar `providers` directamente en la definición de la ruta:

```
{
  path: 'users',
  loadComponent: () => import('./users.component').then(m => m.UsersComponent),
  providers: [UserService]
}
```

Estos servicios tienen un *scope* limitado: se crean al entrar a la ruta y se destruyen al salir.
Perfectos para caches, stores o contexto temporal.

Standalone Components

Componentes independientes que no requieren un NgModule.
Declarados con `standalone: true`.
Permiten importar directamente otros componentes, pipes o directivas sin pasar por un módulo intermedio.

```
@Component({
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  ...
})
```

Angular 18 usa **solo componentes standalone** por defecto.

Zoneless

Modo de ejecución sin `zone.js`.
Angular 18 permite desactivar Zone.js (a través de `bootstrapApplication` con `zone: 'noop'` o usando Signals).
En este modo, la detección de cambios se realiza de forma reactiva (gracias a Signals y recursos), no por parcheo global de APIs del navegador.

Ventajas:

- Rendimiento superior.
- Menor consumo de CPU y memoria.
- Más predecible.
- Integración más limpia con Web Components o frameworks externos.

Track Function (en control flow @for)

Función que Angular usa para optimizar el renderizado de listas (`@for ... track ...`).
Permite identificar los elementos por un ID único y evitar renders innecesarios.

```
@for user of users(); track user.id {
  <div>{{ user.name }}</div>
}
```

Code Splitting / Bundling

Proceso mediante el cual Angular CLI (Webpack o esbuild) divide el código en *chunks* (módulos independientes).

- **Bundle inicial**: contiene solo lo necesario para la vista inicial.
- **Lazy chunks**: se descargan dinámicamente al navegar.
- **Eager chunks**: se incluyen desde el arranque.

En Angular 18, la CLI usa *esbuild* por defecto y mejora el árbol de dependencias, reduciendo hasta un 30 % el tamaño del bundle.

LCP (Largest Contentful Paint)

Métrica de rendimiento web que mide el tiempo que tarda el contenido principal en mostrarse.
Angular 18 mejora el LCP gracias a:

- Lazy loading eficiente.
- Zone-less rendering.
- Signals que evitan renders innecesarios.

AbortController

API nativa de JS para cancelar operaciones asíncronas (`fetch`, `HttpClient` con `withFetch`).
Angular 18 lo usa internamente para abortar peticiones HTTP automáticamente (por ejemplo, cuando cambia un `rxResource`).

Providers Hierarchy

Jerarquía de inyección en Angular:

1. **Platform Injector** – global (browser).
2. **Root Injector** – creado en `bootstrapApplication`.
3. **Route Injector** – por ruta (`providers: []`).
4. **Component Injector** – local al componente.

Cada nivel puede sobrescribir dependencias de los superiores.

Reactive Composition

Patrón de programación basado en signals + observables + efectos.
Permite describir flujos de datos donde la UI se actualiza automáticamente según cambios de estado, sin intervención manual (sin `detectChanges()` ni zonas).

Change Detection sin Zone

En modo zoneless, Angular no intercepta los eventos del navegador.
Los componentes se actualizan automáticamente cuando cambia un `signal`, un `resource`, o se ejecuta un `computed`/`effect`.
Esto hace innecesario el `ChangeDetectorRef`.

Stream vs Loader (rxResource)

- **`stream:`** → acepta un *Observable* (flujo RxJS).
- **`loader:`** → acepta una función async que devuelve una *Promise*.
  Angular 18 los separa explícitamente (`resource` vs `rxResource`) para evitar confusiones entre ambos mundos.

Preloading Strategy

Técnica para cargar *lazy modules* de fondo después de la carga inicial, mejorando la navegación subsecuente sin afectar LCP.

Configurado en el router con `provideRouter(routes, withPreloading())`.