



Introduzione ad Angular 16

A cura di Christian Girardi

Argomenti

- ▶ Variabili Template
- ▶ Creare delle direttive
- ▶ Pipe
- ▶ Service
- ▶ Routing
- ▶ Routing Parametri
- ▶ Redirect ed errori
- ▶ Routing Guard
- ▶ Observable
- ▶ Template Driven
- ▶ Reactive Form
- ▶ modulo HTTP
- ▶ Build





Reactive Form

Per Reactive Forms intendiamo form gestiti interamente nel ts, per iniziare riprendiamo esattamente il form visto nei template driven e ripuliamolo da tutto:

```
<form #formPrincipale="ngForm" (ngSubmit)="onSubmit(formPrincipale)">
  <label for="fname">nome:</label><br>
  <input type="text" id="fname" name="fname" ngModel required><br>
  <label for="femail">email:</label><br>
  <input type="email" id="femail" name="femail" ngModel required email><br>
  <label for="lname">Cognome:</label><br>
  <input type="text" id="lname" name="lname" ngModel required><br><br>
  <button type="submit" [disabled]="!formPrincipale.valid">Invia</button>
</form>

<hr>

<form>
  <label for="fname">nome:</label><br>
  <input type="text" id="fname1" name="fname1"><br>
  <label for="femail">email:</label><br>
  <input type="email" id="femail1" name="femail1"><br>
  <label for="lname">Cognome:</label><br>
  <input type="text" id="lname1" name="lname1"><br><br>
  <button type="submit">Invia</button>
</form>
```

servizi works!

nome:

email:

Cognome:

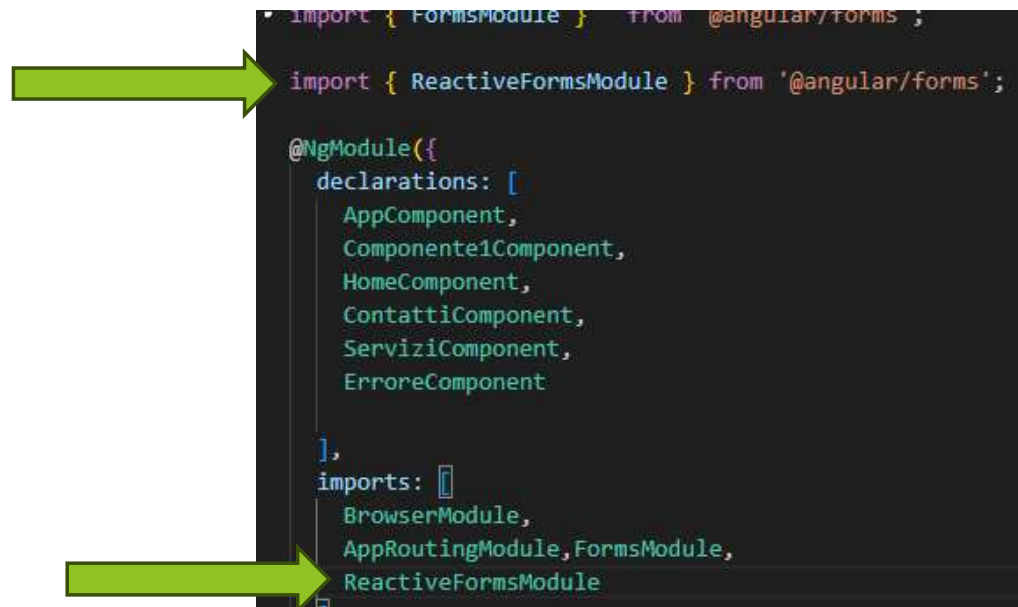
nome:

email:

Cognome:

Reactive Form

Andiamo su app.module ed aggiungiamo ReactiveFormsModule :



```
import { FormsModule } from '@angular/forms';  
  
import { ReactiveFormsModule } from '@angular/forms';  
  
@NgModule({  
  declarations: [  
    AppComponent,  
    Component1Component,  
    HomeComponent,  
    ContattiComponent,  
    ServiziComponent,  
    ErroreComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule, FormsModule,  
    ReactiveFormsModule  
  ]  
})  
export class AppModule { }  
export default AppModule;
```



Reactive Form

Ora torniamo nel ts del nostro componente che ospita il form e creiamo un form group:



```
@Component({
  selector: 'app-servizi',
  templateUrl: './servizi.component.html',
  styleUrls: ['./servizi.component.css']
})
export class ServiziComponent {
  formreattivo!: FormGroup;
```

