Angular

- Requisiti
  - Web: HTML, CSS, JavaScript + TypeScript; Node.JS, uso di npm
- Argomenti principali
  - Applicazione (AppComponent), Modulo (@NgModule)
  - Componenti, Model (relativi a Component), Direttive
  - Template Driven Form, Reactive Form
  - Pipe standard e custom
  - Servizi
  - Routing
  - HTTP via Observable
  - Unit test via Karma / Jasmine
- Repository di riferimento
  - https://github.com/egalli64/ngi

Angular

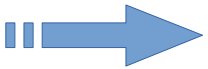
- Framework per lo sviluppo di webapp basato su NodeJS e TypeScript
  - https://angular.io/ (Google 2016)
- Evoluzione di AngularJS, sviluppato da Miško Hevery (2010)
  - Definizione di elementi HTML custom
- Installazione via npm
  - Angular CLI (Command Line Interface): npm install -g @angular/cli
    - NPM è parte di Node.js: https://nodejs.org/en/download/
    - Per rimuovere un package via npm, si usa il comando uninstall
  - Verifica della versione dell'Angular CLI installato: ng --version (10)

Workspace e starter app

- Angular CLI è basato su Webpack, semplifica il lavoro con Angular
- Dalla directory che intendiamo usare come workspace:
  - ng new *my-app*
  - Si possono accettare le scelte di default proposte
- Alla fine del (non breve) processo
  - Cambiare directory a quella dell'app (*my-app*, in questo caso)
  - Compilazione, esecuzione dell'app e apertura del browser
    - ng serve -o
  - Per default il server corre su
    - http://localhost:4200/
    - ng serve --port *nnnn* → il server corre sulla porta specificata

ng serve

- **angular.json**, proprietà projects.*my-app*.architect.build.options.main
  - Indica il typescript da eseguire a startup, default: src/main.ts
- **main.ts** importa il modulo di partenza, default: AppModule definita in app/app.module.ts
  - Ne crea una istanza ed esegue il bootstrap
- **AppModule**, file app.module.ts, decorata da NgModule con le proprietà
  - declarations: lista di componenti definite nel modulo
    - L'esecuzione di "ng generate component" la aggiorna automaticamente
  - imports: dipendenze da altri moduli, per uso in template o per dependency injection (DI)
  - providers: servizi che devono essere disponibili via DI
  - bootstrap: componente per l'avvio dell'app, default: AppComponent
- **AppComponent** definisce l'elemento 'app-root' che viene usato nel body di **index.html**
  - Una applicazione Angular è una collezione di component usati in pagine HTML standard
  - Component Angular: incapsula dati, logica e view – può contenere altre component



Creazione di una component

- Nella root dell'applicazione

  ng generate component hello

```
C:\dev\my-app>ng generate component hello
CREATE src/app/hello/hello.component.html (20 bytes)
CREATE src/app/hello/hello.component.spec.ts (621 bytes)
CREATE src/app/hello/hello.component.ts (265 bytes)
CREATE src/app/hello/hello.component.css (0 bytes)
UPDATE src/app/app.module.ts (392 bytes)
```

- .component.ts contiene la definizione di una classe decorata
  - Il decorator ***Component***
    - Meta-informazioni
  - La classe *xyz**Component***
    - Implementa OnInit

```
@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {
  constructor() { }
  ngOnInit() {}
}
```

Il decorator Component

- Definisce le seguenti proprietà
  - selector: nome del tag associato al component
  - Il codice HTML associato è definito, a scelta, via una di queste due proprietà:
    - templateUrl: URL del file che lo contiene
    - template: inline
  - styleUrls: URL dei file in cui è specificato lo stile associato al solo elemento corrente
- Si può usare il nuovo elemento con il nome definito in ‘selector’
  - La sintassi **{{** expression **}}** permette di accedere membri di un component
    - Detta: template binding / mustache tag / string interpolation
    - Tipicamente solo per proprietà

```
<p>hello works!</p>
```

hello.component.html

```
<h1>{{title}}</h1>
<app-hello></app-hello>
```

app.component.html

Proprietà in Component

- Nella root dell'app, creo una nuova Component
  - ng generate component user
- Aggiungo l'elemento nella view dell'AppComponent
- Aggiungo una proprietà alla Component
  - e la inizializzo nel costruttore
- Modifico il frammento HTML associato

app.component.html

```
<h1>{{title}}</h1>
<app-hello></app-hello>
<app-user></app-user>
```

user.component.html

```
<span>{{ name }}</span>
```

user.component.ts

```
export class UserComponent // ...
  name: string;

  constructor() {
    this.name = 'Tom';
  }

  // ...
```

