Sistemi Web — Angular

Lezione 1: Teoria

Fondamenti Web • Angular Core • TypeScript • CLI & Anatomia progetto

Roadmap del modulo (5 lezioni)

L1: Fondamenti Web · Angular core · TypeScript · CLI

L2: Direttive · Pipes · Services/DI · RxJS base

L3: Router · HttpClient · Interceptor · Guards

L4: Reactive Forms · Validazione · Accessibilità

L5: Gestione dello stato Stato · Performance · Test · Build

Agenda - Lezione 1

Teoria

- Fondamenti del Web
- Angular core
- TypeScript essenziale
- CLI & anatomia progetto

Pratica

- Esercitazione typescript
- Esercitazione Angular Core

Obiettivi di apprendimento

- Architettura **client–server** e **HTTP**
- Concetti core di **Angular** (componenti, binding, DI)
- **TypeScript** per codice più sicuro e manutenibile
- Organizzazione progetto con **Angular CLI**

Fondamenti del Web

Cos’è il Web

- Risorse identificate da **URL** scambiate via **HTTP/HTTPS**.
- **Client (browser)** ↔️ **Server**; **DNS** risolve nomi in IP.
- Documenti, API JSON, asset statici; modello **richiesta/risposta**.

Struttura di un URL

`schema://host:porta/percorso?query#frammento`

- Schema (es. https), host e porta, percorso, query string.
- Il **frammento** (#) è gestito dal client (es. SPA, ancore).

```
# Esempi di URL
https://example.com:443/users/42?expand=posts#details
https://api.example.com/v1/tasks?boardId=1&limit=20
```

HTTP: metodi e semantica

- **GET** (lettura, idempotente) · **POST** (creazione)
- **PUT** (sostituzione, idempotente) · **PATCH** (aggiornamento parziale)
- **DELETE** (rimozione, idempotente)

```
# CRUD basilare con curl
curl -X GET  http://localhost:3001/products?_page=1
curl -X POST http://localhost:3001/orders -H "Content-Type: application/json" -d '{"total": 99.9}'
```

HTTP: status code essenziali

- **2xx:** successo (200 OK, 201 Created)
- **3xx:** redirezioni (304 Not Modified)
- **4xx:** errori client (400, 401, 403, 404)
- **5xx:** errori server (500, 503)

Header & content negotiation

```
# Richiesta
GET /api/tasks HTTP/1.1
Host: localhost:3001
Accept: application/json
Authorization: Bearer <token>

# Risposta
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "abc123"
Cache-Control: no-store
```

Idempotenza & retry sicuri

- Idempotenza: N esecuzioni == 1 esecuzione (**GET/PUT/DELETE**)
- Abilita retry e caching controllati
- **POST/PATCH** in generale **non** è idempotente

Cookie, sessione e Web Storage

- Cookie: **Secure**, **HttpOnly**, **SameSite**
- `localStorage` / `sessionStorage`: **non** per segreti

```
// localStorage: evitare segreti
localStorage.setItem("theme", "dark");
const theme = localStorage.getItem("theme");

// Cookie HttpOnly non è leggibile via JS
```

Same-Origin Policy & CORS

- Same-Origin Policy limita accesso tra origini diverse
- CORS abilita eccezioni via header lato server

```
// Fetch con credenziali (se CORS lo consente)
fetch("http://localhost:3001/tasks", { credentials: "include" })
  .then(r => r.json())
  .then(console.log);
```

REST — Introduzione

- **REST** è uno stile architetturale per API incentrate su **risorse** identificate da URL e manipolate tramite **metodi HTTP**.
- Le interazioni sono **stateless**: ogni richiesta contiene tutto il contesto; il server non mantiene stato di sessione applicativa.
- Le risorse hanno **rappresentazioni** (es. JSON) scambiate con header **Content-Type** e **status code** standard.
- **Interfaccia uniforme**: convenzioni coerenti su URL, metodi e codici; favorisce **interoperabilità** e **cache** lungo la catena.
- **Vantaggi principali**: semplicità (riusa HTTP), **scalabilità** (stateless + caching), **evolvibilità** (versioni nelle rappresentazioni/URL) e **indipendenza** client–server.

