Sistemi Web — Laboratorio Angular

Lezione 4: Reactive Forms • Validazione • Checkout

Obiettivo: completare la pagina di **checkout** con:

- form reattivo tipizzato
- validazione lato client
- UX e **accessibilità** degli errori
- integrazione con **carrello** e ordine

Punto di partenza (progetto)

Assumiamo già presenti:

- Progetto Angular `shop` con routing configurato
- Catalogo prodotti funzionante (lista + dettaglio)
- `CartService` o equivalente con stato carrello in-memory
- Layout base con header e pulsante "Vai al checkout"

In questo lab aggiungiamo **solo**:

- componente `Checkout` + Reactive Form
- gestione errori + a11y
- simulazione creazione ordine da carrello

STEP 1 — Checkout page + FormGroup

Consegna

1. Crea componente pagina `CheckoutPageComponent` (standalone) in `app/features/checkout/checkout-page.ts` + `checkout-page.html`
2. Configura route `/checkout` che mostra la pagina
3. Definisci un **Reactive Form** con:
   - `customer`: `firstName`, `lastName`, `email`
   - `address`: `street`, `city`, `zip`
   - `shippingMethod`, `privacy`
4. Usa validator built-in:
   - `required`, `email`, `minLength`, `pattern` per `zip`
5. Disabilita il submit se `form.invalid`

Output: form visibile, con valori iniziali vuoti e pulsante submit disabilitato.

STEP 0 - Crea componente

```
ng generate service core/services/order-service --standalone
```

```
<!-- header.html -->
<button matIconButton (click)="goToCheckout()">
    <mat-icon>shopping_cart</mat-icon>
</button>
```

```
// heeader.ts
  goToCheckout(): void {
    this.router.navigate(['/checkout']);
  }
// app.routes.ts
{ path: 'checkout', component: CheckoutPage  },
```

STEP 1 — Struttura componente

```
// app/features/checkout/checkout-page.ts
...
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatCheckboxModule } from '@angular/material/checkbox';

@Component({
  selector: 'app-checkout-page',
  standalone: true,
  imports: [
    ReactiveFormsModule,
    MatFormFieldModule,
    MatInputModule,
    MatCheckboxModule
  ],
  templateUrl: './checkout-page.html'
})
export class CheckoutPageComponent {
  private fb = inject(FormBuilder);
}
```

STEP 1 — FormGroup

```
// dentro CheckoutPageComponent
readonly form = this.fb.group({
  customer: this.fb.group({
    firstName: ['', [Validators.required, Validators.minLength(2)]],
    lastName:  ['', [Validators.required, Validators.minLength(2)]],
    email:     ['', [Validators.required, Validators.email]]
  }),
  address: this.fb.group({
    street: ['', Validators.required],
    city:   ['', Validators.required],
    zip:    ['', [Validators.required, Validators.pattern(/^[0-9]{5}$/)]]
  }),
  shippingMethod: ['standard', Validators.required],
  privacy:        [false, Validators.requiredTrue]
});
```

STEP 1 — Binding form

```
<!-- app/features/checkout/checkout-page.html -->
<form class="checkout-form"
      [formGroup]="form"
      (ngSubmit)="onSubmit()">
  <section formGroupName="customer">
    <h2>Dati cliente</h2>
    <mat-form-field appearance="outline">
      <mat-label>Nome</mat-label>
      <input matInput formControlName="firstName" />
    </mat-form-field>

    <mat-form-field appearance="outline">
      <mat-label>Cognome</mat-label>
      <input matInput formControlName="lastName" />
    </mat-form-field>
  </section>
</form>
```

STEP 1 — Submit e validità

```
<section formGroupName="address">
  <h2>Indirizzo</h2>
  <mat-form-field appearance="outline">
    <mat-label>Via</mat-label>
    <input matInput formControlName="street" />
  </mat-form-field>

  <mat-form-field appearance="outline">
    <mat-label>CAP</mat-label>
    <input matInput formControlName="zip" />
  </mat-form-field>
</section>

<mat-checkbox formControlName="privacy">
  Accetto termini e privacy
</mat-checkbox>

<button mat-raised-button
        color="primary"
        type="submit"
        [disabled]="form.invalid">
  Conferma ordine
</button>
```

STEP 2 — Errori utente + Accessibilità

Consegna

1. Mostra errori **inline** per i campi principali:
   - `Nome`, `Cognome`, `Email`, `CAP`, `privacy`
2. Gli errori devono essere visibili solo dopo `touched` / `dirty`
3. Aggiungi un **“error summary”** in cima al form:
   - testo breve con count errori
   - area `aria-live="polite"` e `role="status"`
4. Al submit, se il form è invalido:
   - chiama `markAllAsTouched()`
   - porta il **focus** al primo campo non valido

Output: form che spiega chiaramente cosa c’è da correggere.

