Sistemi Web · Angular

Form · Reactive Forms · Production Optimizations

Modulo 1 — Form in Angular

Panorama generale e scelte architetturali

Perché i form nelle SPA sono “difficili”

- Molta **logica di business**:
  - regole, vincoli, formati, dipendenze fra campi.
- Aspettative alte su **UX**:
  - errori chiari, suggerimenti, salvataggio, undo.
- Vincoli tecnici:
  - chiamate HTTP, debounce, validazioni async, errori di rete.
- State management complicato:
  - dirty / touched / pending / disabled,
  - valori parziali, step multipli.

Cosa deve fare bene un form

- Raccogliere **dati validi**:
  - vincoli tecnici (email, CF, telefono),
  - vincoli di business (min/max, combinazioni ammesse).
- Comunicare **stato**:
  - cosa è obbligatorio, cosa è opzionale,
  - cosa è in errore, cosa è in caricamento.
- Integrarsi con il backend:
  - trasformare il modello del form → **DTO** per API,
  - gestire successo/fallimento e riprovare.
- Rispettare **a11y e tastiera**:
  - label associati, ordine di tab, messaggi leggibili anche da screen reader.

Le due famiglie di form in Angular

Angular offre due approcci principali:

- **Template-driven forms**
  - Logica dichiarata **nel template** (`ngModel`, `ngForm`).
  - Pensati per form piccoli / semplici.
- **Reactive Forms**
  - Modello di form esplicito in **TypeScript** (`FormControl`, `FormGroup`, `FormArray`).
  - Più scalabili, testabili e riusabili.

Entrambi:

- usano la stessa “base” (`AbstractControl`),
- espongono stati (`valid`, `dirty`, `pending`, …),

Esempio form semplice — Template-driven

```
<form #checkoutForm="ngForm" (ngSubmit)="onSubmit(checkoutForm.value)">
  <label>Nome
    <input name="fullName" ngModel required />
  </label>
  <button type="submit" [disabled]="checkoutForm.invalid">
    Conferma ordine
  </button>
</form>
```

Caratteristiche:

- Il **modello del form** è creato dalle direttive nel template (`ngForm` , `ngModel`).
- Nel TS ricevi un oggetto “piatto” con i valori.
- Meno boilerplate, ma logica distribuita nei template.

Esempio form semplice — Reactive Forms

```ts
import { FormBuilder, Validators } from '@angular/forms';

checkoutForm = this.fb.group({
  fullName: this.fb.nonNullable.control('', { validators: [Validators.required] }),
});

constructor(private fb: FormBuilder) {}

onSubmit() {
  console.log(this.checkoutForm.value);
}
```

```html
<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">
  <label>Nome
    <input formControlName="fullName" />
  </label>
  <button type="submit" [disabled]="checkoutForm.invalid">
    Conferma ordine
  </button>
</form>
```

Reactive vs Template-driven — differenze chiave

Template-driven

- Setup del modello: **implicito** nel template.
- Data flow: legato al ciclo di change detection.
- Validazione: direttive (`required`, `ngModel`, …).
- Test: serve spesso interagire con il DOM.

Reactive

- Setup del modello: **esplicito** in TS (`FormGroup`, `FormControl`).
- Data flow: accesso ai valori **sincrono** e prevedibile.
- Validazione: funzioni pure, più facile da riusare e testare.
- Test: si possono testare i form **senza template**, come normali oggetti.

Come scegliere l’approccio

Reactive Forms → **default** quando:

- i form sono **centrali** nell’app (es. backoffice, checkout),
- servono validazioni complesse o cross-field,
- ci sono **step multipli**, wizard, dinamiche,
- vuoi integrare RxJS, Undo/Redo, salvataggio in bozza.

Template-driven può bastare quando:

- c’è un singolo form molto semplice,
- la logica è quasi tutta nel template,
- la priorità è scrivere pochissimo codice “usa e getta”.

Errori comuni nell’architettura dei form

- Mettere **tutta la logica** in un unico component “enorme”.
- Mischiare:
  - logica di validazione,
  - chiamate HTTP,
  - presentazione (messaggi, colori, icone).
- Affidare tutta la validazione al **backend**, senza feedback lato client.
- Ignorare lo **stato** del form:
  - niente gestione di dirty/touched,
  - nessun blocco del doppio submit,
  - nessun indicatore di caricamento.