Esempio risorsa: Product — path REST

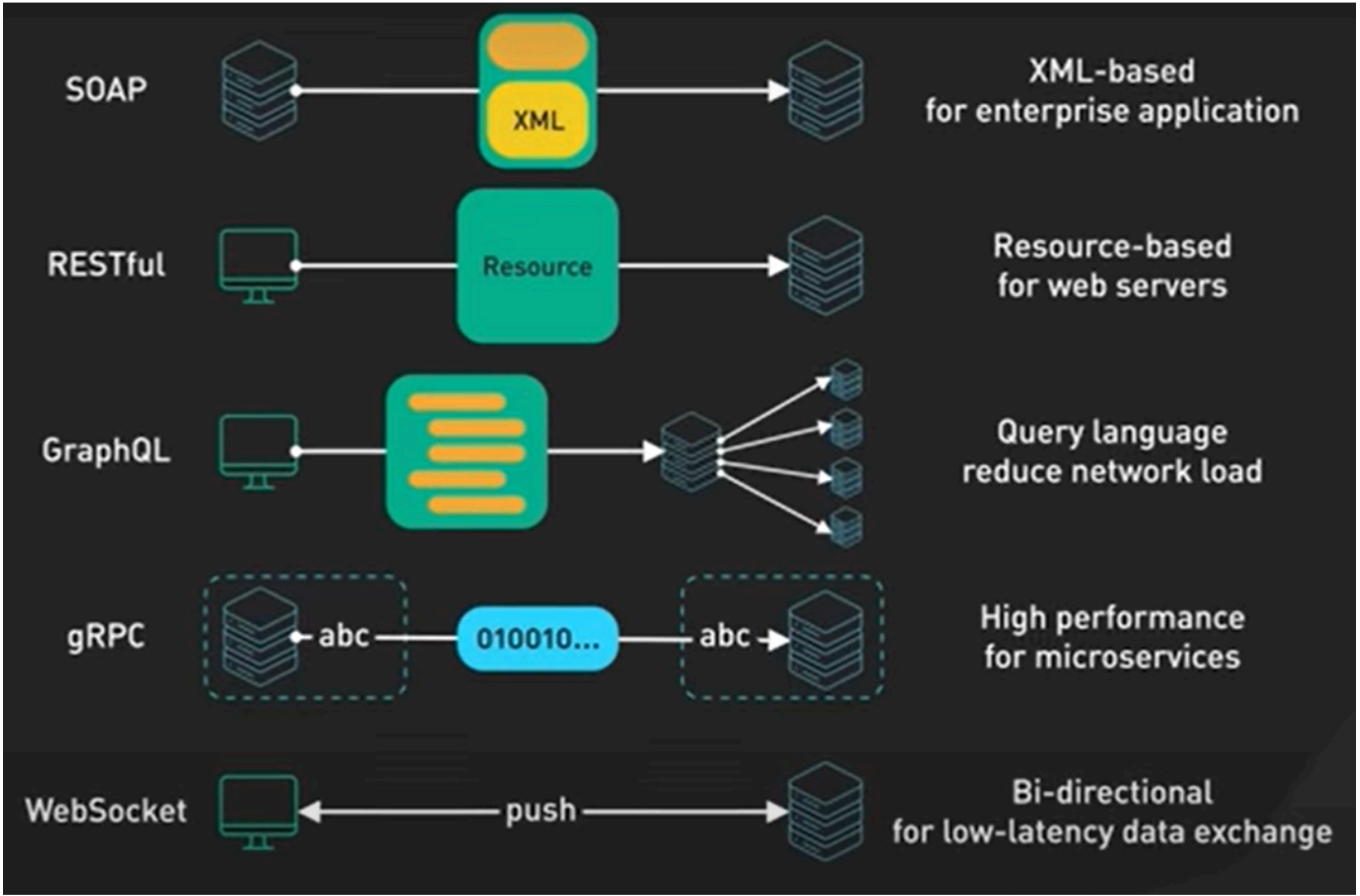
- Risorsa **Product** con identificatore `:id`.
- Path collezione vs. risorsa singola.
- Query params tipici per lista: `page`, `limit`, `q`, `categoryId`, `sort`, `order`.

```
GET    /products?page=1&limit=12&categoryId=c1&q=runner&sort=price&order=asc
GET    /products/:id
POST   /products
PUT    /products/:id
PATCH  /products/:id
DELETE /products/:id
```