Connettiamo il front con il backend:



```
<form [formGroup]="formreattivo">
  <label for="fname">nome:</label><br>
  <input type="text" id="fname1" name="fname1"><br>
  <label for="femail">email:</label><br>
  <input type="email" id="femail1" name="femail1"><br>
  <label for="lname">Cognome:</label><br>
  <input type="text" id="lname1" name="lname1"><br><br>
  <button type="submit">Invia</button>
</form>
```



Reactive Form

Continuiamo a scrivere il nostro codice in Oninit:

```
ngOnInit(): void {  
  
  this.formreattivo = new FormGroup({  
    nome: new FormControl(),  
    cognome: new FormControl(),  
    email: new FormControl(),  
  
  })  
}
```

Ora colleghiamo html:



```
<form [formGroup]="formreattivo">  
  <label for="fname">nome:</label><br>  
  <input type="text" id="fname1" name="fname1" formControlName="nome"><br>  
  <label for="femail">email:</label><br>  
  <input type="email" id="femail1" name="femail1" formControlName="email"><br>  
  <label for="lname">Cognome:</label><br>  
  <input type="text" id="lname1" name="lname1" formControlName="cognome"><br><br>  
  <button type="submit">Invia</button>  
</form>
```



Reactive Form

Provate a fare un test:

```
this.formreattivo = new FormGroup({  
  nome: new FormControl('test'),  
  cognome: new FormControl(),  
  email: new FormControl(),  
});
```

nome:

email:

Cognome:



Reactive Form

Andiamo a creare un metodo per il submit:

```
onSubmitReactive(){  
}
```

Modifichiamo html:

```
<form [formGroup]="formreattivo" (ngSubmit)="onSubmitReactive()">  
  <label for="fname">nome:</label><br>
```

```
onSubmitReactive(){  
  console.log(this.formreattivo);  
}
```



[servizi.component.ts:29](#)
► FormGroup {_pendingDirty: false, _hasOwnPendingAsyncValidator: false, _pendingTouched: false, _parent: null, _onCollectionChange: f, ...}

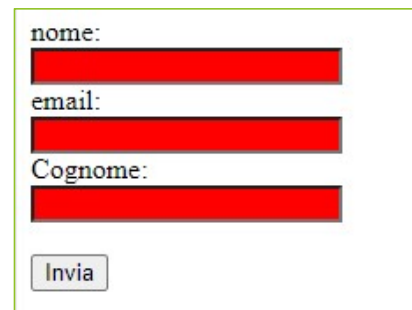


Reactive Form

Ora aggiungiamo le validazioni via codice:

```
this.formreattivo = new FormGroup({  
  nome: new FormControl(null, Validators.required),  
  cognome: new FormControl(null, Validators.required),  
  email: new FormControl( null, [Validators.required, Validators.email] ),  
})
```

Array di validatori



nome:
email:
Cognome:
Invia



Reactive Form

Per chiudere il form, applichiamo come per l'altra tipologia di esercizio vista precedentemente qualcosa al bottone:

```
<label for="lname">Cognome:</label><br>
<input type="text" id="lname1" name="lname1" formControlName="cognome"><br><br>
<button type="submit" [disabled]="!formreattivo.valid">Invia</button>
</form>
```

nome:
[red input field]
email:
[red input field]
Cognome:
[red input field]
Invia

N.B. per un accesso diretto al componente:

```
onSubmitReactive(){
  console.log(this.formreattivo);
  console.log(this.formreattivo.get('nome')?.value);
}
```

Reactive Form

Potremmo per esempio fare di meglio:



```
<form [formGroup]="formreattivo" (ngSubmit)="onSubmitReactive()">
  <label for="fname">nome:</label><br>
  <input type="text" id="fname1" name="fname1" formControlName="nome"><br>
  <p *ngIf="!formreattivo.get('nome')?.valid && formreattivo.get('nome')?.touched ">nome non valido</p>
  <label for="fname">email:</label><br>
```

A mockup of a web form with a light gray border. It contains four input fields, each with a red border indicating it is required. The first field is labeled 'nome:' and has the text 'nome non valido' below it. The second field is labeled 'email:'. The third field is labeled 'Cognome:'. At the bottom left of the form is a gray button labeled 'Invia'.

Modulo HTTP

E' un sotto-modulo che ci permette di lavorare con i dati esterni.

