Sistemi Web Laboratorio Angular

Lezione 3: HttpClient • Dettaglio • Errori/Fallback

Obiettivo: collegare il catalogo alla **Mock API REST**, aggiungere il **dettaglio prodotto** e gestire **loading/error**.

Mock API & endpoint (json-server + Swagger)

Mock server locale (progetto `shop-mock-api`):

- Base URL API: `http://localhost:3000`
- Endpoint usati in L3:
  - `GET /products` — catalogo prodotti
  - `GET /products/{id}` — dettaglio prodotto (id **string**)
- Endpoint per lezioni successive:
  - `GET /orders`
  - `POST /orders`
- Documentazione:
  - Swagger UI: `http://localhost:3000/docs`

Modello `Product`

Useremo un modello coerente con la Mock API:

```
export interface Product {
  id: string;
  title: string;
  description: string;
  price: number;
  originalPrice: number;
  sale: boolean;
  thumbnail?: string;
  tags?: string[];
  createdAt: string;
}
```

- `id` è una **stringa** (es. `"p-001"`)
- `sale` + `originalPrice` permettono di gestire sconti
- `thumbnail` punta a un’immagine remota (placeholder)

Contesto progetto

Abbiamo già:

- pagina `/products` con:
  - card prodotto (titolo, prezzo, immagine)
  - filtri (testo, prezzo min/max, solo sconti)
  - sort e paginazione client-side
- `ProductService.list(): Observable<Product[]>`
  - in L2: sorgente dati in-memory (mock locali)
  - `products$` consumato con `AsyncPipe` nella lista

- cambiamo la sorgente → **Mock API REST**
- aggiungiamo la pagina `/products/:id`
- introduciamo il view model **loading/success/error**

STEP 1 — Catalogo → Mock API

Consegna

1. Abilita `HttpClient` a livello di app (bootstrap).
2. Modifica `ProductService` per usare la Mock API:
   - `GET http://localhost:3000/products`
3. Mantieni invariata la firma:
   - `list(): Observable<Product[]>`
4. Non modificare la pagina di lista:
   - `ProductListComponent` continua a usare `products$` + `AsyncPipe`

Output: la UI del catalogo è identica, ma i dati arrivano dalla Mock API.

STEP 1 — Setup HttpClient (app.config.ts)

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient } from '@angular/common/http';
import { AppComponent } from './app/app.component';

export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
    // altri provider...
  ],
});
```

- `provideHttpClient` importato da `@angular/common/http`
- aggiunto nei `providers` del `bootstrapApplication`

STEP 1 — ProductService.list() via HTTP

```typescript
// app/core/services/product.ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Product } from '../models/product';

@Injectable({ providedIn: 'root' })
export class ProductService {
  private readonly baseUrl = 'http://localhost:3000/api';

  constructor(private readonly http: HttpClient) {}

  list(): Observable<Product[]> {
    return this.http.get<Product[]>(`${this.baseUrl}/products`);
  }
}
```

STEP 1 — Debug

Per verificare:

1. Avvia la Mock API (nella cartella `shop-mock-api`):

   ```
   npm install
   npm run dev
   ```

2. Ricarica `/products`:
   - tab **Network** → **GET /products** → **200**
   - risposta JSON con ~50 prodotti
3. In caso di errori:
   - `HttpClient` non trovato → controlla `provideHttpClient()`
   - URL sbagliato → prova manualmente `http://localhost:3000/products` nel browser

STEP 1 — Output atteso

Comportamento atteso:

- UI `/products` **identica** (stesse card, filtri, sort, paginazione).
- Nessun errore in console.
- In Network:
  - `GET /products` chiamato alla prima apertura della pagina.
- Se spegni la Mock API:
  - la lista non carica (in STEP 3 gestiremo meglio errori/loading).

STEP 2 — Pagina dettaglio prodotto

1. Aggiungi una rotta `/products/:id`:
   - `ProductDetailPageComponent` standalone.
2. Estendi `ProductService` con:
   - `getById(id: string): Observable<Product>`
   - `GET http://localhost:3000/api/products/{id}`
3. Nel dettaglio:
   - leggi `id` dalla rotta
   - espone `product$: Observable<Product>`
4. Nella lista:
   - aggiungi bottone/link “Dettagli” che naviga a `/products/{{product.id}}`

STEP 2 — Routing `/products/:id`