STEP 2 — Helper per errori

Crea metodi di utilità nel componente:

```
getControl(path: string) {
  return this.form.get(path);
}

hasError(path: string, errorCode: string): boolean {
  const control = this.getControl(path);
  return !!control && control.hasError(errorCode) && control.touched;
}
```

E nel template:

```
<mat-error *ngIf="hasError('customer.firstName','required')">
  Il nome è obbligatorio.
</mat-error>
```

STEP 2 — Submit + focus

```
// CheckoutPageComponent
showSummary = false;

onSubmit(): void {
  if (this.form.invalid) {
    this.form.markAllAsTouched();
    this.showSummary = true;
    this.focusFirstInvalid();
    return;
  }
  this.showSummary = false;
}

private focusFirstInvalid(): void {
  const firstInvalid = document.querySelector(
    'form .ng-invalid[formControlName]'
  ) as HTMLElement | null;
  firstInvalid?.focus();
}
```

STEP 2 — Error summary

```
<div class="error-summary"
     *ngIf="showSummary && form.invalid"
     role="status"
     aria-live="polite">
  <p>
    Il form contiene errori. Controlla i campi evidenziati
    prima di procedere.
  </p>
</div>
```

Singoli campi:

```
<mat-form-field appearance="outline">
    <mat-label>Email</mat-label>
    <input matInput formControlName="email" />
    @if (hasError('customer.email','required')) {
    <mat-error>
    L'email è obbligatoria.
    </mat-error>
    }
    @if (hasError('customer.email','email')) {
    <mat-error>
    Inserisci un indirizzo email valido.
    </mat-error>
    }
</mat-form-field>
```

STEP 3 — Carrello → Ordine (POST mock)

1. Creare il service `CartService`
2. Mostra nel template:
   - lista articoli in carrello (titolo, qty, prezzo)
   - **totale** dell'ordine
3. Impedisci il submit se il carrello è vuoto
4. Crea un modello `Order` in `app/core/models/order.ts` con:
   - `customer`, `address`, `items`, `total`, `createdAt`
5. Simula un `OrderService` con metodo `create(order: Order)` che:
   - restituisce un `Observable<Order>` mockato
   - gestisce `loading`, `success` e `error`

STEP 3 — Modello Order

Modello `Order`:

```
// app/core/models/order.ts
import { Product } from "./product";

export interface Order {
  customer: unknown;
  address: unknown;
  items: Product[];
  total: number;
  createdAt: string;
}
```

STEP 3 — Servizi e stream

```
ng generate service core/services/order
ng generate service core/services/cart
```

```
export class Cart {
  product$ = inject(ProductService);

  list() {
    return this.product$.list().pipe(
      // Prendi solo i primi 5 elementi come esempio
      map(products => products.slice(0, 5))
    );
  }
}
```

STEP 3 — Servizi e stream

```
// OrderService
export class Cart {
  product$ = inject(ProductService);

  list() {
    return this.product$.list().pipe(
      // Prendi solo i primi 5 elementi come esempio
      map(products => products.slice(0, 5))
    );
  }
}
```

STEP 3 — Servizi e stream

```
// dentro CheckoutPageComponent
import { CartService } from '../../core/services/cart';
import { OrderService } from '../../core/services/order';
import { map } from 'rxjs';

private cart = inject(CartService);
private orderService = inject(OrderService);

readonly items$ = this.cart.list();
readonly total$ = this.items$.pipe(
map(items => items.reduce(
    (sum, item) => sum + item.price, 0))
);

loading = false;
orderSuccess = false;
orderError = false;
```

STEP 3 — Soluzione (HTML) · Riepilogo carrello

```
<section class="cart-summary">
  <h2>Riepilogo carrello</h2>

  @if ((items$ | async)?.length === 0) {
    <p>Il carrello è vuoto. Aggiungi prodotti prima di procedere.</p>
  } @else {
    <ul>
      @for (item of items$ | async; track item.id) {
        <li>
          {{ item.title }} — {{ item.price | currency:'EUR' }}
        </li>
      }
    </ul>

    <p class="total">
      Totale: <strong>{{ total$ | async | currency:'EUR' }}</strong>
    </p>
  }
</section>
```

E ricorda di disabilitare il submit se il carrello è vuoto (es. `[disabled]="form.invalid || !(items$ | async)?.length"`).