Modulo 2 — Fondamenti di Reactive Forms

Form model, controlli base, tipizzazione e flusso dati

Cos’è una Reactive Form

- Le Reactive Forms sono un approccio **model-driven**: il form è prima di tutto un **modello in TypeScript**, non il template.
- Ogni input del form è rappresentato da un oggetto (`FormControl`, `FormGroup`, `FormArray`).
- Lo stato del form (valore, validità, stato utente) è mantenuto e aggiornato dal **form model**.
- Il template si limita a **fare binding** al modello usando direttive (`[formControl]`, `formGroup`, `formControlName`).

Componenti Chiave

- `AbstractControl` è la **classe base** da cui derivano tutti i controlli: `FormControl`, `FormGroup`, `FormArray`, `FormRecord`.
- `FormControl`
  - rappresenta un **singolo campo** (input, select, textarea...),
  - traccia valore, validità, interazioni utente (touched, dirty).
- `FormGroup`
  - è un insieme **nominale** di controlli (es. sezione indirizzo).
- `FormArray`
  - collezione **indicizzata** di controlli (es. lista di tag o indirizzi multipli).

Reactive Forms come “single source of truth”

- Il **form model** è la fonte di verità:
  - il template si aggiorna quando cambia il modello,
  - il modello si aggiorna quando l’utente interagisce.
- Flusso dati:
  - `value` → fotografia immediata del valore corrente,
  - `valueChanges` → stream di cambiamenti nel tempo.
- Ogni controllo espone:
  - `value`, `valid`, `errors`, `dirty`, `touched`, `disabled`, `pending`…

Reactive Forms e TypeScript: typed forms

- Dalla v14 le Reactive Forms sono **tipizzate**: i controlli sono generici.
- Possiamo specificare il **tipo di valore** per ogni controllo:
  - `FormControl<string>`, `FormControl<number>`, ecc.
- `FormBuilder.nonNullable` aiuta a creare form **senza** `null` nelle proprietà:
  - modelli più sicuri, meno `!` e cast.
- Vantaggi:
  - IntelliSense migliore, errori a **compile-time** invece che a runtime,
  - refactoring più facile su campi e nomi.

Primo esempio — un singolo FormControl

```
nameControl = new FormControl<string>('', { nonNullable: true });

get name(): string {
  return this.nameControl.value;
}

onClearName() {
  this.nameControl.setValue('');
}
```

```
<label for="name">Nome completo</label>
<input id="name" type="text" [formControl]="nameControl" />
<p>Valore corrente: {{ name }}</p>
<button type="button" (click)="onClearName()">Svuota</button>
```

Costruire un FormGroup tipizzato

```
private readonly fb = inject(FormBuilder);

readonly checkoutForm = this.fb.nonNullable.group({
  fullName: '',
  email: '',
});

onSubmit() {
  const value = this.checkoutForm.value;
  // { fullName: string; email: string }
  console.log('Dati checkout', value);
}
```

FormArray — quando i campi sono “ripetibili”

```
readonly tags = this.fb.nonNullable.array([
  this.fb.control('Running'),
]);

addTag() {
  this.tags.push(this.fb.control('Nuovo tag'));
}
```

```
<div formArrayName="tags">
  <span *ngFor="let c of tags.controls; index as i">
    {{ c.value }}
  </span>
</div>
<button type="button" (click)="addTag()">Aggiungi tag</button>
```

Osservare il form nel tempo: valueChanges & status

```
ngOnInit() {
  this.checkoutForm.valueChanges.subscribe(value => {
    console.log('Form cambiato', value);
  });
}

get isValidAndDirty(): boolean {
  return this.checkoutForm.valid && this.checkoutForm.dirty;
}
```

Prima/Dopo — senza modello vs Reactive Forms

Prima (senza Reactive Forms)

- Campi letti con `querySelector` o `ngModel` sparsi nel template.
- Validazione distribuita in funzioni random nel componente.
- Difficile capire “lo stato del form” in un singolo punto.

