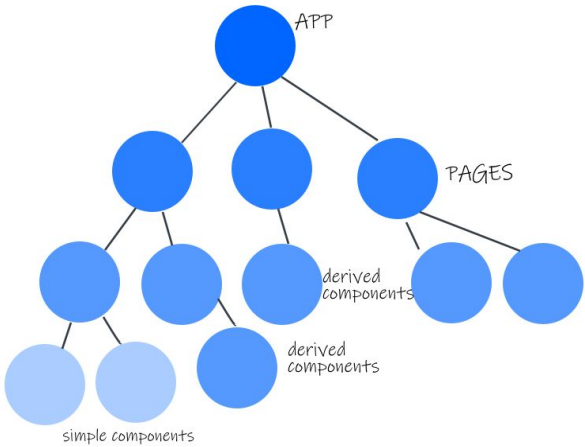
I 5 tipi di componenti in Angular e come progettare l'architettura di un'applicazione avanzata



Come progettare un'applicazione Angular? Che tipi di componenti ci sono e come strutturarli? Qual è il design migliore per organizzare i componenti di un'applicazione? In questo capitolo vedremo come progettare i componenti per garantire un alto livello di manutenibilità mantenendo semplice la struttura della nostra applicazione.

Autore

Ciao, sono Salvatore e trovi tutto su di me sul sito https://linkedin.com/in/salvatoreromeo. Ho scritto questo libro principalmente perché mi piace scrivere e condividere esperienza.

Nella vita mi occupo di sviluppo software fin da quando avevo 12 anni. Oggi è il mio lavoro principale. Insegno Ingegneria del Software all'Università degli Studi di Perugia e un corso di 3 giorni su Angular per le aziende insieme al Codemotion. Il corso è da poco stato trasformato in un libro che puoi trovare su Amazon.

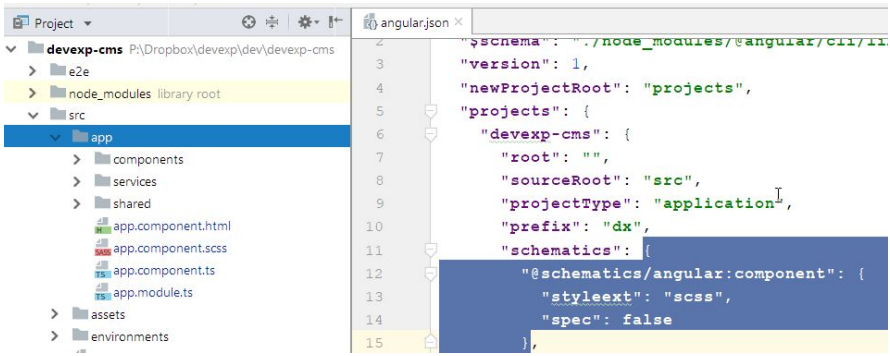
Se vuoi maggiori informazioni su di me contattami pure tramite linkedin o tramite il sito https://devexp.io. Lì potrai anche trovare articoli sul front-end, altri libri del gruppo devexp.io e i codici sorgenti pubblicati su Github.

Sommario

Introduzione

Progettare un'applicazione Angular non è facile. Progettare e sviluppare un'applicazione Angular di una certa importanza è ancora più difficile.
In questo articolo vedremo come organizzare una parte importante della nostra applicazione: i componenti.

Progettare un'applicazione Angular avanzata



Quando si usa un framework come Angular per applicazioni serie ben presto la semplice conoscenza del framework non è più sufficiente e lo sviluppatore ha bisogno di capire come usare al meglio tutte le possibilità del framework.
Usare al meglio il framework significa quali scelte di design preferire, come migliorare la struttura base di un'applicazione e come organizzare tutte le parti dell'applicazione al meglio, definendo:

- l'architettura dei componenti
- le API e le trasformazioni di dati da e verso il back-end
- il tema (senza legarsi troppo ad un particolare framework come Bootstrap)
- l'accesso ai dati (Redux, MVC, ...)
- le architetture multi-progetto con la creazione librerie
- l'integrazione di API HTML5 per gestione DB offline, API mobile e manipolazione file

In questo capitolo ci occuperemo di come progettare la struttura dei componenti, distinguendo i vari tipi di componenti a livello

più astratto e proponendo la soluzione migliore a livello di design.