```
ng generate component features/product-detail-page --standalone
```

```
// app/app.routes.ts
import { Routes } from '@angular/router';
import { ProductListPageComponent } from './features/products/product-list-page';
import { ProductDetailPageComponent } from './features/products/product-detail-page';

export const routes: Routes = [
  { path: 'products', component: ProductListPageComponent },
  { path: 'products/:id', component: ProductDetailPage },
  { path: '', pathMatch: 'full', redirectTo: 'products' },
];
```

STEP 2 — ProductService.getById

```typescript
// app/core/services/product.ts
getById(id: string): Observable<Product> {
  return this.http.get<Product>(`${this.baseUrl}/products/${id}`);
}
```

- `id` è una stringa, es. `"p-001"`
- l’URL finale sarà `http://localhost:3000/api/products/p-001`

STEP 2 — product-detail-page.ts

```
// app/features/products/product-detail/product-detail-page.ts
import { Component, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { switchMap, map } from 'rxjs';
import { ProductService } from '../../../core/services/product';
import { Product } from '../../../core/models/product';

@Component({
  selector: 'app-product-detail-page',
  standalone: true,
  templateUrl: './product-detail-page.html',
  imports: [RouterModule, NgIf, AsyncPipe, MatCardModule, MatButtonModule],
})
export class ProductDetailPageComponent {
  private route = inject(ActivatedRoute);
  private svc = inject(ProductService);

  readonly product$ = this.route.paramMap.pipe(
    map(params => params.get('id') as string),
    switchMap(id => this.svc.getById(id)),
  );
}
```

STEP 2 — product-detail-page.html

```html
<!-- app/features/products/product-detail/product-detail-page.html -->
@if (product$ | async; as product) {
  <mat-card>
    <mat-card-header>
      <mat-card-title>{{ product.title }}</mat-card-title>
      <mat-card-subtitle>
        @if (product.sale) {
          <span class="price-original">
            {{ product.originalPrice | currency:'EUR' }}
          </span>
          <span class="price-sale">
            {{ product.price | currency:'EUR' }}
          </span>
        } @else {
          {{ product.price | currency:'EUR' }}
        }
      </mat-card-subtitle>
    </mat-card-header>

    <img
      mat-card-image
      [src]="product.thumbnail"
      [alt]="product.title"
    />

    <mat-card-content>
      <p>{{ product.description }}</p>
    </mat-card-content>

    <mat-card-actions>
      <button mat-raised-button color="primary" routerLink="/products">
        Torna al catalogo
      </button>
    </mat-card-actions>
  </mat-card>
}
```

STEP 2 — Link “Dettagli” nella lista

```html
<!-- app/features/products/product-list/product-card.html -->
<article>
  <!-- contenuto card esistente -->
  <button [routerLink]="['/products', product.id]">
    Dettagli
  </button>
</article>
```

- usa `product.id` così com’è (string)
- nessuna logica aggiuntiva lato TS

STEP 3 — Stato HTTP con componente condiviso

Obiettivo

1. Creare un componente condiviso `HttpStateCardComponent` che gestisce:
   - stato **loading** (spinner Material),
   - stato **error** (card Material con messaggio),
   - contenuto di **success** via `<ng-content>`.
2. Usarlo nella pagina catalogo:
   - `ProductListPageComponent` passa solo lo **stream di stato**.
3. (Bonus) Aggiungere un **interceptor** per loggare / normalizzare gli errori HTTP.

STEP 3 — Tipo di stato condiviso

```
// app/shared/http/http-state.ts
export type ProductsState<T> =
  | { status: 'loading' }
  | { status: 'success'; data: T[] }
  | { status: 'error'; message: string };
```

- `status` descrive lo stato HTTP generico
- `data` contiene la lista prodotti filtrata/ordinata
- `message` è il testo da mostrare in caso di errore