Dopo (con Reactive Forms)

- Un **unico modello** (`FormGroup`) che contiene tutti i campi e stati.
- Validazione, valore e stato concentrati negli oggetti `FormControl` / `FormGroup`.
- Template più “stupido”: mostra errori e stato leggendo proprietà del modello.

Modulo 3 — Validazione con Reactive Forms

Stati, validator built-in, custom e cross-field

Stati dei controlli e ciclo di vita della validazione

Ogni `AbstractControl` espone stati importanti:

- **valid / invalid**
  - `valid = true` se **tutti** i validator sono soddisfatti.
- **pristine / dirty**
  - `pristine`: mai modificato dall'utente.
  - `dirty`: l'utente ha cambiato il valore almeno una volta.
- **touched / untouched**
  - `touched`: l'utente ha "visitato" il campo (focus + blur).
- **disabled / enabled**
  - controlli disabilitati **non** partecipano alla validazione.
- **pending**
  - ci sono validazioni async in corso (es. chiamata API).

Validator sincroni built-in

Angular fornisce molti validator pronti per i casi tipici:

- `Validators.required` — il campo non può essere vuoto.
- `Validators.email` — formato email generico.
- `Validators.min`, `Validators.max` — limiti numerici.
- `Validators.minLength`, `Validators.maxLength` — lunghezza stringa.
- `Validators.pattern` — espressione regolare personalizzata.

Caratteristiche:

- Sono **funzioni pure**: stesso input → stesso output.
- Restituiscono:
  - `null` se il valore è valido,
  - un oggetto `ValidationErrors` se invalido (es. `{ required: true }`).

Validator custom e cross-field

Per le regole specifiche di dominio:

- **Validator custom su controllo**
  - es. codice fiscale, CAP italiano, codice coupon.
  - funzione che accetta un `AbstractControl` e restituisce `ValidationErrors | null`.
- **Validator su FormGroup (cross-field)**
  - controllano la **relazione** tra più campi:
    - password = conferma,
    - data inizio < data fine,
    - stessa città per fatturazione/spedizione.

Validator custom e cross-field

- **Async validator**
  - usano HTTP / servizi esterni:
    - email già registrata,
    - coupon valido,
    - username disponibile.

Pattern comune:

- ogni tipo di errore ha una **chiave** (es. `invalidCoupon`, `passwordsMismatch`),
- il template mappa queste chiavi con **messaggi leggibili**.

Esempio — Validator built-in su form di checkout

```ts
// component.ts
readonly checkoutForm = this.fb.nonNullable.group({
  fullName: ['', [Validators.required, Validators.minLength(3)]],
  email: ['', [Validators.required, Validators.email]],
});
```

```html
<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">
  <label>Nome completo
    <input formControlName="fullName" />
  </label>

  <label>Email
    <input formControlName="email" type="email" />
  </label>

  <button type="submit" [disabled]="checkoutForm.invalid">
    Conferma ordine
  </button>
</form>
```

Mostrare gli errori nel template

```
get fullNameCtrl() {
  return this.checkoutForm.controls.fullName;
}
```

```
<label>Nome completo
  <input formControlName="fullName" />
</label>

<p *ngIf="fullNameCtrl.invalid && (fullNameCtrl.dirty || fullNameCtrl.touched)">
  <span *ngIf="fullNameCtrl.errors?.['required']">
    Il nome è obbligatorio.
  </span>
  <span *ngIf="fullNameCtrl.errors?.['minlength']">
    Minimo 3 caratteri.
  </span>
</p>
```

Esempio — Validator custom su controllo (CAP italiano)

```
function italianZip(control: AbstractControl<string>): ValidationErrors | null {
  const value = control.value ?? '';
  const ok = /^[0-9]{5}$/.test(value);
  return ok ? null : { italianZip: true };
}

readonly addressForm = this.fb.nonNullable.group({
  zipCode: ['', [Validators.required, italianZip]],
});
```

```
<input formControlName="zipCode" />

<p *ngIf="addressForm.controls.zipCode.errors?.['italianZip']">
  Inserisci un CAP a 5 cifre.
</p>
```