STEP 3 — Invio ordine

```
onSubmit(): void {
  if (this.form.invalid) {
    this.form.markAllAsTouched();
    this.showSummary = true;
    this.focusFirstInvalid();
    return;
  }

  this.loading = true;
  this.orderSuccess = false;
  this.orderError = false;

  const value = this.form.getRawValue();
  this.items$.pipe(take(1)).subscribe(items => {
    const order: Order = {
      customer: value.customer!,
      address:  value.address!,
      items,
      total: items.reduce(
        (sum, it) => sum + it.price * it.quantity, 0),
      createdAt: new Date().toISOString()
    };

    this.orderService.create(order).subscribe({
      next: () => {
        this.loading = false;
        this.orderSuccess = true;
        this.cart.clear();
        this.form.reset();
      },
      error: () => {
        this.loading = false;
        this.orderError = true;
      }
    });
  });
}
```

STEP 4 — Guard su Checkout

1. Crea un servizio `AuthService` in `app/core/services/auth.service.ts` che:
   - tenga in memoria se l'utente è loggato (`isLoggedIn: boolean`)
   - esponga metodi `login()` e `logout()` per cambiare stato
2. Crea una **guard** `checkoutGuard` che:
   - permette l'accesso a `/checkout` solo se l'utente è loggato
   - in caso contrario, reindirizza a `/login` (o alla home `/`)
3. Aggancia la guard alla route del checkout
4. Aggiungi nell'header un bottone **Login/Logout** che usa `AuthService`

Output: se non sei loggato, non puoi aprire la pagina di checkout.

STEP 4 — AuthService finto (in-memory)

```
 ng generate service core/services/auth-service
```

```
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class AuthService {
  isLoggedIn = false;

  login(): void { this.isLoggedIn = true; }
  logout(): void { this.isLoggedIn = false; }
}
```

STEP 4 — Guard su checkout

```
ng generate guard core/guard/checkout-guard
```

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../services/auth-service';

export const checkoutGuard: CanActivateFn = () => {
  const auth = inject(AuthService);
  const router = inject(Router);
  return auth.isLoggedIn
    ? true
    : router.createUrlTree(['/login']);
};
```

Logica:

- se `isLoggedIn === true` → `true` (navigazione consentita)
- altrimenti → `UrlTree` verso `/login` (redirect automatico)

STEP 4 — Soluzione (TS/HTML) · Route + Header

Aggancia la guard alla route del checkout (es. in `app.routes.ts`):

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

Esempio di toggle login nell’header (shell):

```
constructor(public auth: AuthService) {}

toggleLogin(): void {
  this.auth.isLoggedIn ? this.auth.logout() : this.auth.login();
}
```

```
<button mat-button (click)="toggleLogin()">
  {{ auth.isLoggedIn ? 'Logout' : 'Login' }}
</button>
```

STEP 4 — Ottimizzazioni di produzione

Obiettivo

Preparare l'app per la **produzione**, toccando solo i punti essenziali:

- build ottimizzata (`ng build --configuration production`)
- controllo dimensioni bundle (budgets)
- **Dockerfile minimale** per servire lo shop come SPA statica
- comandi di build & run per verificare il deploy locale

STEP 4 — Build di produzione

Cosa fare

1. Aggiungi in `package.json` uno script dedicato alla build di produzione:
   - `"build:prod": "ng build --configuration production"`
2. Lancia la build di produzione:
   - `npm run build:prod`
3. Verifica l'output nella cartella `dist/`:
   - controlla che esista `dist/shop` (o il nome del tuo progetto)
4. Controlla i **budgets** in `angular.json`:
   - individua la sezione `budgets` della configurazione `production`
   - riduci il budget di `initial` a un valore realistico (es. ~500 kB) per notare gli warning/error

STEP 4 — Script e configurazione build

Esempio di `scripts` in **package.json**:

