Sistemi Web – Angular

Lezione 3: Router, HTTPCLient, Interceptor, Guard

Modulo 1 — Router & navigazione SPA

Obiettivo: capire come il **Router Angular** collega gli URL ai componenti e progettare il flow di pagine del nostro shop.

Perché ci serve un Router?

- In una **SPA** carichiamo una sola pagina HTML e cambiamo **vista** lato client.
- Dobbiamo mappare **URL leggibili** → componenti (Home, Catalogo, Dettaglio, Carrello…).
- Il Router si occupa di:
  - capire quale route corrisponde all'URL,
  - creare/distruggere i componenti giusti,
  - aggiornare la **barra degli indirizzi** senza ricaricare la pagina.

Abilitare il Router (recap prima leezione)

Per usare il Router in un’app standalone:

```
// main.ts
bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)]
});
```

Concetti chiave:

- `routes`: tabella di configurazione (array di `Route`).
- `provideRouter(...)`: registra il Router e le rotte a livello di app.
- Senza questa configurazione, `routerLink` / `<router-outlet>` **non funzionano**.

Definire le rotte del catalogo

```
// app.routes.ts
export const routes: Routes = [
  { path: '', redirectTo: 'products', pathMatch: 'full' },
  { path: 'products', component: ProductListComponent },
  { path: 'products/:id', component: ProductDetailComponent },
  { path: 'cart', component: CartPageComponent },
  { path: 'checkout', component: CheckoutPageComponent },
  { path: '**', redirectTo: 'products' }
];
```

- `path`: segmento di URL da abbinare.
- `component`: pagina da mostrare.
- `redirectTo` + `pathMatch: 'full'`: redirect dalla root `/` alla lista prodotti.
- `**`: route di fallback per URL non trovati (404 semplice).

Template base: `<router-outlet>` e link

```
<!-- app.component.html -->
<nav>
  <a routerLink="/products" routerLinkActive="active">Catalogo</a>
  <a routerLink="/cart" routerLinkActive="active">Carrello</a>
</nav>

<router-outlet></router-outlet>
```

- `<router-outlet>`: elemento dove il Router inserisce il componente della route attiva.
- `routerLink`: link dichiarativo alla route (meglio di `href`).
- `routerLinkActive`: aggiunge una classe CSS quando il link è attivo.

Navigazione imperativa dal codice

A volte vogliamo navigare da codice (es. dopo un click su card o dopo il checkout):

```
export class ProductListComponent {
  private router = inject(Router);
  // constructor(private router: Router) {}

  goToDetails(product: Product) {
    this.router.navigate(['/products', product.id]);
  }
}
```

- **Dichiarativo**: `routerLink` nei template.
- **Imperativo**: `router.navigate(...)` nel codice TS.
- Usa il navigatore imperativo per workflow (es. “ordine completato → vai a /orders/:id”).

Parametri di rotta (`/products/:id`)

Nel dettaglio prodotto dobbiamo leggere `:id` dall'URL:

```
export class ProductDetailComponent {
  private route = inject(ActivatedRoute);
  private productService = inject(ProductService);

  product$ = this.route.paramMap.pipe(
    map(params => params.get('id')!),
    switchMap(id => this.productService.getById(id))
  );
}
```

- `paramMap`: contiene i parametri dinamici (`:id`, `:slug`, ...).
- Usiamo RxJS (`map`, `switchMap`) per trasformare l'ID in una chiamata al servizio.

Query string per filtri e ordinamenti

URL come `/products?q=scarpe&sort=price` permettono di:

- condividere un link con **filtro e sort** già impostati;
- ripristinare lo stato dopo refresh;
- evitare stato persisto solo in memoria.

```
onFilterChange(filter: ProductFilter) {
  this.router.navigate([], {
    queryParams: { q: filter.search, sort: filter.sort },
    queryParamsHandling: 'merge'
  });
}
```

- `navigate([], ...)`: resta sulla stessa route, cambia solo i parametri.
- `queryParamsHandling: 'merge'`: unisce ai parametri già presenti.

Errori comuni & buona UX nel routing

- Dimenticare `provideRouter(routes)` → `routerLink` non funziona.
- Mancare la wildcard `**` → URL sbagliati mostrano una pagina vuota.
- Usare `href` invece di `routerLink` → ricarica completa della pagina.
- Usare `<button>` per navigazione “di tipo link” senza ruoli/aria corretti.

Buone pratiche:

- Preferisci `<a routerLink>...</a>` per la navigazione principale.
- Gestisci una pagina 404 dedicata invece di un semplice redirect.
- Cura il focus (es. spostarlo sul titolo della nuova pagina dopo la navigazione).