Esempio — Validator cross-field su FormGroup

```ts
function passwordsMatch(group: AbstractControl): ValidationErrors | null {
  const { password, confirm } = group.value as { password: string; confirm: string };
  return password === confirm ? null : { passwordsMismatch: true };
}

readonly passwordForm = this.fb.nonNullable.group(
  {
    password: ['', Validators.required],
    confirm: ['', Validators.required],
  },
  { validators: [passwordsMatch] }
);
```

```html
<div [formGroup]="passwordForm">
  <input formControlName="password" type="password" />
  <input formControlName="confirm" type="password" />

  <p *ngIf="passwordForm.errors?.['passwordsMismatch'] && passwordForm.touched">
    Le password non coincidono.
  ...
```

Validator asincroni

Schema tipico (semplificato):

```
function couponValidator(service: CouponService) {
  return (control: AbstractControl<string>) =>
    control.value === ''
      ? of(null)
      : service.check(control.value).pipe(
          map(isValid => (isValid ? null : { invalidCoupon: true }))
        );
}
```

Validator asincroni

Uso nel form:

- configurato come **async validator** del controllo coupon,
- Angular imposta lo stato del controllo a `pending` finché la chiamata non termina,
- quando arriva la risposta:
  - `null` → campo valido,
  - `{ invalidCoupon: true }` → campo invalid.

Prima/Dopo & Errori comuni nella validazione

Prima (anti-pattern frequenti)

- Tutta la validazione fatta “a mano” dentro `onSubmit()`.
- Messaggio generico “Form non valido” senza indicazioni sui campi.
- Niente distinzione tra errori lato client e lato server.

Dopo (con Reactive Forms)

- Le regole vivono nei **validator** di controllo/gruppo.
- Messaggi chiari per ogni tipo di errore, basati sulle chiavi di `errors`.
- Validazione client complementare a quella server:
  - controlli rapidi lato client,
  - vincoli definitivi lato backend.

Recap Modulo 4

Recap

- Ogni controllo espone stati (`valid`, `dirty`, `touched`, `pending`, ...) che guidano la UI.
- I validator built-in coprono i casi comuni (`required`, `email`, `minLength`, ...).
- I validator custom permettono di modellare regole di dominio (CAP, CF, coupon, ...).
- I validator cross-field vivono a livello di `FormGroup` e controllano relazioni tra campi.
- I validator async integrano servizi esterni e usano lo stato `pending`.

Modulo 4 — UX dei form

Errore, feedback, submit e integrazione con il flusso utente

Perché la UX dei form è cruciale

- I form sono spesso il passo più **sensibile**:
  - login, registrazione, pagamento, ordine.
- Errori gestiti male portano a:
  - frustrazione, abbandono, ticket di supporto.
- Una buona UX di form:
  - riduce errori,
  - aumenta **fiducia** (soprattutto nei pagamenti),
  - velocizza il completamento.

Quando mostrare gli errori

Strategie comuni:

- **Solo on submit**
  - pro: non stressa subito l’utente,
  - contro: lista di errori tutti insieme alla fine.
- **Subito on change**
  - pro: feedback immediato,
  - contro: può risultare aggressivo (“sei in errore” mentre stai ancora scrivendo).
- **Ibrida (consigliata)**
  - per ogni campo: mostra errore se `invalid && (dirty || touched)`,
  - all’invio: marca tutti i campi come touched per far emergere gli errori.

Messaggi di errore chiari e specifici

Linee guida:

- Evita: “Campo non valido”.
- Preferisci:
  - **cosa** è sbagliato (“Inserisci un’email valida”),
  - **come** correggerlo (“Minimo 3 caratteri”, “Usa solo numeri”).
- Un solo messaggio principale per volta per campo.
- Usa microcopy coerente in tutta l’app.

```
<p *ngIf="emailCtrl.invalid && (emailCtrl.dirty || emailCtrl.touched)">
  <span *ngIf="emailCtrl.errors?.['required']">
    L'email è obbligatoria.
  </span>
  <span *ngIf="emailCtrl.errors?.['email']">
    Inserisci un indirizzo email valido (es. nome@dominio.it).
  </span>
```