JSON & API REST

`GET http://localhost:3001/task/t1`

```
{
  "id": "t1",
  "title": "Refactor servizio API",
  "createdAt": "2025-09-01T09:00:00Z",
  "tags": ["tech-debt","prio:high"]
}
```


SOAP
XML
XML-based
for enterprise application
RESTful
Resource
Resource-based
for web servers
GraphQL
Query language
reduce network load
gRPC
abc
010010...
abc
High performance
for microservices
WebSocket
push
Bi-directional
for low-latency data exchange

REST API

REST REQUEST

GET https://sample.com/person 1

REST JSON

```
{
    “firstName” : “John”,
    “middleName”: “Andrew”
    “lastName”:  “Smith”,
    “email”: “jas1992@gmail.com”,
}
```

GraphQL

GRAPHQL QUERY

```
{
person  {
        firstName
        lastName
    }
}
```

GRAPHQL QUERY

```
{
   “data”: {
       “person”: {
           “firstName”: “John”,
           “lastName”: “Smith”,
       }
    }
}
```

Ciclo di una richiesta

1. Costruzione URL/parametri → Invio richiesta
2. Ricezione status+header → Parsing corpo
3. Gestione errori → Aggiornamento UI

```
// fetch con gestione errori
async function loadTasks() {
  const res = await fetch("http://localhost:3001/tasks?boardId=1");
  if (!res.ok) throw new Error("HTTP " + res.status);
  return res.json();
}
loadTasks().then(render).catch(showError);
```

Single Page Application

Introduzione

Single Page Application (SPA)

- Una **Single Page Application** è un'applicazione web che carica **una sola pagina HTML** e aggiorna dinamicamente i contenuti tramite **JavaScript** e **API HTTP**, senza ricaricare l'intera pagina dal server.
- Le SPA gestiscono la **navigazione lato client** (via `Router`) e mantengono uno stato applicativo nel browser.
- Il server fornisce solo i dati (API REST/JSON), mentre la logica di presentazione risiede nel client.

```
flowchart LR
  A[Browser] -->|Richiede index.html| B[Server]
  A -- REST API --> B
  B -->|JSON| A
  A -->|Aggiorna solo il contenuto| A
```

SPA vs MPA — Confronto

Aspetto	SPA	MPA (Multi-Page Application)
Struttura	Una sola pagina HTML	Ogni pagina è servita dal server
Navigazione	Gestita dal **client router**	Ogni link → nuova richiesta HTTP
Performance	Prima richiesta più lenta, poi navigazione fluida	Prima pagina rapida, ma reload completo ad ogni cambio
Carico server	Minore (dati JSON)	Maggiore (rendering lato server)
SEO	Richiede SSR o prerender (es. Angular Universal)	Indicizzabile nativamente
Quando usarla	App interattive, dashboard, e-commerce, gestionali	Siti informativi, blog, documentazione

