



Corso Front End Developer

Angular

Emanuele Galli

www.linkedin.com/in/egalli/

Angular











- Framework per lo sviluppo di webapp basato su TypeScript – Google 2016
 - <https://angular.io/>
 - “One framework. Mobile & desktop.”
- Evoluzione di AngularJS, sviluppato da Miško Hevery (2010)
 - Definizione di elementi HTML custom
- Basato su Node.js (al momento è richiesta versione 10.9+)
- Installazione via npm
 - Angular CLI (Command Line Interface): `npm install -g @angular/cli` 
 - Verifica della versione dell'Angular CLI installato: `ng --version`

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 8086** → il server corre sulla porta specificata

abbiamo avuto il codice su code digitando
sulla cmd della mia cartella "code ."
VEDI VISUAL CODE

ng serve

- **angular.json**, proprietà `projects.my-app.build.architect.options.main` determina l'esecuzione di `main.ts`   
-  `main.ts` importa la classe `AppModule` definita in `app/app.module.ts` 
- `AppModule` decorata da `NgModule` con le proprietà
 -  `declarations`: lista di componenti definite nel modulo, "ng generate component" la aggiorna automaticamente
 - `imports`: dipendenze da altri moduli, per uso in template o per DI (dependency injection) 
 - `providers`: servizi che devono essere disponibili via DI
-  `bootstrap`: componente per l'avvio dell'app – `AppComponent`
- `AppComponent` definisce l'elemento HTML 'app-root' 
- Che viene usato nel body di `index.html` 

sarà angular a generare la
connessione e se qualcosa va storto
angular tirerà l'eccezione

Creazione di un 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 class decorata
 - Component decorator
 - Meta-informazioni
 - `xyzComponent` class
 - 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 dell'elemento nel DOM
 - 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 dell'elemento
- Si può usare il nuovo elemento con il nome definito in 'selector'
 - La sintassi `{{ expression }}` (template binding/mustache tag) permette di accedere proprietà di una componente

hello.component.html

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

```
<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 il nuovo elemento alla app Component
- Aggiungo una proprietà alla sua 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>
```

```
export class UserComponent // ...  
  name: string;  
  
  constructor() {  
    this.name = 'Tom';  
  }  
  
  // ...
```

user.component.ts

La direttiva *ngFor

- Nella root dell'app, creo una nuova Component
 - ng generate component **users**
- Modifico la app Component per usare il nuovo elemento
- Aggiungo una proprietà array alla sua Component e la inizializzo nel costruttore
- Modifico il frammento HTML associato per eseguire un **for each loop** via direttiva ***ngFor**

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.html


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





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

looppo sugli elementi di quella classe







Il decorator Input

- Modifica del component user
 - Importazione del decorator Input
 - Decorazione della proprietà name
 - Rimozione del set di name nel constructor
- Modifica del component users
 - Il template HTML accede la proprietà di user usando la sintassi *[property]*

```
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) variabile locale 
 - Attributo **(click)** del submit button associato ad add(), che prende le template variable come parametri 
- Nella classe AddItemComponent, il metodo add() gestisce la chiamata dal form 

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

```
<form>
  <h3>Add item</h3>
  <div>
    <label for="id">GUID:</label>
    <input name="id" #id>
  </div>
  <div>
    <label for="name">Name:</label>
    <input name="name" #name>
  </div>
  <div>
    <button (click)="add(id, name);">
      OK
    </button>
  </div>
</form>
```

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



Applicazione

- Albero di Component
 - La radice è la componente App, ovvero l'applicazione stessa, indicata in angular.json
 - Per default il componente root ha nome AppComponent ed è rappresentata dall'elemento HTML con nome 'app-root'
- È un componente
 - Una applicazione può essere parte di un'altra applicazione
- Esecuzione dell'applicazione
 - 'ng serve' esegue main.ts, che importa (tra l'altro) l'AppModule corrente

Modulo


- Contenitore di funzionalità per applicazione
- È una semplice classe
 - Nome di default AppModule
- Decorata con NgModule per specificare
 - declarations, imports, providers, bootstrap
- Decorator @: introdotto in JavaScript con ES7
 - Funzione che decora (annota) un elemento del linguaggio

Componente

- Blocco fondamentale di applicazioni Angular
 - `ng generate component`
 - 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, `XyzComponent`
 - Descrive il controller
- Accesso al controller dalla view: template binding
 - `{{ expression }}` → riferimento nell'HTML a proprietà/metodi del controller 

Model per component

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