Gestire la submit: loading e doppio invio

Pattern tipico nel component:

```
isSubmitting = false;

onSubmit() {
  if (this.checkoutForm.invalid) {
    this.checkoutForm.markAllAsTouched();
    return;
  }

  this.isSubmitting = true;
  this.orderService.create(this.checkoutForm.value).subscribe({
    next: () => { this.isSubmitting = false; /* success */ },
    error: () => { this.isSubmitting = false; /* mostra errore */ },
  });
}
```

Errori globali e integrazione con il backend

Due livelli di errore:

- **Errore di campo**
  - es. email mancante, CAP non valido,
  - gestito via validator e `errors` sul controllo.
- **Errore globale**
  - problemi lato server (es. pagamento rifiutato, stock esaurito),
  - da mostrare in un’area visibile sopra il form.

Pattern:

- mantieni un `formErrorMessage` nel component per errori globali,
- non cancellare i valori già inseriti,

Modulo 5 — Performance base

OnPush & `trackBy` su liste e form

Change Detection: cosa fa Angular

- Angular mantiene un **albero di componenti**.
- Strategia **default**:
  - ad ogni evento / timer / HTTP, controlla quasi tutto l'albero,
  - per vedere se lo stato è cambiato e aggiornare il DOM.
- Su app grandi:
  - molti controlli ripetuti,
  - liste che vengono ricreate spesso,
  - componenti che si aggiornano anche quando i dati non cambiano.

Obiettivo:

- **evitare controlli inutili**,
- aggiornare solo dove serve.

OnPush: cos’è e cosa cambia

`ChangeDetectionStrategy.OnPush` dice ad Angular:

- “Questo componente va ricontrollato solo se”:
  - riceve **nuovi @Input()** (nuovo riferimento),
  - gestisce un **evento** dalla sua vista,
  - un Observable usato con `async` pipe emette un nuovo valore.
- Vantaggi:
  - meno componenti ricontrollati → migliore **performance**,
  - flusso dati più **prevedibile** (immutabilità + stream),
  - ideale per liste, card e componenti di presentazione.

Richiede:

- pensare in modo più **immutabile** (nuovi array/oggetti invece di mutarli).

Esempio — componente lista prodotti con OnPush

```
@Component({
  selector: 'app-product-list',
  standalone: true,
  templateUrl: './product-list.html',
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class ProductListComponent {
  @Input({ required: true }) products!: readonly Product[];
}
```

```
<ul>
  <li *ngFor="let p of products">
    {{ p.title }} — {{ p.price | currency:'EUR' }}
  </li>
</ul>
```

Perché OnPush aiuta liste e form

- Liste (catalogo, carrello):
  - meno DOM ricreato quando cambia solo una parte della lista,
  - perfetto insieme a `trackBy`.
- Form:
  - componenti di input riusabili che reagiscono solo a cambiamenti reali,
  - meno rendering quando cambiano altri pezzi della pagina.
- In generale:
  - rende più evidente *da dove* arrivano i dati,
  - incoraggia pattern di **stato esplicito** (store, servizi, RxJS).

Ottimizzare *ngFor con trackBy

Per default, `*ngFor`:

- usa l'identità dell'oggetto per rilevare cambiamenti,
- spesso ricrea l'intero DOM della lista quando cambia qualcosa,
- questo può essere costoso su liste lunghe.

Con `trackBy`:

- forniamo una **funzione** che restituisce un id stabile per ogni item,
- Angular riusa i nodi DOM esistenti e aggiorna solo quelli che cambiano,
- meno flicker, migliore performance e a11y.

Esempio — trackBy su lista prodotti

```
readonly products: readonly Product[] = [];

trackByProductId(index: number, product: Product): string {
  return product.id;
}
```

```
<li *ngFor="let p of products; trackBy: trackByProductId">
  {{ p.title }} — {{ p.price | currency:'EUR' }}
</li>
```

Errori comuni con OnPush + trackBy

- Usare OnPush ma:
  - **mutare** l'array (`push`, `splice`) invece di crearne uno nuovo,
  - aspettarsi aggiornamenti "magicamente" senza nuovi riferimenti.
- Dimenticare `trackBy` su liste grandi:
  - scroll che salta,
  - componenti figli che vengono ricreati.
- Iniettare servizi che mutano lo stato internamente senza emettere eventi/stream.