Una direttiva: NgFor

- Nella root dell'app, creo una nuova Component
  - ng generate component users
- Uso il nuovo elemento nella AppComponent
- Aggiungo un'array come proprietà nella Component
  - Inizializzata nel costruttore
- Nel frammento HTML associato metto un ngFor
  - Prefissato con asterisco, indica la modifica del DOM
  - Equivalente a un for each loop
  - Ogni loop genera un nuovo elemento

app.component.html

```
<h1>{{title}}</h1>
<app-hello></app-hello>
<app-users></app-users>
```

```
export class UsersComponent // …
  names: string[];

  constructor() {
    this.names = ['Tom', 'Bob', 'Sid'];
  }
  // ...
```

users.component.ts

```
let names = ['a', 'b', 'c'];
for (let name of names) {
    console.log(name);
}
```

```
<ul>
    <li *ngFor="let name of names">{{ name }}</li>
</ul>
```

users.component.html

Il decorator Input

- Modifica della component **user**
  - Importazione del decorator Input
  - Decorazione della proprietà name
  - Rimozione del set di name nel constructor
- Modifica della component **users**
  - Il template HTML accede la proprietà di user usando la sintassi [*property*]
    - [*property*] → ***property binding***
      - cambia la proprietà nel controller …
      - … la view viene aggiornata

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

@Component({ /* … */ })
export class UserComponent // …
  @Input() name: string;

  constructor() { }

  // …
```

```
<ul>
   <li *ngFor="let name of names">
      <app-user [name]="name"></app-user>
   </li>
</ul>
```

Gestire i form

- ng generate component addItem
- Elemento app-add-item in app.component.html
- Form in add-item.component.html
  - Input associati a template variable (#name)
  - Attributo (click) del submit button associato ad add(), che prende le template variable come parametri
    - (*event*)="*method()*"→***event binding***
- Nella classe AddItemComponent, il metodo add() gestisce la chiamata dal form

```
<h1>{{title}}</h1>
<!-- … -->
<app-add-item></app-add-item>
```

```
<h2>Add item</h2>
<form>
        <input placeholder="enter id" #id>
        <input placeholder="name ..." #name>
        <button (click)="add(id, name);">
           OK
        </button>
</form>
```