Recap Modulo 1 — Router & navigazione

Takeaway:

- Il Router mappa URL leggibili → componenti dell'app.
- `provideRouter(routes)` abilita il Router nell'app standalone.
- Rotte ben progettate: redirect iniziale, route con parametri, wildcard 404.
- Template base = `<nav>` con `routerLink` + `<router-outlet>` per il contenuto.
- Parametri di rotta e query string permettono link strutturati e stato condivisibile.

Modulo 2 — HttpClient & integrazione API

Dal mock al backend

Prima (L2): prodotti “hard-coded” in memoria.

```
@Injectable({ providedIn:'root' })
export class ProductStore {
  private productsSubject = new BehaviorSubject<Product[]>(MOCK_PRODUCTS);
  products$ = this.productsSubject.asObservable();
}
```

Ora: i dati arrivano da un’API REST (`/api/products` , `/api/products/:id` , `/api/orders` …).

Cos’è HttpClient

- Servizio fornito da `@angular/common/http`.
- Espone metodi per i principali **verbi HTTP**:
  - `get`, `post`, `put`, `patch`, `delete`, …
- Ogni metodo restituisce un `Observable<T>`:
  - la richiesta parte quando ci si **sottoscrive** (`subscribe`),
  - il valore emesso è già **JSON deserializzato**.

```
http = inject(HttpClient);
```

Abilitare HttpClient nell’app (standalone)

In una app standalone usiamo `provideHttpClient` in fase di bootstrap:

```
import { provideHttpClient } from '@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient()
  ]
});
```

- Da Angular moderno si preferisce `provideHttpClient()` a `HttpClientModule`.
- Così `HttpClient` diventa **iniettabile** in servizi e componenti.

DTO API vs modello di dominio

L'API espone un JSON con la propria struttura (DTO):

```ts
export interface ProductDto {
  id: string;
  title: string;
  description: string;
  priceInCents: number;
  image_url: string;
}
```

Nel frontend potremmo voler usare un modello **adattato** (`Product`):

- nomi diversi (`title` → `name`, `image_url` → `thumbnail`);
- prezzi in euro invece che in centesimi;
- proprietà derivate (tag, label per l'UI).

Servizio API per la lista prodotti (GET)

Creiamo un servizio dedicato alle chiamate HTTP dei prodotti:

```
@Injectable({ providedIn:'root' })
export class ProductApiService {
  http = inject(HttpClient);
  apiUrl = inject(API_URL);

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

- `get<ProductDto[]>` tipizza la risposta.
- Usiamo `API_URL` (InjectionToken visto in L2) per il base URL.

Mappare DTO → modello di dominio

Il servizio API restituisce `ProductDto`; lo store usa `Product`.

```
function mapProductDto(dto: ProductDto): Product {
  return {
    id: dto.id,
    name: dto.title,
    price: dto.priceInCents / 100,
    thumbnail: dto.image_url
  };
}

products$: Observable<Product[]> =
  this.api.list().pipe(map(dtos => dtos.map(mapProductDto)));
```

- Il mapping è **puro**: nessun effetto collaterale, facile da testare.
- Il componente consuma direttamente `Observable<Product[]>`.

Dettaglio prodotto: GET `/products/:id`

Nel servizio API:

```
getById(id: string): Observable<ProductDto> {
  return this.http.get<ProductDto>(`${this.#apiUrl}/products/${id}`);
}
```

Nel componente dettaglio (semplificato):

```
product$ = this.route.paramMap.pipe(
  map(params => params.get('id')!),
  switchMap(id => this.api.getById(id))
);
```

- Primo passo: leggere l'`id` dalla route.
- Secondo passo: chiamare l'API e trasformare il DTO in `Product`.

Operazioni di scrittura: POST `/orders`

Per inviare un ordine al backend:

```
create(order: NewOrder): Observable<OrderConfirmation> {
  return this.http.post<OrderConfirmation>(
    `${this.#apiUrl}/orders`,
    order
  );
}
```

- Il **body** (`order`) viene serializzato in JSON.
- Il tipo generico (`<OrderConfirmation>`) descrive la risposta attesa.

Errori comuni con HttpClient & API

```
//  Sbagliato: subscribe nel servizio
create(order: NewOrder) {
  this.http.post('/api/orders', order).subscribe();
}