Single Page Application

No page refresh on request

Traditional Web Application

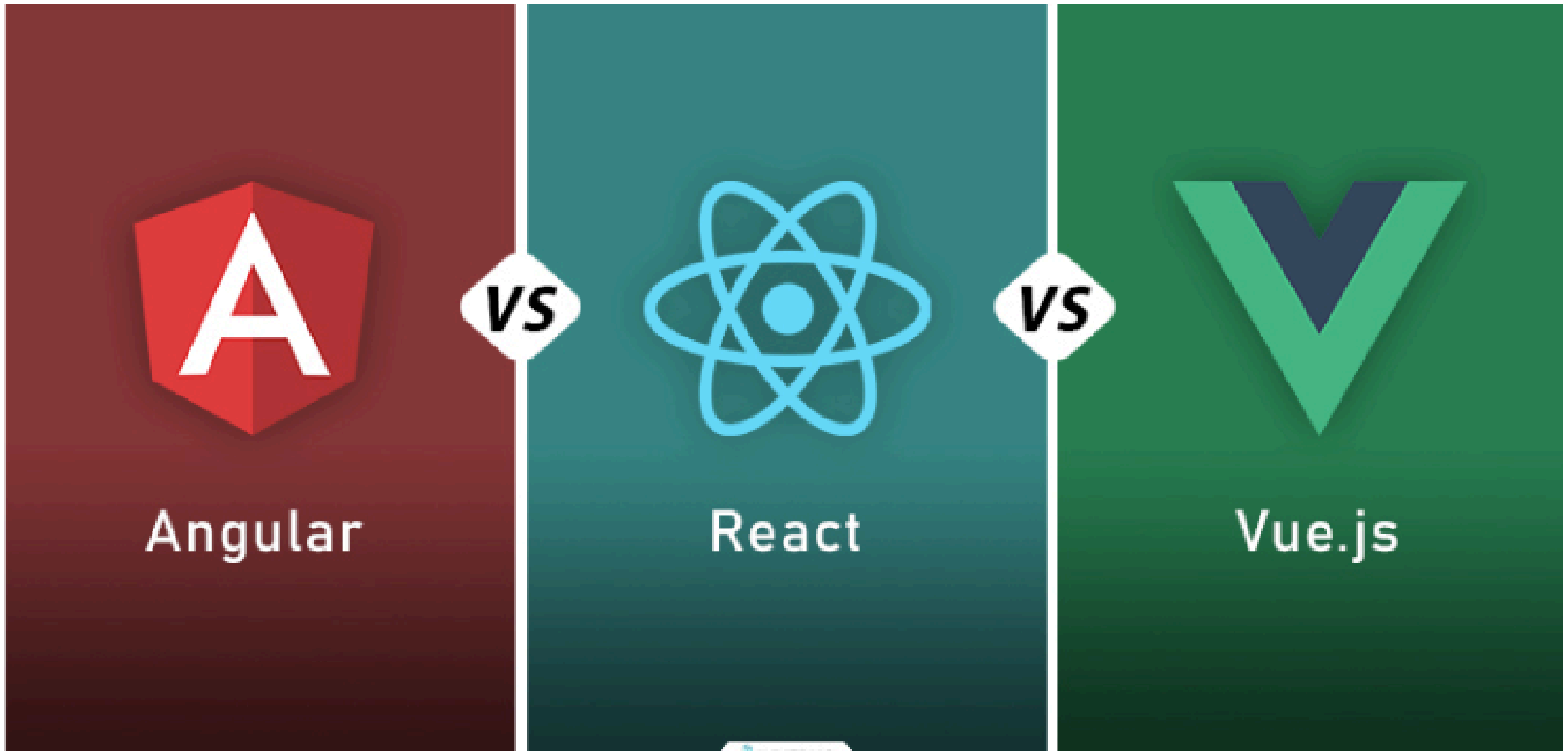
Whole page refresh on request

Angular core

Perché Angular per SPA

- Framework completo: componenti, DI, routing, form, HttpClient
- Tooling solido, stile coerente → adatto a team

Alternative



Angular vs React vs Vue

Criterio	Angular	React	Vue
Natura	Framework completo	Libreria UI minimale	Framework progressivo
Template & TS	Template HTML + **TypeScript-first**	**JSX** (TS opzionale)	**SFC** (`<template>` `<script>`) con ottimo TS
Dipendenze incluse	Router, **DI**, Forms, HttpClient	Scegli librerie (Router, state, form)	Router ufficiale + **Pinia** + ecosistema curato