```
add(id: HTMLInputElement, name: HTMLInputElement): boolean {
  console.log(`(${id.value}, ${name.value})`);
  return false;
}
```

Applicazione

- È un albero di Component
  - La root dell’albero è una componente che rappresenta l’applicazione stessa
    - Il suo nome di default è AppComponent
    - Rappresentata dal tag HTML ‘app-root’
  - Il rendering di una componente implica il rendering dei suoi figli
- È una componente
  - Una applicazione può essere parte di un’altra applicazione
- Esecuzione dell’applicazione
  - ‘ng serve’ esegue main.ts, che (tra l’altro) importa l’AppModule corrente

Modulo

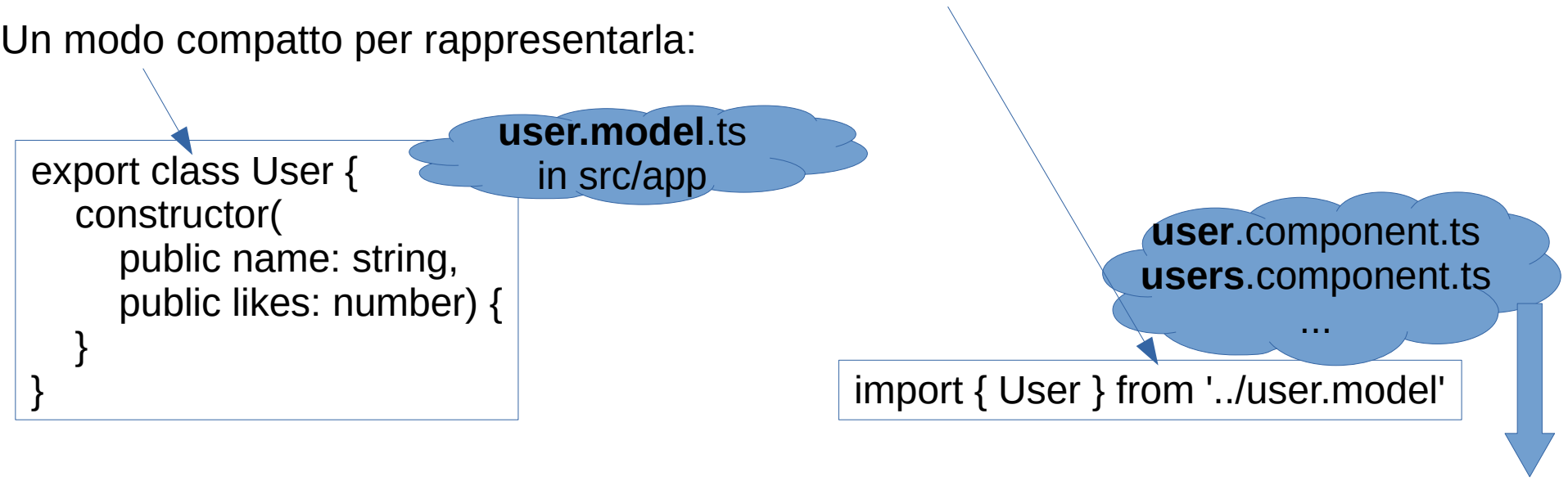
- Contenitore di component all’interno di una applicazione
  - Aiuta a organizzare le parti che gestiscono funzionalità comuni
- È una semplice classe
  - Nome di default AppModule
- Decorata con NgModule per specificare
  - declarations, imports, exports, providers, bootstrap
  - Decorator @: al momento disponibile in JS solo via transpiling
    - Funzione che decora (annota) un elemento del linguaggio

Componente

- Blocco fondamentale di applicazioni Angular
  - ng generate component *xyz*
  - Classe TypeScript che, per convenzione, ha un nome nella forma *xyz*.component.ts
- Composto da
  - Component decorator, configurazione del componente
    - selector: nome dell'elemento (o attributo per un div) HTML
    - template/templateUrl: codice HTML associato, descrive la view
    - styles/styleUrls: CSS per il solo componente corrente ed eventuali discendenti
  - Classe decorata, *Xyz*Component
    - Descrive il controller
- Accesso al controller dalla view: template binding
  - {{ expression }} → riferimento nell'HTML a proprietà/metodi del controller
- Per il test, viene creato un file karma: *xyz*.component.spec.ts

Model per component

- È spesso utile avere una classe che rappresenta il model relativo a una component
- ng generate class User --type=model
- import nei ‘component.ts’ che la usano (ad es. User e Users)
- Un modo compatto per rappresentarla:



Model View Controller

users.component.ts

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

import { User } from '../user.model'

@Component({ /* … */ })
export class UsersComponent implements OnInit {
  users: Array<User>;

  constructor() {
    this.users = [new User('Tom', 2), new User('Bob', 1), new User('Sid', 3)];
  }

  ngOnInit() {}

  moreLikes(user: User) {
    console.log(`Likes for ${user.name} are ${user.likes}`);
  }
}
```