//  Corretto: lasciare l'Observable al chiamante
create(order: NewOrder): Observable<OrderConfirmation> {
  return this.http.post<OrderConfirmation>('/api/orders', order);
}
```

Altri errori tipici:

- usare `any` invece di tipi espliciti (`ProductDto`, `NewOrder`, …);
- concatenare stringhe per URL ovunque, invece di riusare `API_URL`;
- ignorare completamente gli errori (404, 500, rete) → UX fragile.

Recap Modulo 2 — HttpClient & API

Takeaway:

- `HttpClient` si abilita con `provideHttpClient()` e si ottiene via DI (`inject(HttpClient)`).
- Le API espongono DTO (`ProductDto`), che mappiamo in modelli di dominio (`Product`).
- I servizi API incapsulano gli endpoint (`/products`, `/products/:id`, `/orders`).
- Usiamo i generici (`get<T>`, `post<T>`) per avere richieste/risposte **type-safe**.
- I servizi devono **restituire Observable**, non sottoscriversi internamente.

Domande di autoverifica:

1. Perché è utile distinguere tra `ProductDto` e `Product` nel frontend?
2. Dove metteresti la logica di mapping DTO → modello di dominio e perché?

Modulo 3 — Stato HTTP, errori & fallback UI

Obiettivo: modellare in modo chiaro **loading / success / error / empty** e dare all’utente un feedback chiaro quando qualcosa va storto.

Perché modellare lo stato HTTP?

Quando chiamiamo un’API succedono diverse cose:

- la richiesta parte → **loading**;
- la risposta arriva con dati → **success**;
- non ci sono dati (lista vuota) → **empty**;
- c’è un problema (404, 500, rete) → **error**.

Senza un modello esplicito rischiamo:

- tanti booleani sparsi (`isLoading`, `hasError`, `isEmpty`…);
- UI incoerente (spinner dimenticati, errori senza messaggio);
- codice difficile da capire e testare.

Unione discriminata per lo stato HTTP

Definiamo un tipo per rappresentare lo stato di una richiesta:

```
export type HttpState<T> =
  | { status: 'loading' }
  | { status: 'success'; data: T }
  | { status: 'empty' }
  | { status: 'error'; message: string };
```

Vantaggi:

- uno solo “interruttore” (`status`) per tutti i casi;
- impossibile avere combinazioni assurde (`loading` + `error` insieme);
- template più leggibile: `ngSwitch` su `status`.

Dal dato grezzo allo stato HTTP

Creiamo una funzione che converte un `Observable<T[]>` in `Observable<HttpState<T[]>>`:

```
function toHttpStateList<T>(
  source$: Observable<T[]>
): Observable<HttpState<T[]>> {
  return source$.pipe(
    map(items =>
      items.length === 0
        ? { status: 'empty' }
        : { status: 'success', data: items }
    ),
    startWith({ status: 'loading' as const }),
    catchError(() =>
      of({ status: 'error' as const, message: 'Errore di caricamento.' })
    )
  );
}
```

Esempio: ViewModel per la lista prodotti

Nel servizio/store della pagina lista prodotti:

```
@Injectable({ providedIn: 'root' })
export class ProductListStore {
  constructor(private api: ProductApiService) {}

  readonly state$: Observable<HttpState<Product[]>> =
    toHttpStateList(this.api.list());
}
```

- Lo store espone un solo `state$` ben tipizzato.
- Il componente consuma `state$` e si concentra su **come visualizzarlo**, non su RxJS.

Template: mostrare loading / success / empty / error

```
<section>
  @if (state$ | async; as state) {
    @if (state.status === 'loading') {
      <app-spinner></app-spinner>
    }
    @if (state.status === 'success') {
      <app-product-grid [products]="state.data"></app-product-grid>
    }
    @if (state.status === 'empty') {
      <p>
        Nessun prodotto trovato.
      </p>
    }
    @if (state.status === 'error') {
      <p role="alert">
        {{ state.message }}
      </p>
    }
  }
</section>
```

Gestire gli errori di HttpClient

Possiamo incapsulare la gestione degli errori nel servizio API o nello store:

```
this.state$ = this.api.list().pipe(
  map(dtos => dtos.map(mapProductDto)),
  toHttpStateList(),
);
```

Oppure separando i passi:

```
const data$ = this.api.list().pipe(
  map(dtos => dtos.map(mapProductDto))
);

this.state$ = toHttpStateList(data$);
```

- L’errore HTTP viene tradotto in `status: 'error'`.
- Il componente non deve conoscere dettagli di `HttpErrorResponse`.

Fallback UI: 404, offline, error page

Oltre allo stato locale della pagina, possiamo:

- mostrare una **pagina 404** se il prodotto non esiste;
- avere una pagina **errore generico** per problemi gravi;
- offrire un'azione di **retry** (bottone "Riprova").