Angular vs React vs Vue

Criterio	Angular	React	Vue
Reattività/rendering	Change Detection + **Signals**	Virtual DOM + **Hooks**	Sistema reattivo a **Proxy** + `computed`
Tooling & SSR	**CLI**, `ng update`, **SSR/hydration**	Vite + **Next.js**	Vite + **Nuxt**
Curva di apprendimento	Più ripida	Media, molto flessibile	incrementale
Quando sceglierlo	Enterprise/team grandi, standard condivisi	Prototipazione rapida, ecosistema vasto	Adozione graduale, app piccole/medie

Perché Angular

- Stack integrato end-to-end (Router, HttpClient, Forms, i18n, test) che riduce le scelte e mantiene il codice coerente tra team.
- Architettura + DI e TypeScript → manutenibilità, testabilità e refactor sicuri su basi condivise.
- Tooling maturo (Angular CLI, schematics, ng update) con build ottimizzate e LTS prevedibile per upgrade senza **(troppi)** problemi.
- Performance by default: Change Detection OnPush, Signals, lazy loadComponent, prefetch, SSR, supporto a @defer.
- Developer & User Experience: template dichiarativi, pipe/direttive, Reactive Forms, a11y e i18n gia integrati.
- Scalabilità enterprise: monorepo-friendly, convenzioni e linting, strutture per feature.

Architettura a componenti

- Vista (template), Logica (classe), Stili (scoped)
- Riuso, testabilità, separazione responsabilità

Componente

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

@Component({
  selector: 'app-task-card',
  standalone: true,
  template: `<article class="card">
      <h3>{{ title }}</h3>
      <button (click)="onEdit()">Modifica</button>
    </article>`
})
export class TaskCardComponent {
  title = 'Titolo task';
  onEdit() { /* ... */ }
}
```

Cos’è un componente Angular

Un **componente** è l’unità base di un’app Angular:

- Incapsula **logica**, **template HTML** e **stili CSS**.
- Rappresenta un **blocco riutilizzabile** dell’interfaccia utente.
- È definito da una **classe TypeScript** decorata con `@Component`.
- Il **selector** lo collega al DOM: ogni tag corrisponde a un’istanza del componente.

```
@Component({
  selector: 'app-hello',
  template: `<h3>Ciao {{ name }}!</h3>`,
  styles: [`h3 { color: #2563eb; }`]
})
export class HelloComponent {
  name = 'Angular';
}
```

Componenti Parent e Child

I componenti si organizzano in una **gerarchia**:

- Il **Parent** passa dati al **Child** tramite `@Input()`.
- Il **Child** emette eventi al **Parent** con `@Output()`.
- Il flusso è **unidirezionale**: dati scendono, eventi risalgono.

Componenti Parent e Child

```
// Child: product-card.component.ts
@Component({
  selector: 'app-product-card',
  template: `<h3>{{product.title}}</h3>
             <button (click)="add.emit(product)">Aggiungi</button>`
})
export class ProductCardComponent {
  @Input() product!: Product;
  @Output() add = new EventEmitter<Product>();
}

// Parent: product-list.component.html
<app-product-card
  *ngFor="let p of products"
  [product]="p"
  (add)="onAdd($event)">
</app-product-card>
```

Gerarchia dei componenti

Un’app Angular è un **albero di componenti**:

il **Root component** (`AppComponent`) contiene figli, che a loro volta possono avere altri figli.