Per prima cosa si va su app.modules e si importa:



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

@NgModule({
  declarations: [
    AppComponent,
    Componente1Component,
    HomeComponent,
    ContattiComponent,
    ServiziComponent,
    ErroreComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule, FormsModule,
    ReactiveFormsModule,
    HttpClientModule
  ]
})
```



Modulo HTTP

Possiamo utilizzare direttamente HttpClient nel componente, ma è meglio accedervi tramite il servizio.

Creiamo un nuovo servizio con l'aiuto del comando angular-cli :

ng generate service nome-servizio

```
import { Injectable } from '@angular/core';  
⚡  
@Injectable({  
  providedIn: 'root'  
})  
export class NomeServizioService {  
  constructor() { }  
}
```



Modulo HTTP

Modifichiamo il codice del servizio:

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

@Injectable({
  providedIn: 'root'
})
export class NomeServizioService {

  constructor(private http: HttpClient) { }

}
```



Modulo HTTP

```
get<T>(url: string, options?: { headers?: [HttpHeaders];  
context?: [HttpContext];  
observe?: "body";  
params?: [HttpParams];  
reportProgress?: boolean;  
responseType?: "json";  
withCredentials?: boolean;  
}): Observable<T>
```



Modulo HTTP

options:

```
{  
  headers?: [HttpHeaders],  
  observe?: 'body' | 'events' | 'response',  
  params?: [HttpParams],  
  reportProgress?: boolean,  
  responseType?: 'arraybuffer' | 'blob' | 'json' | 'text',  
  withCredentials?: boolean,  
}
```



Modulo HTTP

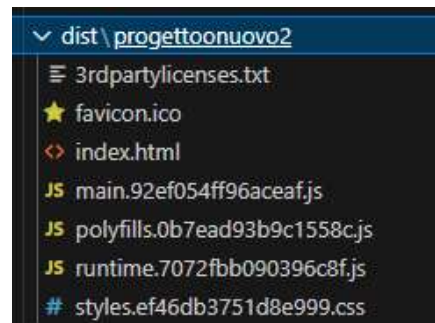
```
import { Component } from '@angular/core';
import { HttpService } from './http.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'titolo progetto';
  risultato : any;
  constructor(private httpService: HttpService) { }
  ngOnInit() { this.httpService.getValore().subscribe(
    (response) => { this.risultato = response; },
    (error) => { console.log(error); });
  }
}
```