Prerequisiti

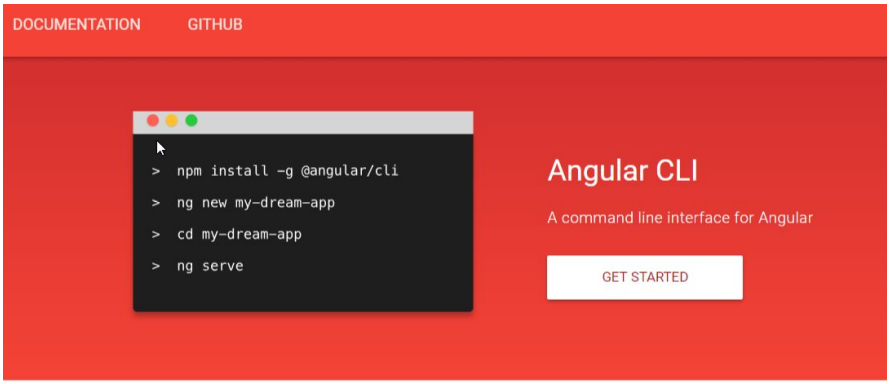
Per comprendere bene questo capitolo occorre una conoscenza base di Angular e in particolare sono necessari i seguenti concetti:

- componente Angular
- input e output di un componente
- Angular CLI
- transclusion

Per approfondirli si può fare riferimento al sito ufficiale https://angular.io o al libro di testo su Angular: https://www.amazon.it/dp/1983315591

L’architettura dei componenti

Quando la nostra applicazione Angular inizia a crescere avremo ormai creato centinaia di componenti. Se non adottiamo una struttura logica ben organizzata sin dall’inizio rischiamo di rendere il progetto poco manutenibile.



Un primo aiuto ci viene fornito dalla CLI, che per ogni componente creato (col comando *ng g c nome-componente*) adotta delle convenzioni utili:

- la separazione di logica, template e CSS in file separati
- un nome descrittivo per i file: *nome.component.ts*

Se però in fase di creazione non specifichiamo altro, la CLI inserisce tutti i componenti sotto la cartella *app*.

A livello di design questa scelta non è ottimale, perché avere centinaia di componenti sotto la stessa cartella rende il progetto ingestibile. Tra l’altro servizi, pipe e directive sarebbero ancora nella stessa cartella.

Una prima soluzione è quella di individuare componenti appartenenti alla stessa area e creare un nuovo modulo usando sempre la CLI:

ng g m MODULE_NAME

Poi possiamo ragggruppare i componenti appartenenti a questa stessa area aggiungendoli in questo specifico modulo. Usando la CLI la sintassi per ottenere questo obiettivo è:

ng g c MODULE_NAME/components/COMPONENT_NAME

Con questo comando la CLI crea il componente sotto la cartella *components* (distinguendo da servizi, pipe e directives) e nel modulo appropriato (aggiungendolo correttamente alle *declarations* dell'*@NgModule*).

Esiste però un modo per identificare correttamente un'area?
Cominciamo a distinguere i tipi di componenti per capire meglio quali moduli andare a creare.

I 5 tipi di componenti Angular

Dopo aver progettato decine applicazioni Angular anche complesse, posso dire che esistono 5 tipi di componenti:

- Componente root
- Componenti pagina
- Componenti semplici e derivati
- Componenti condivisi
- Componenti singleton

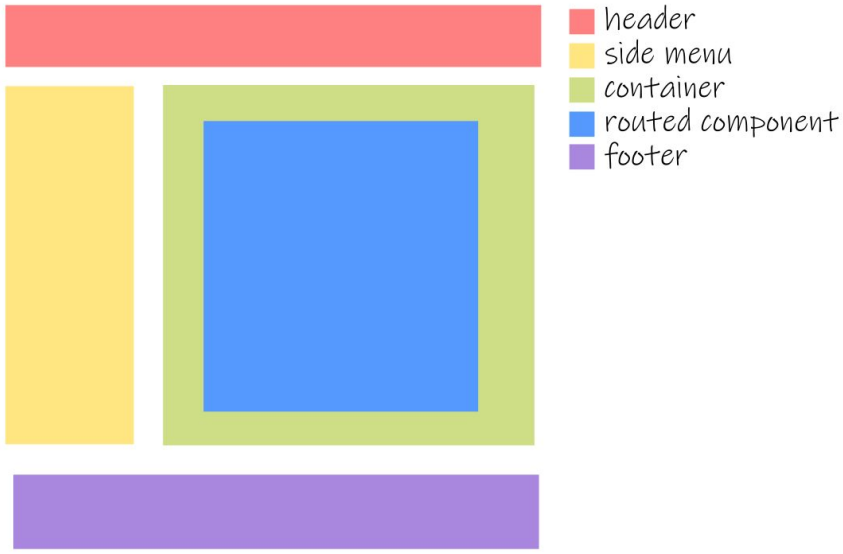
Andiamoli a conoscere uno per uno.