```
A["🟦 AppComponent (Root)"]
|  B["🟩 HeaderComponent"]
|  C["🟨 ProductListComponent"]
|  |  D["🟧 ProductCardComponent"]
|  E["🟪 FooterComponent"]

A --> B
A --> C
A --> E
C --> D
```

Gerarchia dei componenti

- Ogni componente è **incapsulato** (template, logica, stile).
- Il **flusso dati** è gerarchico:
  - `@Input()` → dal parent al child
  - `@Output()` → dal child al parent
- Angular gestisce il **Change Detection Tree** per aggiornare solo i rami coinvolti.

Componenti Standalone (v15+)

- **Standalone component** = non serve più dichiararli in un `NgModule`.
- Si dichiara direttamente `standalone: true` nel decoratore.
- Gli altri componenti, direttive e pipe da usare nel template vanno aggiunti a `imports: []`.
- Favorisce **modularità** e **lazy loading** più semplici.

Componenti Standalone (v15+)

```
@Component({
  selector: 'app-product-card',
  standalone: true,
  imports: [CurrencyPipe],
  template: `<p>{{ price | currency:'EUR' }}</p>`
})
export class ProductCardComponent {
  @Input() price!: number;
}
```

Vantaggi

- Meno boilerplate (nessun `NgModule` intermedio).
- Migliore tree-shaking e caricamento rapido.
- Struttura più chiara per feature e routing dinamico.

Componenti tradizionali vs Standalone

Aspetto	Tradizionali (NgModule)	Standalone (moderno)
Dichiarazione	Dentro `@NgModule.declarations`	Nel decoratore → `standalone: true`
Import	Moduli interi (`CommonModule`, `SharedModule`)	Componenti, pipe o direttive specifiche in `imports: []`
Utilizzo	Richiede che il modulo sia importato altrove	Può essere caricato e usato direttamente
Lazy Loading	Solo tramite moduli	Diretto con `loadComponent()`
Manutenibilità	Più verboso	Più semplice e modulare
Default da	Angular ≤14	Angular ≥17

Ciclo di vita dei componenti

- Hook principali: `ngOnChanges` (reagisci a nuovi `@Input`), `ngOnInit` (init/subscribe), `ngAfterViewInit` (accesso a ViewChild), `ngOnDestroy` (cleanup).
- **Regola d'oro**: niente side-effects nel `constructor`; usa gli hook. Preferisci `async` pipe ai subscribe manuali.
- Gestisci risorse con **unsubscribe sicuro** (es. `takeUntilDestroyed`)
- Usa `ngOnChanges(ch: SimpleChanges)` per differenze tra vecchio/nuovo input.

```
@Component({ selector:'app-card', standalone:true, template:`…` })
export class CardComponent implements OnInit, OnChanges, OnDestroy {
  @Input() product!: Product;
  ngOnInit(){ /* inizializza view-model, avvia fetch leggeri */ }
  ngOnChanges(ch: SimpleChanges){ if (ch['product']) { /* sync UI */ } }
  ngOnDestroy(){ /* cleanup: timer, listener, controller.abort() */ }
}
```

Change Detection — Default vs OnPush

- **Default**: controlla l'albero a ogni evento/async; semplice ma può essere costoso.
- **OnPush**: aggiorna il componente quando cambia **la reference degli** `@Input`, su **eventi** interni o emissioni via `async` **pipe**. Favorisce **immutabilità**.
- Trigger comuni: eventi DOM, `setTimeout`, promise/HTTP completate, emissioni Observable.
- Linee guida: modella dati **immutabili**, usa `async` pipe, preferisci `trackBy` in liste; se serve forza, `markForCheck()`.

Change Detection — Esempio

```
@Component({
  selector:'app-list', standalone:true,
  // Setta la strategia per la gestione del cambiamento
  changeDetection: ChangeDetectionStrategy.OnPush,
  template:`<ul *ngIf="items$ as items">
              <li *ngFor="let i of items">{{ i.title }}</li>
            </ul>`
})
export class ListComponent {
  items = this.api.list();
  constructor(private api: ProductApi) {}
}
```

Data binding — tipi & quando usarli

- **Interpolazione** `{{ expr }}` → testo **read-only** nel template (titoli, etichette). Semplice e sicura.
- **Property binding** `[prop]="expr"` → valori dinamici su **DOM/Inputs** (`[src]`, `[disabled]`, `[value]`). Default per passare dati ai componenti figli.
- **Attribute/Class/Style** `[attr.aria-*]`, `[class.active]`, `[style.width.px]` → a11y, classi/inline style o **attributi non standard**. Usare quando la proprietà DOM non esiste.
- **Event binding** `(event)="handler($event)"` → gestire **eventi DOM/Output** (`(click)`, `(valueChange)`). Flusso dati **dal figlio al genitore**.
- **Two-way binding** `[(ngModel)]="model"` → form **semplici** o prototipi. In app strutturate preferire **Reactive Forms**.

Data binding: tre forme