```
<button *ngIf="state.status === 'error'" (click)="reload()">
  Riprova
</button>
```

```
reload() {
  this.state$ = toHttpStateList(this.api.list());
}
```

- Dare sempre una via d'uscita all'utente: "torna al catalogo", "riprovare", "contatta supporto".

Prima vs dopo: booleani sparsi vs stato unico

Prima (anti-pattern):

```
isLoading = false;
hasError = false;
products: Product[] = [];

load() {
  this.isLoading = true;
  this.hasError = false;
  this.api.list().subscribe({
    next: products => {
      this.products = products;
      this.isLoading = false;
    },
    error: () => {
      this.hasError = true;
      this.isLoading = false;
    }
  });
}
```

Dopo (ViewModel HTTP + Observable):

```
state$ = toHttpStateList(
  this.api.list().pipe(map(dtos => dtos.map(mapProductDto)))
);
```

- Meno stati mutabile, meno rami, più facile da testare.

Recap Modulo 3 — Stato HTTP & fallback UI

Takeaway:

- Ogni chiamata HTTP ha almeno 4 stati: **loading, success, empty, error**.
- Una **union discriminata** (`HttpState<T>`) rende espliciti e mutuamente esclusivi questi stati.
- Lo store/servizio espone un unico `state$` → il componente decide come mostrarlo.
- Messaggi di errore e fallback (404, retry, redirect) devono essere pensati per l’utente, non solo per il dev.
- L’UX migliora quando l’utente sa sempre cosa sta succedendo.

Modulo 4 — Interceptor & Guards

Cosa sono gli HTTP Interceptor?

Un **Interceptor** è una funzione che:

- si infila tra `HttpClient` e il server;
- vede tutte le richieste e le risposte;
- può:
  - aggiungere header (es. `Authorization`, `Accept-Language`),
  - fare logging / timing,
  - intercettare e trasformare **errori**,
  - mostrare/nascondere spinner globali.

> Pensalo come un “middleware HTTP” lato frontend.

Dichiarare un functional interceptor

Esempio: aggiungere un header `Authorization` se l'utente è loggato.

```
export const authInterceptor: HttpInterceptorFn =
  (req, next) => {
    const auth = inject(AuthService);
    const token = auth.getToken();

    if (!token) return next(req);

    const authReq = req.clone({
      setHeaders: { Authorization: `Bearer ${token}` }
    });

    return next(authReq);
  };
```

- Tipo `HttpInterceptorFn`: nuova forma funzionale dell'interceptor.

Registrare un interceptor con `provideHttpClient`

Nel bootstrap dell'app (standalone):

```
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient(
      withInterceptors([authInterceptor])
    )
  ]
});
```

- `provideHttpClient(...)` abilita HttpClient.
- `withInterceptors([...])` registra la catena degli interceptor.

Interceptor per la gestione globale degli errori

Possiamo centralizzare la trasformazione degli errori HTTP:

```
export const errorInterceptor: HttpInterceptorFn =
  (req, next) => next(req).pipe(
    catchError(error => {
      console.error('Errore HTTP', error);
      return throwError(() =>
        new Error('Errore di rete, riprova più tardi.')
      );
    })
  );
```

- Log tecnico in console.
- Messaggio generico più “pulito” verso l’app/utente.

Perché usare Interceptor?

Esempi pratici:

- **Auth**: aggiungere il token per tutte le chiamate protette.
- **Lingua**: header `Accept-Language: it-IT` per il backend.
- **UX**: interceptor “spinner” che:
  - incrementa un contatore richieste in corso,
  - mostra un loader globale finché il contatore > 0.
- **Telemetry/log**: loggare durata delle richieste, endpoint più lenti, ecc.

Cosa sono le Route Guard?

Le **Route Guard** sono i “guardiani” del Router: decidono cosa succede **prima** che una rotta venga attivata o scelta.

Principali tipi:

- `CanActivate` → può entrare in questa rotta?
- `CanMatch` → questa rotta può essere scelta per questo URL?
- `CanDeactivate` → posso uscire da questo componente?
- `Resolve` → posso pre-caricare dati prima di entrare?

Tutte lavorano **prima** che il componente venga creato (o prima che la rotta venga scelta).

Cosa restituisce una Guard

Una Guard può restituire:

- `boolean`
  - `true` → navigazione consentita
  - `false` → navigazione bloccata
- `UrlTree` → navigazione **reindirizzata** (redirect)
- oppure le versioni asincrone:
  - `Observable<boolean | UrlTree>`
  - `Promise<boolean | UrlTree>`

Esempi d’uso:

- sync → controllo su stato già in memoria (es. `isLoggedIn`, `hasRole('admin')`)
- async → chiamata al backend, lettura da storage esterno, ecc.

`CanActivate`

Functional guard `CanActivateFn` (controllo login):