```
export class User {  
  constructor(  
    public name: string,  
    public likes: number) {  
  }  
}
```

user.model.ts

user.component.ts
users.component.ts
...

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

Model View Controller

```
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}`);  
  }  
}
```

users.component.ts

@Input e @Output



```
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

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

users.component.html

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

user.component.html

Directive

- ngIf: visualizzazione condizionale
- ngSwitch: scelta multipla
 - ngSwitchCase
 - ngSwitchDefault
- ngStyle: assegnazione di stile
- ngClass: assegnazione di classi
- ngFor: ripetizione di elementi
- ngNonBindable: esclusione dal binding
- ngForm: gestione dei form
- ngModel: two-way data binding

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

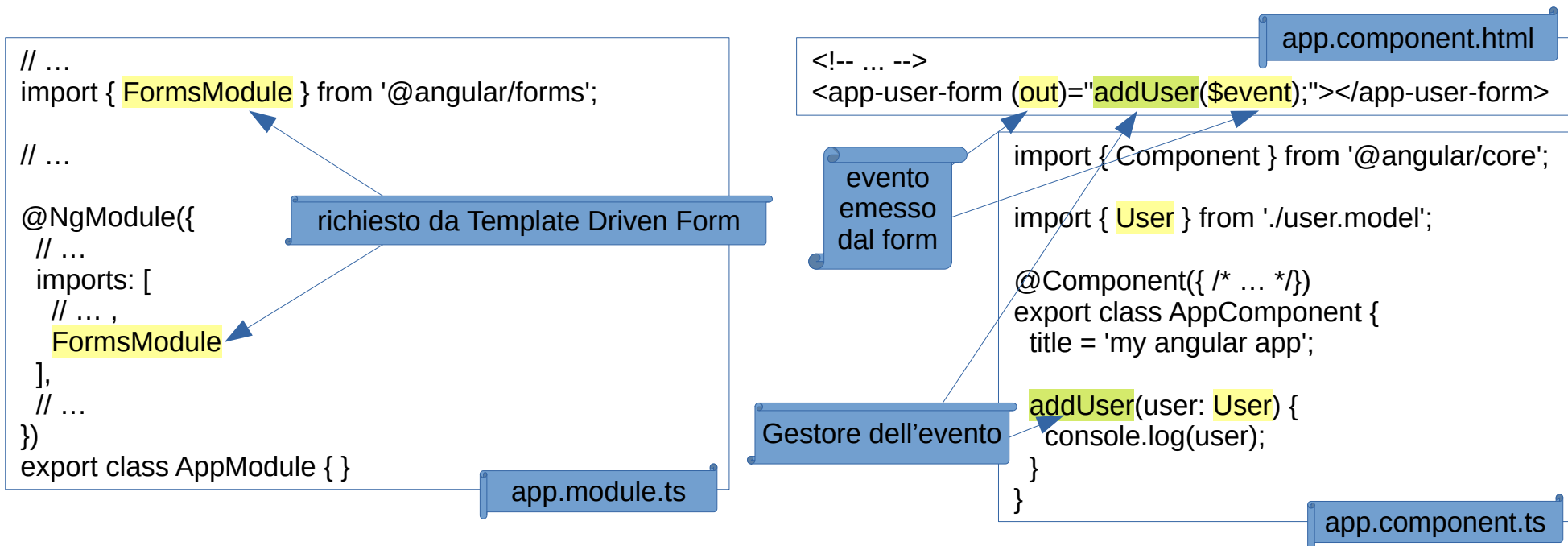
  <div [ngSwitch]="user.name">
    <span *ngSwitchCase="Tom" [ngClass]="{zzz: true}">
      Hi
    </span>
    <span *ngSwitchCase="Bob" [ngClass]="{zzz: false}">
      Hello
    </span>
    <span *ngSwitchDefault>Good morning</span>
    {{user.name}}
  </div>
  <button (click)="plusOne();">Like</button>
</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 (e ngModel)