@Input, @Output, EventEmitter

```
import {
  Component, OnInit, Input, Output, EventEmitter
} from '@angular/core';

import { User } from '../user.model'

@Component({ /* … */ })
export class UserComponent implements OnInit {
  @Input() user: User;
  @Output() liked: EventEmitter<User>;

  constructor() { this.liked = new EventEmitter(); }

  ngOnInit() { }

  plusOne() {
    this.user.likes += 1;
    this.liked.emit(this.user);
  }
}
```

user.component.ts

users.component.html

```
<ul>
   <li *ngFor="let user of users">
      <app-user [user]="user" (liked)="moreLikes($event)">
      </app-user>
   </li>
</ul>
```

l'evento generato

input / output binding

Subscribe implicito nell'assegnamento

output binding su un evento

```
<span>{{ user.name }}: {{ user.likes }}</span>
<div>
   <button (click)="plusOne()">Like</button>
</div>
```

user.component.html

Alcune direttive

- **ngIf**: rimosso se la condizione è false
- **ngSwitch**: alternativa a ngIf per scelta multipla
  - **ngSwitchCase**: sono ammessi valori duplicati
  - **ngSwitchDefault**: opzionale
- **ngStyle**: regola CSS
  - Forma semplificata: [style.color]='blue'
- **ngClass**: assegnazione di una classe
  - Ad es: condizionata a una variabile del component
- **ngFor**: ripetizione di elementi
  - Vedi slide 8
- **ngNonBindable**: esclusione dal binding
- ngForm: gestione dei form
  - ngSubmit: gestione del submit in un NgForm
- ngModel: two-way data binding

Attributo → comportamento dinamico all'elemento

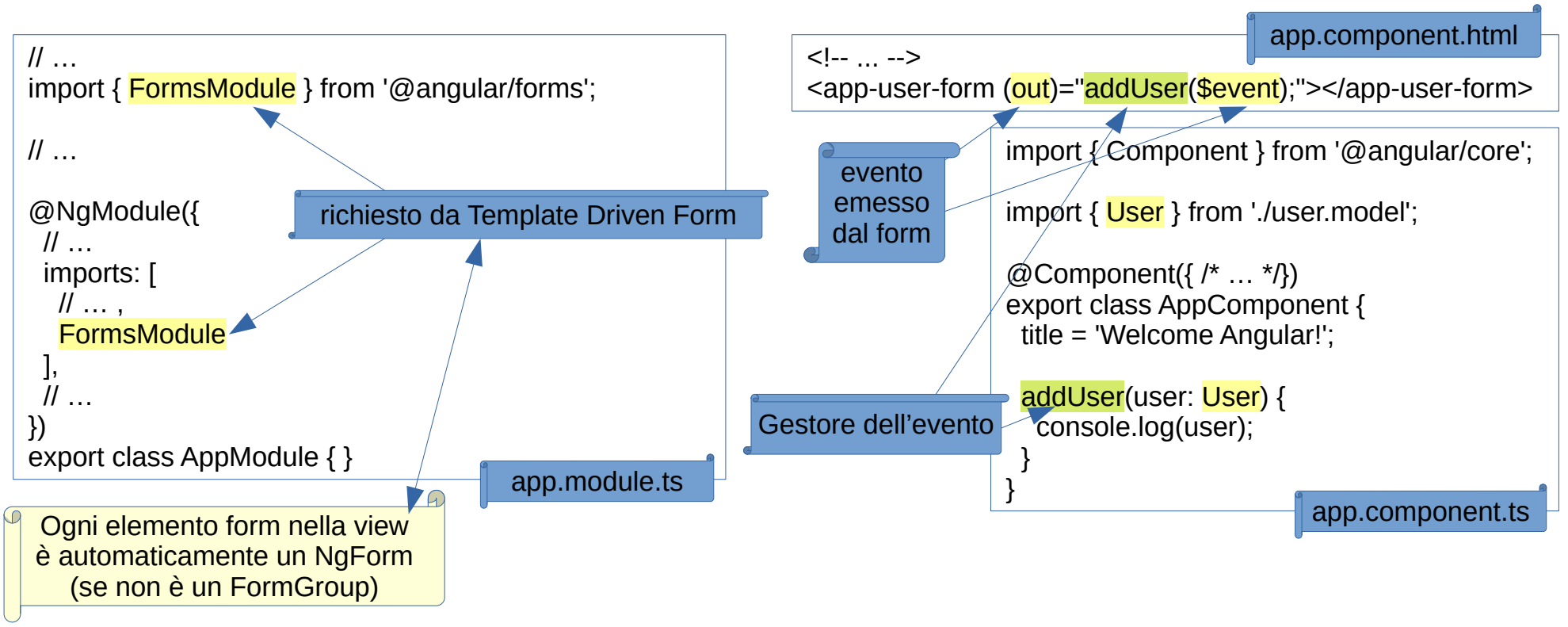
```
<span>{{ user.name }}: {{ user.likes }}</span>
<div>
   <div *ngIf="user.likes % 2" [ngStyle]="{color: 'blue'}">
      Odd number of likes
      <span ngNonBindable>{{unbound}}</span>
   </div>

   <div [ngSwitch]="user.name">
      <span *ngSwitchCase="'Tom'">Hi </span>
      <span *ngSwitchCase="'Bob'">Hello </span>
      <span *ngSwitchDefault>Good morning </span>
      <span [ngClass]="{aqua: back}">{{user.name}}</span>
   </div>
   <button (click)="plusOne()">Like</button>
   <button (click)="swapBack()">Swap Background</button>
</div>
```