Componente root

Il componente root della nostra applicazione è il componente che si occuperà di contenere la struttura base:

- menu header e/o laterale
- footer
- contenitore stilizzato per i componenti pagina

Quando usiamo la CLI il componente root sarà generato automaticamente: si tratta del componente APP.



Se inoltre specifichiamo di adottare il routing in fase di creazione (col parametro *--routing*), la CLI creerà anche un modulo per le pagine di routing.
Nel componente root potremo anche definire lo stile per il contenitore di tutte le pagine, con un opportuno *padding* a *margin*.

Componenti pagina

Ogni volta che abbiamo necessità di avere una nuova pagina di routing possiamo creare un nuovo componente che raggrupperà tutti i sotto-componenti in quella pagina.
Una buona scelta di design è creare parallelamente al componente pagina anche un modulo per quella pagina.
Dentro questo modulo inseriremo il componente pagina stesso ed eventuali componenti specifici per quella pagina.
Se ad esempio voglio creare una pagina che rappresenta un blog, creerò il modulo *blog* e il componente *blog-page*.

ng g m blog
ng g c blog/components/blog-page

Dentro una pagina ci saranno tre tipi di componenti:

- componenti condivisi
- componenti semplici
- componenti derivati

Un componente pagina è infine responsabile di:

- caricare dati dal server usando le API
- gestire eventuali parametri dell'URL
- abilitare opportunamente parti del template
- passare gli input ai componenti usati nel template
- gestire gli output dei componenti usati nel template
- comunicare col server a seguito di eventi

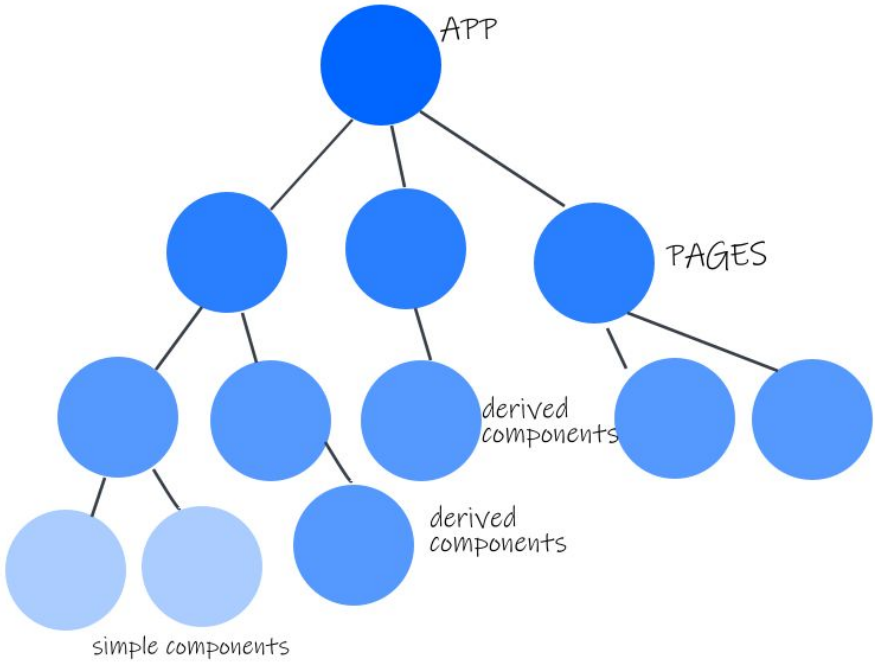
Componenti semplici e derivati

I *componenti semplici* sono quei componenti che all'interno contengono solamente elementi HTML5 nativi. Di solito sono creati per un caso d'uso specifico. Un *componente derivato* è invece composto da elementi HTML e/o da altri componenti derivati.
Ogni volta che un componente semplice diventa grande in termini di linee di codice allora è bene definire dei sotto-componenti, soprattutto se parte del template si ripete più volte nella pagina con la stessa struttura e logica.