NOME VOSTRO SERVIZIO



Build

L'istruzione che andremo a lanciare per costruire il nostro progetto è:
ng build



<https://angular.io/cli/build>



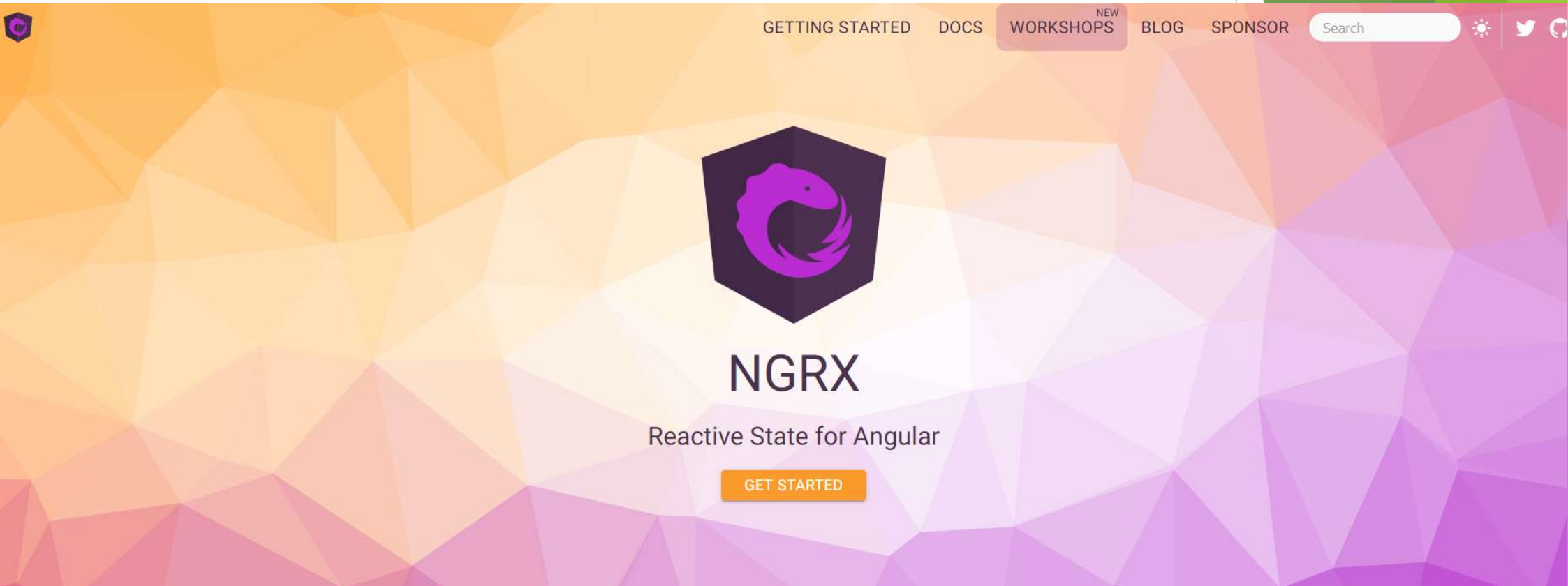
Redux Angular

Redux è una libreria di gestione dello stato per applicazioni JavaScript, spesso associata a React, ma può essere utilizzata con Angular e altri framework frontend. Redux si basa su un concetto di architettura a flusso unidirezionale dei dati, fornendo uno store centrale immutabile dove viene conservato lo stato dell'applicazione.

In Angular, Redux può essere integrato utilizzando librerie come `ngrx`, che fornisce un'implementazione di Redux specificamente progettata per Angular.



Redux Angular



**UMANA
FORMA**

Redux Angular

@ngrx/store è il pacchetto principale di ngrx che consente di gestire lo stato dell'applicazione attraverso un singolo store. Questo store contiene uno stato globale dell'applicazione che può essere accessibile e modificato da diversi componenti.

1. **Store centrale:** Uno store globale che contiene lo stato unico dell'applicazione.
2. **Flusso unidirezionale dei dati:** Gli stati sono immutabili e vengono aggiornati attraverso azioni.
3. **Azioni:** Descrivono le modifiche allo stato. Vengono dispacciate dai componenti e gestite dai reducers.
4. **Reducers:** Funzioni pure che specificano come lo stato dell'applicazione viene trasformato in risposta a un'azione. Riducono lo stato precedente a uno nuovo sulla base dell'azione ricevuta.
5. **Selettori:** Estraggono parti specifiche dello stato globale per essere utilizzate nei componenti.



Redux Angular

Utilizzando @ngrx/store in un'app Angular, si crea una struttura prevedibile e gestibile per lo stato dell'applicazione, consentendo la gestione centralizzata dei dati e semplificando il flusso di informazioni tra i componenti.

In sostanza, Redux in Angular (implementato tramite @ngrx/store) offre un modo robusto e prevedibile per gestire lo stato dell'applicazione in modo consistente, particolarmente utile in applicazioni complesse con molte interazioni tra i componenti.

<https://ngrx.io/docs>



Utilizzo template di terze parti

Angular Material

Angular Material è la libreria di componenti ufficiale di Angular, che offre una raccolta completa di UI e si mantiene al passo con le ultime funzionalità di Angular e le modifiche alle API. Offre anche un supporto integrato per l'accessibilità, generando markup per consentire la navigazione da tastiera e guidare le tecnologie assistive come gli screen reader.

<https://material.angular.io/>



Utilizzo template di terze parti

Bootstrap

è una libreria open-source costruita sulla base di Bootstrap CSS, che fornisce componenti e pattern di design che molti professionisti dello sviluppo già conoscono. Questo riduce la curva di apprendimento per i nuovi progetti, rendendola una scelta affidabile per realizzare applicazioni Angular in modo rapido ed efficiente.

<https://getbootstrap.com/>



Utilizzo template di terze parti

Kendo UI

Kendo UI è una libreria commerciale costruita tenendo conto delle prestazioni, garantendo tempi di caricamento rapidi e un'esperienza utente fluida. Offre inoltre temi e opzioni di stile per migliorare l'aspetto della vostra applicazione, oltre a un'ampia documentazione e a un team di supporto dedicato.

<https://www.telerik.com/kendo-angular-ui>



Utilizzo template di terze parti

PrimeNG

PrimeNG è una libreria open-source progettata per essere facile da usare e da personalizzare. Include anche funzioni avanzate di accessibilità e supporto all'internazionalizzazione, che la rendono un'ottima scelta per le applicazioni globali.

<https://primeng.org/>



Utilizzo template di terze parti

Nebular

Nebular è una raccolta di oltre 40 componenti Angular UI disponibili in quattro temi personalizzabili. La libreria, creata dalla società di sviluppo web Akveo, è dotata anche di un modulo di autenticazione utente e di un modulo di sicurezza basato su ACL per controllare l'accesso più granulare a risorse specifiche. Akveo può anche aiutarvi a creare la vostra applicazione di bacheca di amministrazione con il kit **ngx-admin** costruito grazie ai moduli Nebular.

<https://akveo.github.io/nebular/>



Utilizzo template di terze parti

<https://angular.io/guide/component-overview>



UMANA
FORMA