user.component.html

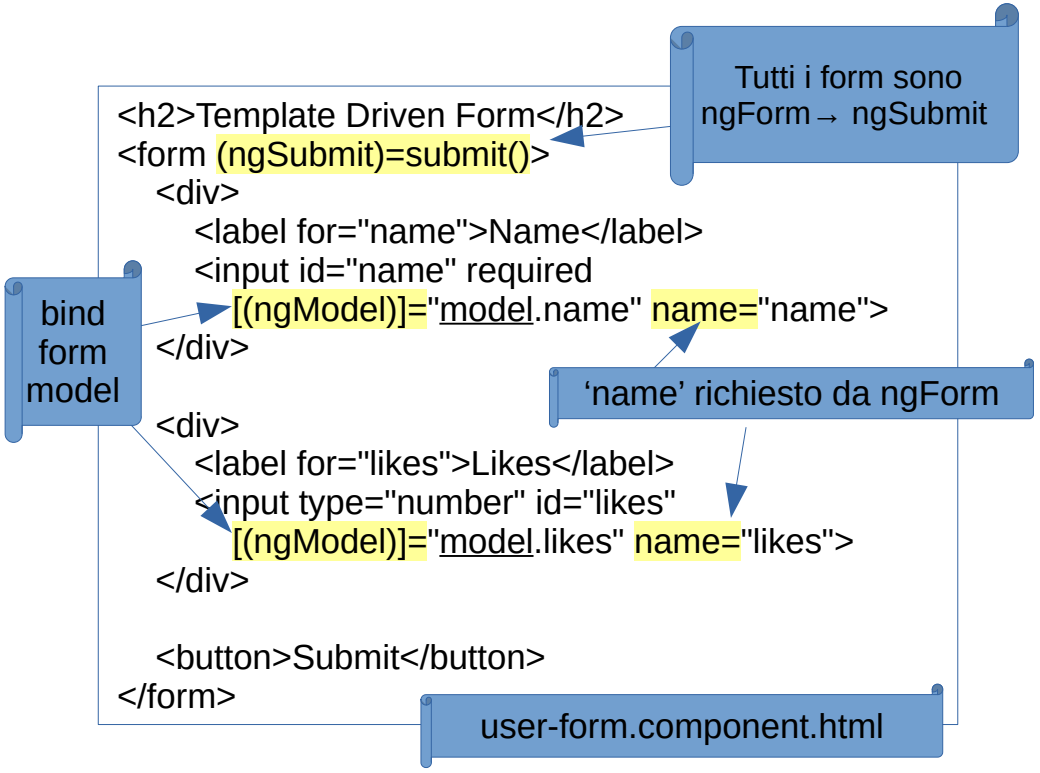
Template Driven Form

- Form Angular definito da
  - Una classe TypeScript per gestire dati e interazioni
    - ng generate component UserForm
    - Il modulo deve importare FormsModule
  - Un template basato su HTML
    - Usa le direttive ngForm, ngSubmit e ngModel