Un componente semplice specifico per una pagina non ha mai nel costruttore dei parametri che lo accoppiano a dei servizi e, quindi, non farà mai chiamate al server. Un componente

derivato di pagina potrebbe farne, ma cercherei di gestire i dati definendo degli input se possibile.

La struttura della nostra applicazione comincia a prendere forma:



Quello sopra è un esempio di albero di componenti che descrive la struttura di un'applicazione a runtime.

Notiamo come un componente derivato può contenere anche altri componenti derivati e come una pagina può contenere anche componenti semplici.

Componenti condivisi

Quando un componente è talmente generico e può essere usato in diversi moduli pagina, allora possiamo definirlo all'interno di un modulo *shared*. Esempi di componenti condivisi potrebbero essere:

- componente *icon*
- componente *tabs*
- componente *loading*

I componenti condivisi possono far parte di una pagina oppure possono essere parte di componenti derivati.
Esistono anche **componenti condivisi complessi, che gestiscono eventualmente dei dati e delle chiamate al server** poiché ovunque vengono usati ci sarebbe bisogno di effettuare quella chiamata per inizializzare il componente.

Componenti singleton

Esiste infine un'ultima categoria di componenti, che ho voluto chiamare *componenti singleton*. Si tratta di tutti quei componenti che, quando attivi, compaiono al più una volta nella pagina.

Pensiamo ad esempio ai seguenti componenti:

- *modal* per mostrare informazioni
- *error* per mostrare che c'è stato un errore
- *toast* per notificare con un piccolo tooltip un'informazione

Questi componenti sono generalmente inseriti nel template dell'*app.component.html* e possiamo modificare il loro

contenuto tramite un servizio strettamente accoppiato con il componente:

- *ModalService* per il *modal*
- *ErrorService* per l'*error*
- *ToastService* per il *toast*

Ad esempio il componente error e il servizio corrispondente potrebbero essere così fatti:

error.component.ts:

```
@Component({
 selector: 'error',
 templateUrl: './error.component.html',
 styleUrls: ['./error.component.scss']
})
export class ErrorComponent
                implements OnInit {
 errorService: ErrorService;

 constructor(error: ErrorHandler) {
   this.errorService =
            error as ErrorService
 }

 ngOnInit() {
 }

 onClose() {
     this.errorService.discard()
 }
```

```
  reload() {
    window.location.reload()
  }

}
```

Template *error.component.html*:

```
<popup  [title]="errorService.error.code"
*ngIf="errorService.error"
(close)="onClose()">

  {{'error.internal' | translate}}
  <pre>
    {{errorService.error.message}}
  </pre>

  <button (click)="reload()">
    {{'error.reload' | translate}}
  </button>

</popup>
```

Per usare il componente scrivo in *app.component.ts*:

```
<error></error>
```

Per notificare un errore basterà iniettare l'*ErrorService* nel costruttore e usare i metodi per mostrare il messaggio d'errore opportuno.

Con i componenti singleton si conclude la nostra classificazione dei componenti. Proviamo a fare un esempio.

Componenti crud e routing

L'esempio classico che sarà presente in ogni applicazione è quello per la gestione di un sotto-sistema CRUD.

Immaginiamo di dover gestire gli utenti di un sistema. Come sarà la struttura di questa parte di applicazione?
Per come abbiamo classificato i componenti sopra mi aspetto innanzi tutto:

- un modulo *user.module.ts*
- un componente *user-page.component.ts*
- un componente *user-form.component.ts*

E nel nostro modulo di routing avremo anche i path *users* e *user* associati con i componenti definiti sopra.