Pattern consigliato

- dati esposti come `Observable<readonly Product[]>`,
- `async` pipe nel template,
- OnPush + `trackBy` sugli elenchi.

Modulo 6 — Build di produzione & Docker

Dal `ng serve` al container minimale

`ng serve` vs `ng build`

- `ng serve`:
  - compila in memoria,
  - pensato per lo **sviluppo** (sourcemap, error overlay, HMR),
  - non genera file pronti per deploy.
- `ng build`:
  - compila il progetto in **file statici** nella cartella `dist/`,
  - applica ottimizzazioni (bundling, minify, rimozione dead code) in configurazione produzione.

```
ng build --configuration production
```

Configurazioni di build

Nel workspace Angular (`angular.json`):

- target `build` ha una sezione `configurations`,
- di default ci sono almeno:
  - `development`,
  - `production`.

La configurazione produzione:

- abilita ottimizzazioni aggressive,
- può usare file `environment` diversi,
- è quella usata di default da `ng build` nelle versioni recenti.

Idea chiave:

Output di `ng build` e deploy statico

Dopo `ng build --configuration production` trovi:

- una cartella `dist/<nome-app>/` con:
  - `index.html`,
  - bundle JS/CSS minificati,
  - assets (immagini, font, ecc.).

Per un deploy manuale:

1. creare la build di produzione,
2. copiare la cartella di output su:
   - server web (es. Nginx, Apache),
   - servizio static hosting / CDN.

Dockerfile multi-stage minimale

```
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build -- --configuration production

FROM nginx:stable-alpine
COPY --from=build /app/dist/shop-app/browser /usr/share/nginx/html
```

Idea:

- primo stage (`node`) → fa la build dell'app,
- secondo stage (`nginx`) → serve i file statici generati.

Comandi Docker essenziali

Build dell'immagine:

```
docker build -t shop-app .
```

Esecuzione locale:

```
docker run -p 8080:80 shop-app
```

- L'app sarà disponibile su `http://localhost:8080`.
- Nginx serve l'`index.html` e i bundle generati da Angular.
- Le API (mock o reali) devono essere raggiungibili dalla SPA (CORS, URL corretti...).

Modulo 7 — Testing in Angular

Unit test per funzioni, servizi, HTTP e form

Perché testare un’app Angular

- Evitare regressioni:
  - checkout che smette di funzionare dopo una refactor,
  - filtri catalogo che non filtrano più.
- Documentare il comportamento atteso:
  - “questo service calcola il totale così”
  - “questo form è valido solo in queste condizioni”.
- Sbloccare refactor più sereni:
  - cambiare internamente componenti e servizi,
  - mantenendo intatti i contratti.

Tipologie di test

Livelli tipici:

- funzioni (Unit test)
- servizi
- componenti
- E2E

Strumenti principali in Angular

- **Test runner + test framework**
  - di solito Jasmine + Karma (o simili) preconfigurati dal CLI.
- **Angular TestBed**
  - crea un “mini modulo di test” che emula un `@NgModule`,
  - permette di configurare provider, import e componenti.
- **ComponentFixture**
  - wrapper per interagire con il component in test,
  - accede a istanza, template e trigger di change detection.
- **HttpTestingController**
  - finge il backend per `HttpClient`,

Testare funzioni (senza Angular)

```ts
export function calcTotal(prices: number[]): number {
  return prices.reduce((sum, p) => sum + p, 0);
}
```

```ts
describe('calcTotal', () => {
  it('somma tutti i prezzi', () => {
    const result = calcTotal([10, 5, 2]);
    expect(result).toBe(17);
  });
});
```

- Niente Angular, solo Jasmine:
  - `describe` raggruppa test,
  - `it` definisce un caso,
  - `expect` esprime l'asserzione.