```
<!-- Interpolazione -->
<h3>{{ product.title }}</h3>

<!-- Property binding -->
<img [src]="product.thumbnail" [alt]="product.title">

<!-- Attribute/Class/Style -->
<button [disabled]="isSaving" (click)="save()">Salva</button>

<!--Two-way binding -->
<label>Q.tà <input [(ngModel)]="qty"></label>

<!-- Event -->
<app-rating [value]="product.rating" (valueChange)="onRate($event)"></app-rating>
```

Direttive

- Una **direttiva** è una classe che aggiunge **comportamento** o **struttura** al DOM.
- **Strutturali**: modificano il layout aggiungendo/rimuovendo nodi (es. `*ngIf`, `*ngFor`, `*ngSwitch` — moderni: `@if`, `@for`).
- **Attributo**: cambiano aspetto/comportamento di un elemento (es. `[ngClass]`, `[ngStyle]`, `ngModel`, direttive custom).
- **Uso tipico**: show/hide, iterazioni, gestione classi/stili, a11y con `aria-*`, riuso di logica nel template.
- **Buone pratiche**: template leggeri, funzioni pure, usare `trackBy` / `track` nelle liste, direttive piccole e focalizzate.

Direttive — esempio

```
export interface Product { id: string; title: string; price: number; }
products: Product[] = [
  { id: 'p1', title: 'Runner Pro', price: 89.9 },
  { id: 'p2', title: 'Cappellino',  price: 19.9 }
];
showSaleOnly = false;
toggleSaleOnly(){ this.showSaleOnly = !this.showSaleOnly; }
get visible(): Product[] {
  return this.showSaleOnly
    ? this.products.filter(p=>p.price<20)
    : this.products; }
```

Direttive — esempio

```
<button (click)="toggleSaleOnly()">
  {{ showSaleOnly ? 'Mostra tutti' : 'Solo sconti' }}
</button>
<ul>
  <li *ngFor="let p of visible" [class.sale]="p.price < 20">
    {{ p.title }} — {{ p.price | currency:'EUR' }}
  </li>
</ul>
<p *ngIf="!visible.length">Nessun prodotto</p>
```

Direttive - Esempio track / trackBy

```
<!-- Opzione moderna -->
@for (p of products; track p.id) { <app-item [p]="p"></app-item> }

<!-- Opzione *ngFor (legacy & diffusa) -->
<li *ngFor="let p of products; trackBy: trackById">{{ p.title }}</li>
```

```
trackById(_i: number, p: Product){ return p.id; }
```

Pipes

- Una **pipe** trasforma un valore **solo per la vista** (template), es. formattare date, numeri, stringhe.
- **Pure (default)**: ricalcolano solo se cambiano input/parametri → migliori performance.
- **Impure** (`pure: false`): ricalcolano ad ogni change detection → evitare salvo casi particolari.
- **Parametri**: `{{ price | currency:'EUR':'symbol':'1.2-2' }}` ; **concatenzazione**: `{{ name | trim | titlecase }}` .
- **i18n**: formattazione data/numero/currency dipende dalla **locale** configurata dell'app.
- **Linee guida**: niente logica pesante/side-effects; per calcoli costosi, elaborazione nel componente.

Pipe — esempi