```
export const authGuard: CanActivateFn =
  (route, state) => {
    const auth = inject(AuthService);
    if (auth.isLoggedIn()) {
      return true;
    }
    return false;
  };
```

Parametri:

- `route` → info sulla rotta che si vuole attivare
- `state` → info sulla navigazione (`url` di destinazione, ecc.)

Uso tipico: bloccare l'accesso a pagine protette (checkout, profilo, ordini, area admin).

`CanActivate` con redirect

```
export const authGuard: CanActivateFn =
  (route, state) => {
    const auth = inject(AuthService);
    const router = inject(Router);

    if (auth.isLoggedIn()) {
      return true;
    }

    return router.createUrlTree(
      ['/login'],
      { queryParams: { redirectUrl: state.url } }
    );
  };
```

- flusso di navigazione **dichiarativo** (niente `navigate` dentro la guard);
- possiamo usare `redirectUrl` per riportare l'utente alla pagina originale dopo il login.

Applicare `CanActivate` alle rotte

Configurazione nel file delle rotte:

```
export const routes: Routes = [
  { path: 'cart', component: CartPageComponent },
  {
    path: 'checkout',
    component: CheckoutPageComponent,
    canActivate: [authGuard]
  }
];
```

Comportamento:

- utente loggato → entra in `/checkout`;
- utente non loggato → viene reindirizzato a `/login?redirectUrl=/checkout`.

`CanMatch`: scegliere la rotta giusta

`CanMatch` decide se una rotta può essere **considerata** per un certo URL:

```
export const adminMatchGuard: CanMatchFn =
  (route, segments) => {
    const auth = inject(AuthService);
    return auth.hasRole('admin');
  };
```

Configurazione:

```
{
  path: 'admin',
  canMatch: [adminMatchGuard],
  loadComponent: () => import('./admin.page')
}
```

Se `hasRole('admin')` è `false`:

`CanDeactivate`: uscire dalla pagina in sicurezza

`CanDeactivate` protegge pagine con **form non salvati** (checkout, profilo, ecc.):

```
export const unsavedChangesGuard: CanDeactivateFn<CheckoutPageComponent> =
  (component, currentRoute, currentState, nextState) => {
    if (!component.hasUnsavedChanges()) return true;

    return window.confirm(
      'Ci sono modifiche non salvate. Vuoi davvero uscire dalla pagina?'
    );
  };
```

- Se l’utente annulla, la navigazione viene bloccata.
- Previene perdite di dati compilati.

`Resolve`: pre-caricare i dati

`Resolve` permette di caricare i dati **prima** di attivare la rotta:

```
export const productResolver: ResolveFn<Product> =
  (route, state) => {
    const api = inject(ProductApiService);
    const id = route.paramMap.get('id')!;
    return api.getById(id);
  };
```

Configurazione della rotta:

```
{
  path: 'products/:id',
  component: ProductDetailComponent,
  resolve: { product: productResolver }
}
```

Nel componente: i dati arrivano tramite `ActivatedRoute.data`.

Combinare più Guard sulla stessa route

Possiamo usare più guard per controlli diversi:

```
{
  path: 'checkout',
  component: CheckoutPageComponent,
  canActivate: [authGuard, cartNotEmptyGuard]
}
```

```
export const cartNotEmptyGuard: CanActivateFn =
  (route, state) => {
    const cart = inject(CartStore);
    return cart.hasItems();
  };
```

- Se una qualsiasi guard restituisce `false` o un `UrlTree`, la navigazione viene bloccata o reindirizzata.
- Ogni guard fa una cosa sola

Guard vs logica nel componente

Quando usare una Guard?

- per decisioni che influenzano **URL e accesso alla pagina**:
  - login obbligatorio,
  - ruoli/permessi,
  - carrello vuoto → torna al catalogo,
  - form non salvato → conferma prima di uscire.

Quando NON usare una Guard?

- per logiche di **presentazione**:
  - mostrare o meno un blocco di UI,
  - messaggi “carrello vuoto” dentro la pagina,
  - layout diversi a seconda del tipo di utente.

Recap Modulo 4 — Interceptor & Guards

- Gli **Interceptor** sono perfetti per logiche HTTP trasversali (auth, lingua, logging, gestione errori).
- Con `provideHttpClient(withInterceptors([...]))` registriamo interceptor funzionali a livello globale.
- I **Guards** (`CanActivate`, `CanMatch`, …) controllano l'accesso alle route **prima** di creare il componente.