```
{
  "scripts": {
    "start": "ng serve",
    "test": "ng test",
    "build": "ng build",
    "build:prod": "ng build --configuration production"
  }
}
```

STEP 4 — Script e configurazione build

Estratto semplificato da `angular.json`:

```
"configurations": {
  "production": {
    "optimization": true,
    "sourceMap": false,
    "outputHashing": "all",
    "budgets": [
      {
        "type": "initial",
        "maximumWarning": "500kb",
        "maximumError": "1mb"
      }
    ]
  }
}
```

STEP 4 — Docker minimale

1. Crea un file `Dockerfile` nella root del progetto Angular.
2. Implementa una build **multi-stage**:
   - **stage 1**: usa `node` per installare dipendenze e fare la build prod
   - **stage 2**: usa `nginx` per servire i file statici della `dist/`
3. Costruisci l'immagine Docker:
   - `docker build -t shop-web .`
4. Avvia il container:
   - `docker run -p 8080:80 shop-web`
5. Apri `http://localhost:8080` e verifica che la SPA funzioni.

STEP 4 — Dockerfile multi-stage minimale

```
# Stage 1: build Angular app
FROM node:20-alpine AS build
WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build:prod

# Stage 2: serve con Nginx
FROM nginx:alpine

# sostituisci "shop" con il nome del tuo progetto se diverso
COPY --from=build /app/dist/shop/browser /usr/share/nginx/html

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

STEP 4 — Comandi Docker da eseguire

1. Build dell’immagine

```
# dalla root del progetto Angular
docker build -t shop-web .
```

2. Avvio del container

```
# esegui l’immagine esponendo la porta 80 di nginx sulla 8080 locale
docker run --rm -p 8080:80 --name shop-web shop-web
```

Poi apri il browser su:

```
http://localhost:8080
```

STEP 4 — Comandi utili (logs, stop, cleanup)

3. Controllare lo stato e i log

```
# lista dei container attivi
docker ps

# log del container in tempo reale
docker logs -f shop-web
```

4. Fermare il container e ripulire

```
# stop del container
docker stop shop-web

# elenco immagini
docker images

# rimozione esplicita dell’immagine (se serve)
docker rmi shop-web
```

STEP 1 — Test HttpClient con HttpTestingController

Vogliamo scrivere **un test unitario** per `ProductService` che usa `HttpClient`:

1. Crea/controlla il servizio `ProductService` in `app/core/services/product.service.ts` con metodo:
   - `list(): Observable<Product[]>` che fa una `GET` su `/api/products`
2. Crea il file di test: `app/core/services/product.service.spec.ts`
3. Nel test:
   - configura il `TestBed` con `HttpClientTestingModule`
   - ottieni `ProductService` e `HttpTestingController`
   - testa che `list()`:
     - faccia una request `GET` a `/api/products`
     - ritorni i dati mock inviati con `req.flush(mockProducts)`
4. Verifica che **non restino request pendenti** con `httpMock.verify()`.

STEP 1 — Aiuto: setup del TestBed

Suggerimento per il setup nel file `product.service.spec.ts`:

```
import { TestBed } from '@angular/core/testing';
import {
  HttpClientTestingModule,
  HttpTestingController
} from '@angular/common/http/testing';
import { ProductService } from './product.service';

describe('ProductService', () => {
  let service: ProductService;
  let httpMock: HttpTestingController;
```

Nella `beforeEach`:

```
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule]
    });
    service = TestBed.inject(ProductService);
    httpMock = TestBed.inject(HttpTestingController);
  });
```

STEP 1 —Test della GET /api/products

Nel file `product.service.spec.ts` completiamo il test:

```
  afterEach(() => {
    httpMock.verify();
  });

  it('dovrebbe fare una GET su /api/products e restituire i prodotti', () => {
    const mockProducts = [
      { id: 'p1', title: 'Prodotto 1', price: 10 } as any,
      { id: 'p2', title: 'Prodotto 2', price: 20 } as any
    ];

    service.list().subscribe(products => {
      expect(products).toEqual(mockProducts);
    });

    const req = httpMock.expectOne('/api/products');

    expect(req.request.method).toBe('GET');

    req.flush(mockProducts);
  });
});
```

Contatti

Contatti docente

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