```
<p>Prezzo: {{ 89.9 | currency:'EUR' }}</p>
<p>Data ordine: {{ order.createdAt | date:'dd/MM/yyyy' }}</p>
<p>Sconto: {{ 0.15 | percent:'1.0-0' }}</p>
<p>Media: {{ 4.375 | number:'1.1-1' }}</p>
<p>Titolo: {{ 'runner pro' | titlecase }}</p>
<ul>
  <li *ngFor="let kv of product | keyvalue">{{ kv.key }}: {{ kv.value }}</li>
</ul>
```

Gestione degli errori — principi & livelli

- **A più livelli**: UI (messaggi/UX), servizi (mapping errori), **HttpClient/Interceptor**, e **RxJS** (retry/catch).
- **Messaggi chiari & a11y**: usa `role="alert"` / `aria-live` e azioni **Riprova/Contatta**; evita jargon tecnico.
- **Distinzione**: errori **di rete** (offline/time-out), **HTTP** (4xx/5xx), **validazione** form, **logica** (invarianti).
- **Strategie RxJS**: `catchError` per fallback, `retry` / `retryWhen` con backoff per **errori transienti**; re-throw per casi non recuperabili.
- **Interceptor**: gestione centralizzata (es. 401 → logout/refresh, 404 → notifica, 5xx → fallback), sempre **restituire** o **rilanciare** l'errore.
- **Osservabilità**: log strutturati (user/session/correlation id) e tracciamento lato server; niente PII nei log client.

Errori HTTP — esempio pratico (Service + UI)

```ts
// service: mapping errori
@Injectable({ providedIn: 'root' })
export class ProductApi {
  constructor(private http: HttpClient) {}
  list(): Observable<Product[]> {
    return this.http.get<Product[]>('/api/products').pipe(
      retry(1), // solo su errori transienti
      catchError((err: HttpErrorResponse) =>
        throwError(() => new Error(err.status === 0 ? 'offline' : 'api'))
      )
    );
  }
}
```

```html
<!-- UI: messaggio accessibile + retry -->
<section *ngIf="products$ | async as products; else error">
  <ul><li *ngFor="let p of products">{{ p.title }}</li></ul>
</section>
<ng-template #error>
```

CLI & anatomia progetto

Angular CLI: comandi base

```
# Generazione e sviluppo
ng new flowboard --standalone --routing --style=scss
cd flowboard
ng serve -o
ng generate component features/task-list
ng generate service core/task-api
```

Struttura del progetto (moderna)

```
# Albero (semplificato)
src/
  app/
    app.config.ts
    app.routes.ts
    core/        # servizi, guard, interceptor
    shared/      # componenti/pipe riusabili
    features/    # aree funzionali (catalog, product)
  main.ts
```

Provider & routing

```
// app.routes.ts
export const routes: Routes = [
  { path: '', redirectTo: 'boards', pathMatch: 'full' },
  { path: 'boards', loadComponent: () => import('./features/board/board.page') },
  { path: '**', loadComponent: () => import('./shared/not-found.page') }
];

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

Ambienti & configurazioni

```
// environment.ts (dev)
export const environment = {
  production: false,
  apiBaseUrl: 'http://localhost:3001'
};
```

Stili & accessibilità

```
<!-- Pulsante accessibile -->
<button aria-label="Aggiungi task" (click)="add()">+</button>
<!-- Focus visibile e contrasto adeguato nei CSS -->
```

File di configurazione (panoramica)

- **package.json**: dipendenze (Angular, RxJS, test, build) e script.
- **tsconfig.json**: opzioni TypeScript (decorators, sourceMap, strict, ecc.).
- **angular.json**: personalizzazioni build/serve/test della CLI.

Build

```
# Build di produzione
ng build --configuration production
# Output in dist/, pronto per il deploy
```

Questo crea la cartella `dist` contenente tutto il codice in javascript pronto ad essere servito come file statito da un application server (Es. `ngnix` , `apache`)

Recap

1. **HTTP/URL** sono il fondamento delle SPA
2. Un **componente Angular** = classe + template (+ stili)
3. Flusso dati unidirezionale (Input/Output)
4. **HTTP/URL** sono il fondamento delle SPA
5. **TypeScript** migliora qualità e produttività.
6. **CLI** struttura e automatizza build/serve/generate.