Setup


// …
import { FormsModule } from '@angular/forms';
// …
@NgModule({
// …
imports: [
// … ,
FormsModule
],
// …
})
export class AppModule { }
app.module.ts
richiesto da Template Driven Form
Ogni elemento form nella view
è automaticamente un NgForm
(se non è un FormGroup)
<!-- ... -->
<app-user-form (out)="addUser($event);"></app-user-form>
app.component.html
evento
emesso
dal form
Gestore dell'evento
import { Component } from '@angular/core';
import { User } from './user.model';
@Component({ /* … */})
export class AppComponent {
title = 'Welcome Angular!';
addUser(user: User) {
console.log(user);
}
}
app.component.ts

Form




```
<h2>Template Driven Form</h2>
<form (ngSubmit)=submit()>
   <div>
      <label for="name">Name</label>
      <input id="name" required
         [(ngModel)]="model.name" name="name">
   </div>

   <div>
      <label for="likes">Likes</label>
      <input type="number" id="likes"
         [(ngModel)]="model.likes" name="likes">
   </div>

   <button>Submit</button>
</form>
```

```
import { Component, OnInit, Output, EventEmitter }
from '@angular/core';

import { User } from '../user.model'

@Component({ /* … */ })
export class UserFormComponent implements OnInit {
  @Output() out = new EventEmitter<User>();
  model: User;

  constructor() { this.model = new User('Bill', 42); }

  submit() { this.out.emit(this.model); }

  ngOnInit() {}
}
```

user-form.component.ts

Reactive Driven Form

- Più flessibile di Template Driven Form
- Form Angular definito da
  - Una classe TypeScript per gestire dati e interazioni
    - ng generate component UserRForm
    - Uso di FormBuilder, FormGroup, FormControl
    - Il modulo deve importare FormsModule
  - Un template basato su HTML
    - Usa le direttive reactive form (formGroup)

Setup

```
// …
import { ReactiveFormsModule } from '@angular/forms';

// …

@NgModule({
  // …
  imports: [
    // … ,
    ReactiveFormsModule
  ],
  // …
})
export class AppModule { }
```

richiesto da Reactive Form

app.module.ts

app.component.html

```
<!-- ... -->
<app-user-rform></app-user-rform>
```

Form

```
<h2>Reactive Form</h2>
<form [formGroup]="fUser" (ngSubmit)="submit(fUser.value)">
    <div>
        <label>
            Name
            <input formControlName="name">
        </label>
    </div>

    <div>
        <label>
            Likes
            <input type="number" formControlName="likes">
        </label>
    </div>

    <button>Submit</button>
</form>
```

direttiva formGroup

user-rform.component.html

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

import { User } from '../user.model'

@Component({ /* … */ })
export class UserRFormComponent implements OnInit {
  private fUser: FormGroup;

  constructor(fb: FormBuilder) {
    this.fUser = fb.group(new User('Kim', 12));
  }

  submit(user: User) { console.log(user); }

  ngOnInit() {}
}
```

Dependency Injection

user-rform.component.ts

Pipe

- Definisce una funzione che opera una trasformazione sul suo input
- Usati in template expression
  - *expr | pipe [: arg] [| pipe …]*
- Tra i built-in pipes
  - currency, date
  - slice
  - uppercase, lowercase
- Esempi
  - {{ balance | currency:'EUR' }}
  - {{ today | date : dateFormat }}
  - {{ dessert | slice : 3 : 6 }}
  - {{ today | date | uppercase }}

```
export class PipExComponent implements OnInit {
  today: Date;
  isFull: boolean;
  balance: number;
  dessert: string;

  // …

  get dateFormat() {
    return this.isFull ? 'fullDate' : 'shortDate';
  }
```

pip-ex.component.ts

Custom pipe

- ng generate pipe *my*
  - Crea la classe MyPipe @Pipe implementa PipeTransform
  - Aggiorna app.module.ts (@NgModule dell'app)
- Metodo transform()
  - Input: il valore atteso e gli argomenti passati al pipe
  - Output: la trasformazione

Service

- Implementa funzionalità condivise da elementi dell'applicazione.
  - Esempio: FormBuilder
  - Uso tipico: data source
- Gestiti da Angular via **Dependency Injection** (DI)
  - Il decorator **Injectable** indica che va istanziato come Singleton
    - Il parametro passato indica chi gestisce DI
  - DI nelle componenti che lo usano
- Creazione di un nuovo servizio nell'app
  - ng generate service users

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

@Injectable({ providedIn: 'root' })
export class UsersService {
  constructor() { }
}
```

users.service.ts

Un servizio

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

import { User } from './user.model';