```
ng generate component shared/http/http-state-card --standalone --inline-style
```

STEP 3 — `HttpStateCardComponent` (TS)

```
// app/shared/http/http-state-card/http-state-card.component.ts
import { Component, EventEmitter, Input, Output } from '@angular/core';
import { NgIf, NgSwitch, NgSwitchCase } from '@angular/common';
import { MatProgressSpinnerModule }
  from '@angular/material/progress-spinner';
import { MatCardModule } from '@angular/material/card';
import { ProductsState } from '../../http/http-state';

@Component({
  selector: 'app-http-state-card',
  standalone: true,
  imports: [NgIf, NgSwitch, NgSwitchCase, MatProgressSpinnerModule, MatCardModule],
  templateUrl: './http-state-card.component.html',
})
export class HttpStateCardComponent {
  @Input() state: ProductsState | null = null;
  @Output() retry = new EventEmitter<void>();

  onRetry(): void {
    this.retry.emit();
  }
}
```

STEP 3 — `HttpStateCardComponent` (HTML)

```html
<!-- app/shared/http/http-state-card/http-state-card.html -->
@switch (state?.status) {
  @case ('loading') {
    <section class="http-state-loading">
      <mat-progress-spinner
        mode="indeterminate"
        aria-label="Caricamento dati"
      ></mat-progress-spinner>
    </section>
  }

  @case ('error') {
    <mat-card class="http-state-error" aria-live="polite">
      <mat-card-title>Errore</mat-card-title>
      <mat-card-content>
        <p>{{ state?.message }}</p>
        <button mat-button (click)="onRetry()">
          Riprova
        </button>
      </mat-card-content>
    </mat-card>
  }

  @default {
    <!-- stato success: mostriamo il contenuto del parent -->
    <ng-content></ng-content>
  }
}
```

STEP 3 — Stream di stato nel catalogo

Nel componente lista prodotti costruiamo **solo** lo stream di stato:

```
// app/features/products/product-list/product-list.ts

// Sorgente prodotti: UNA sola chiamata HTTP condivisa
private readonly products$ = this.svc.list().pipe(
  shareReplay({ bufferSize: 1, refCount: true }),
);

// Stato HTTP per il componente condiviso
readonly httpState$ = this.products$.pipe(
  map(() => ({ status: 'success' } as const)),
  startWith({ status: 'loading' } as const),
  catchError(() =>
    of({ status: 'error', message: 'Impossibile caricare i prodotti.' } as const),
  ),
);
```

STEP 3 — Template lista con `HttpStateCardComponent`

```html
<!-- app/features/products/product-list/product-list.html -->
<app-http-state-card
  [state]="state$ | async"
  (retry)="reload()"
>
  <!-- stato SUCCESS: contenuto attuale della griglia -->
  <section class="catalog-grid">
    @for (product of (state$ | async)?.data ?? []; track product.id) {
      <!-- card Material del prodotto -->
      <app-product-card [product]="product"></app-product-card>
    }
  </section>
</app-http-state-card>
```

```typescript
reload(): void {
  ...
}
```

Interceptor per logging errori HTTP

Possiamo intercettare gli errori HTTP in un **interceptor**:

```
// app/core/interceptor/http-error.ts
import { HttpErrorResponse, HttpInterceptorFn } from '@angular/common/http';
import { catchError, throwError } from 'rxjs';

export const httpErrorInterceptor: HttpInterceptorFn = (req, next) =>
  next(req).pipe(
    catchError((error: HttpErrorResponse) => {
      console.error('HTTP error', error.status, error.message);
      // in futuro: mappare in un messaggio più leggibile o inviare a un servizio di logging
      return throwError(() => error);
    }),
  );
```

Registrazione nel bootstrap:

```
// main.ts
import { provideHttpClient, withInterceptors } from '@angular/common/http';
import { httpErrorInterceptor } from './app/core/http/http-error.interceptor';

bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient(withInterceptors([httpErrorInterceptor])),
  ],
});
```