Setup per component form



Un component form

```
<h1>User Form</h1>
<form #userForm="ngForm">
  <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 (click)="submit();">Submit</button>
</form>
```

Template variable
Reference alla
direttiva ngForm

bind
form
model

'name' richiesto da ngForm

user-form.component.html

```
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); }

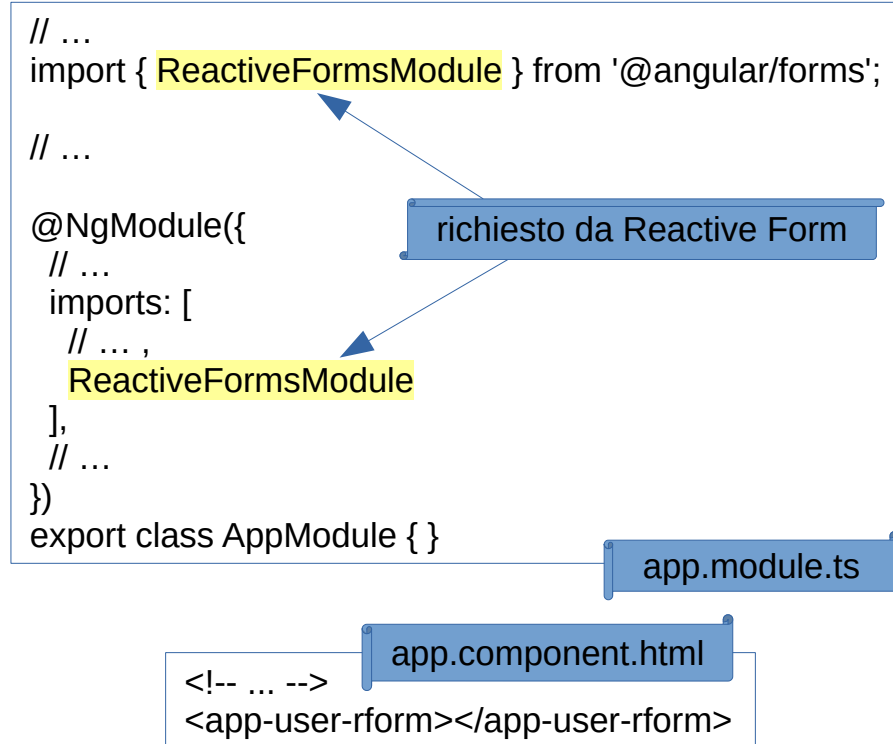
  ngOnInit() {}
}
```

user-form.component.ts

Reactive 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)

Reactive form setup



Un component reactive form

```
<h1>User Reactive Form</h1>
<form [formGroup]="fUser">
  <div>
    <label for="name">Name</label>
    <input formControlName="name">
  </div>

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

  <button (click)="submit(fUser.value);">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 {
  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

Service

- Classe che implementa funzionalità condivise da elementi dell'applicazione. Esempio: FormBuilder
- Gestiti da Angular via Dependency Injection
 - Supporto DI fornito al servizio via decorator Injectable
- 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('Tom', 2),
      new User('Bob', 1),
      new User('Sid', 3)
    ];
  }

  get(): Array<User> { return this.users; }

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

```
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}`);
  }
}
```

Routing

- Divisione dell'app in aree seguendo di solito regole basate sull'URL
- In una SPA si potrebbe avere una sola URL ma si perderebbero i vantaggi dei bookmark
- Il package Angular è @angular/router
 - supporta il client-side routing di HTML5

HTTP

- Libreria Angular per chiamate asincrone
- Tre diversi approcci supportati da JavaScript
 - Callback
 - Promise
 - Observable (preferito da Angular)