@Injectable({ providedIn: 'root' })
export class UsersService {
  private users: Array<User>;

  constructor() {
    this.users = [
      new User('Bob', 1),
      new User('Tom', 2),
      new User('Sid', 3)
    ];
  }

  get() { return this.users; }

  add(user: User) { this.users.push(user); }
}
```

users.service.ts

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

import { UsersService } from '../users.service';
import { User } from '../user.model'

@Component({/*…*/})
export class UsersComponent implements OnInit {
  users: Array<User>;

  constructor(us: UsersService) { this.users = us.get(); }

  ngOnInit() {}

  moreLikes(user: User) {
    console.log(`Likes for ${user.name} are ${user.likes}`);
  }
}
```

users.components.ts

Routing

- Single Page Application (SPA)
  - Basterebbe una sola URL
  - Ma si perderebbero alcuni vantaggi
    - Gestione dello stato dell’app (history, bookmarking)
    - Divisione dell’app in aree ad accesso regolamentato
- Il package Angular è @angular/router
  - supporta il client-side routing di HTML5

Esempio Routing

Routing1Component
Routing2Component
Routing3Component

```
// ...
import { RouterModule, Routes } from '@angular/router';

// ...
import { Routing1Component } from './routing1/routing1.component';
import { Routing2Component } from './routing2/routing2.component';
import { Routing3Component } from './routing3/routing3.component';

// ...
const appRoutes: Routes = [
  { path: 'one', component: Routing1Component },
  { path: 'two', component: Routing2Component },
  { path: 'three', component: Routing3Component }
];

// ...
@NgModule({
  imports: [
  // ...
  RouterModule.forRoot(appRoutes)],
  // ...
```

app.module.ts

```
<!-- … -->
<div>
   <h2>Routing Example</h2>
   <nav>
      <a routerLink="one">First</a> +
      <a routerLink="two">Second</a> +
      <a routerLink="three">Third</a>
   </nav>
   <router-outlet></router-outlet>
</div>
```

app.component.html

HTTP

- Libreria Angular per chiamate asincrone
- Tre diversi approcci supportati da JavaScript
  - Callback
  - Promise
  - Observable (preferito da Angular)
    - Elemento principale della libreria RxJS (ReactiveX)
    - Metodo subscribe(next, error)
      - Lambda per cosa fare con la response o in caso di errore

Esempi HTTP Observable

- Da una componente si effettua una chiamata HTTP
  - Sottoscrizione a HttpClient per ottenere la response
    - Classe iniettabile, alternativa Angular a AJAX / fetch
- Il server deve supportare Cross-origin resource sharing (CORS)
  - Esempio (1) Hello HTTP via componente MyHTTP
    - Backend Node.js https://github.com/egalli64/nesp (o Spring)
  - Esempio (2) REST su Coder: HTTP → service → model → component
    - Backend Spring RESTFul https://github.com/egalli64/sbr

Hello HTTP Observable

```
<h2>HTTP Request</h2>
<button type="button" (click)="request()">Send request</button>
<span *ngIf="!answered"> loading ... </span>
<span> {{message}} </span>
```

my-http.component.html

```
export class MyHttpComponent implements OnInit {
  message: string;
  answered: boolean = true;

constructor(private svc: HelloService) {}

request(): void {
  this.answered = false;
  this.message = '';
  this.svc.getData().subscribe(res => {
      this.message = res['message'];
      this.answered = true; },
    err => {
      this.message = err['message'];
      this.answered = true; });
};

// ...
```

successo

errore

my-http.component.ts

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

@Injectable({ providedIn: 'root' })
export class HelloService {
  readonly url: string = 'http://localhost:8080/hello';