Test di un servizio senza HTTP

```
describe('CartService', () => {
  let service: CartService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [CartService],
    });
    service = TestBed.inject(CartService);
  });

  it('aggiunge un prodotto al carrello', () => {
    service.add({ id: 'p1', price: 10 });
    expect(service.total()).toBe(10);
  });
});
```

- Usiamo TestBed solo per ottenere il servizio come farebbe Angular.
- Niente HTTP, nessun template: test molto rapido.

Setup di un servizio HTTP con HttpTestingController

```
let service: ProductService;
let http: HttpTestingController;

beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [
      ProductService,
      provideHttpClient(withInterceptors([])),
      provideHttpClientTesting(),
    ],
  });
  service = TestBed.inject(ProductService);
  http = TestBed.inject(HttpTestingController);
});
```

- `provideHttpClientTesting()` attiva la libreria di test HTTP,
- `HttpTestingController` intercetta le richieste invece di andare in rete.

Test di una chiamata HTTP

```
it('chiama /products', () => {
  service.list().subscribe();

  const req = http.expectOne('/api/products');
  expect(req.request.method).toBe('GET');

  req.flush([]);
  http.verify();
});
```

- `expectOne` verifica che esista una sola richiesta con quella URL,
- `flush` simula la risposta del server,
- `verify` assicura che non ci siano richieste “orfane”.

Test di un componente con Reactive Form

```
describe('CheckoutComponent', () => {
  it('rende valido il form con dati corretti', () => {
    const fixture = TestBed.createComponent(CheckoutComponent);
    const cmp = fixture.componentInstance;

    cmp.checkoutForm.setValue({ fullName: 'Mario', email: 'm@x.it' });

    expect(cmp.checkoutForm.valid).toBeTrue();
  });
});
```

- Usiamo `createComponent` per ottenere il component,
- manipoliamo il `FormGroup` direttamente,
- verifichiamo che le regole di validazione siano rispettate.

Strategia base per testare form

Per i form reattivi:

- testa **regole di validazione**:
  - form valido con dati buoni,
  - form invalido con combinazioni sbagliate.
- usa `setValue` / `patchValue` per impostare lo stato iniziale,
- verifica `errors` su controlli e gruppi:

```
expect(cmp.checkoutForm.controls.email.errors?.['email'])
  .toBeTrue();
```

- non serve testare ogni singolo binding del template:
  - concentrati su logica e regole di dominio.

Errori comuni nei test Angular

- Usare sempre TestBed anche per funzioni puramente logiche.
- Non verificare le richieste HTTP (mancato uso di HttpTestingController).
- Test fragili:
  - aspettarsi testi/HTML esatti troppo dettagliati,
  - dipendere da implementazioni interne anziché dal comportamento.
- Dimenticare di:
  - chiamare `http.verify()`
  - gestire async (`done`, `fakeAsync`, `waitForAsync`) quando necessario.

Buona pratica:

- partire da test piccoli e mirati + spostare logica “complicata” fuori dai componenti

Recap Modulo 7

Takeaway

- Non tutto richiede Angular: molte regole si testano come funzioni pure.
- TestBed è lo strumento base per servizi e componenti.
- HttpTestingController permette di testare `HttpClient` senza backend reale.
- I form reattivi sono facili da testare manipolando il `FormGroup` e leggendo `errors`.
- Evita test fragili: concentrati su comportamento e regole di dominio, non su dettagli estetici.

Contatti

Contatti docente

- 👤 Ciro Finizio
- 📧 ciro.finizio@unife.it
- 📧 c.finizio@cineca.it
- 💻 GitHub: https://github.com/nome-cognome

Giochi & sandbox per RxJS

- 🍊 **RxJS Fruits** – mini-gioco a livelli sugli operatori
  👉 https://www.rxjs-fruits.com/
- 🔮 **RxJS Marbles (v2)** – diagrammi interattivi degli operatori
  👉 https://rxjsmarbles.dev/
- 🧪 **RxTutor** – sandbox visuale per osservabili e operatori
  👉 https://rxtutor.org/
- 🧩 **ThinkRx – RxJS Playground** – marble playground per sperimentare pipeline
  👉 https://thinkrx.io/rxjs/