  constructor(private http: HttpClient) { }

  getData() { return this.http.get(this.url); }
}
```

hello.service.ts

```
<!-- ... -->
<app-my-http></app-my-http>
```

app.component.html

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

// ...
import { MyHttpComponent } from './my-http/my-http.component';

// in @NgModule
// declarations→MyHttpComponent
// imports→HttpClientModule
```

app.module.ts

Esempio REST HTTP

- Model Coder
  - Basato sul JavaBean omonimo di sbr
  - ng generate class Coder –type=model
- Service Coder
  - Fornisce l'accesso a sbr come RESTful Web Service
  - Basato su HttpClient (DI), i metodi ritornano un Observable
  - ng generate service coder
- Component CoderGetAll
  - Usa il metodo findAll() del service
  - Visualizza i coders come tabella, con ngFor
  - ng generate component coder-get-all
- Component CoderSave
  - Usa il metodo save() del service
  - Immissione dei dati via Reactive Form
  - ng generate component coder-save

```
CREATE src/app/coder.model.spec.ts (156 bytes)
CREATE src/app/coder.model.ts (23 bytes)
```

```
CREATE src/app/coder.service.spec.ts (333 bytes)
CREATE src/app/coder.service.ts (135 bytes)
```

```
CREATE src/app/coder-get-all/coder-get-all.component.html (28 bytes)
CREATE src/app/coder-get-all/coder-get-all.component.spec.ts (663 bytes)
CREATE src/app/coder-get-all/coder-get-all.component.ts (301 bytes)
CREATE src/app/coder-get-all/coder-get-all.component.css (0 bytes)
UPDATE src/app/app.module.ts (2578 bytes)
```

```
CREATE src/app/coder-save/coder-save.component.html (25 bytes)
CREATE src/app/coder-save/coder-save.component.spec.ts (648 bytes)
CREATE src/app/coder-save/coder-save.component.ts (290 bytes)
CREATE src/app/coder-save/coder-save.component.css (0 bytes)
UPDATE src/app/app.module.ts (2596 bytes)
```

Unit test

- **Karma**: esecuzione dei test in un browser
  - karma.conf.js
  - Supporto Angular: **ng test**
    - test.ts → la variabile context specifica quali test eseguire
- **Jasmine**: descrive i risultati attesi in file *.spec.ts
  - describe(description: string, specDefinitions: () => void) // test suit, specs
    - beforeEach() // setup per ogni singolo test
    - it(expectation: string, assertion?: (done: DoneFn) => Promise<void>) // Test, spec
      - expect(actual: T) // singola spec, usa un matcher e ritorna un booleano che indica il successo del test
      - matchers: toBe(), toEqual(), toContain() …
      - pending() // test disabilitato

Esempio

```
export class SimpleService {
  // …
  negate(value: number) {
    return -value;
  }
  // …
}
```

simple.service.ts

```
const context =
  require.context('./', true, /simple\.service\.spec\.ts$/);
```

test.ts

```
// …

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

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(SimpleService);
  });

  // …

  it('should negate', () => {
    const result = service.negate(42);

    expect(result).toBe(-42);
  });

  // …
});
```

simple.service.spec.ts

Angular powered Bootstrap

- Bootstrap widgets the Angular way
  - https://ng-bootstrap.github.io/
- Usa solo Angular e Bootstrap.css (no js)
- Installazione via npm, dependencies:
  - https://ng-bootstrap.github.io/#/getting-started
- Ex: npm install --save @ng-bootstrap/ng-bootstrap bootstrap@4.3.1
- In angular.json, per il progetto, inserire tra gli styles
  - "node_modules/bootstrap/dist/css/bootstrap.min.css"
- In app.module.ts
  - import {NgbModule} from '@ng-bootstrap/ng-bootstrap';
  - imports: [NgbModule, /* … */]
- Se necessario, specifiche import nel controller

```
<ngb-alert type="success" [dismissible]="false">
    Success!
</ngb-alert>
```

```
<ngb-carousel ...>
  <ng-template ngbSlide>
  <!-- … -->
```

```
import { NgbCarouselConfig }
  from '@ng-bootstrap/ng-bootstrap';

// …

constructor(private config: NgbCarouselConfig) {
  config.showNavigationArrows = false;
  config.interval = 6000;
}
```