Nel componente *user-page.component.ts* potrei avere ad esempio:

- una tabella per mostrare gli utenti
- un componente *loading* (componente condiviso)
- dei componenti *icon*: l'ultima cella di ogni riga potrebbe contenere delle azioni, ognuna come icona (componente condiviso)

Il componente *user-form.component.ts* conterrà:

- un form
- potrebbe contenere componenti derivati: ad esempio un componente *address-form.component.ts* se l'utente ha un indirizzo

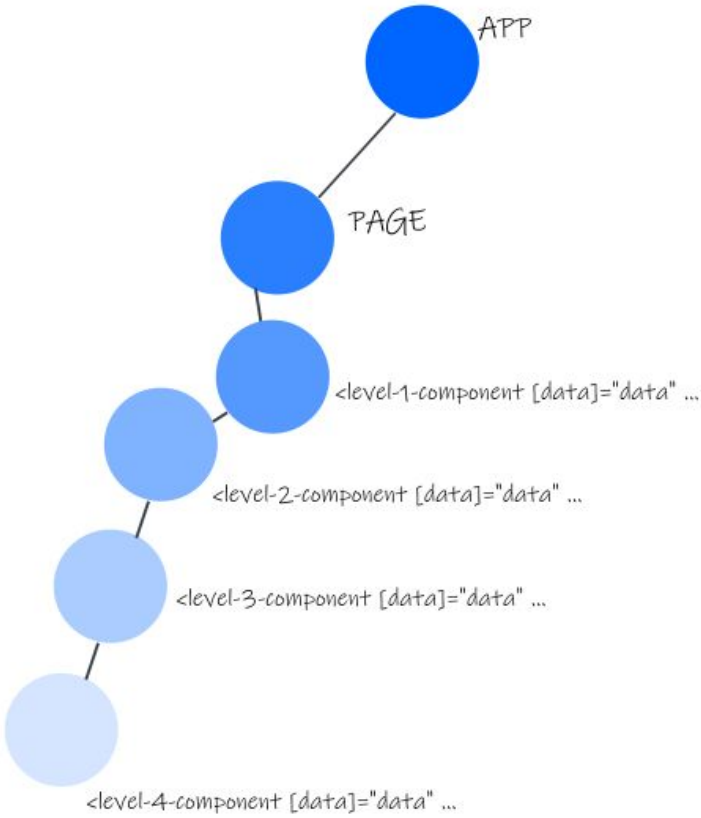
Annidare componenti dentro componenti: usare la translusion per evitare l'effetto catena di input/output

La struttura dell'albero di componenti può presto diventare complessa, ma a livello di design l'errore che non dobbiamo mai fare è di annidare troppi componenti uno dentro l'altro. Se il nostro albero di componenti ha più di 6 livelli, c'è qualcosa che non è progettato al meglio.
Inoltre ben presto si presenterà un problema molto fastidioso: l'*input-output chain*.
Mi piace descriverlo come una variante dello spaghetti code.

Immaginiamo questa situazione:

- Ho un componente derivato dentro un componente derivato dentro un altro componente derivato che sta dentro la pagina (un albero di componenti a 4 livelli + la root)
- Un componente nel quarto livello ha bisogno di un input.
- Un componente di quarto livello emette un output.

Ebbene quando la pagina si occupa di gestire i dati e passarli in input ad altri componenti, per passare un dato di input fino al quarto livello dovrei definire tutta una serie di **input accessori** a cascata, uno in ogni componente ad ogni livello. Questi input accessori non hanno alcuno scopo se non permettere di arrivare all'ultimo livello.



Con questo design il componente *level-1-component* sa che deve passare un dato al *level-2-component* per il *level-4-component*, mentre per un buon design non dovrebbe sapere come è fatto un componente interno. Possiamo eliminare questo input?

Analogamente avremo lo stesso problema per l'output quando la pagina deve gestire un evento del componente più in basso nell'albero.

Per ovviare a questo problema possiamo sfruttare la *transclusion*.

L'idea è di definire anche il componente di quarto livello direttamente nel template del componente pagina con un risultato di questo tipo:

```
<level-1-component>
 <level-2-component>
   <level-3-component>

     <level-4-component
       [data]="dataDefinedIntoPage">
     </level-4-component>

   </level-3-component>
 </level-2-component>
</level-1-component>
```

In questa nuova versione abbiamo diversi vantaggi:

1. La pagina passa direttamente il dato al componente di livello 4 senza passare per i componenti intermedi
2. Non devo definire input accessori in ogni componente solo per passare un dato al componente di livello 4
3. Il codice è molto più leggibile

A livello implementativo, rispetto a prima dobbiamo aggiungere il tag <*ng-content*> nei componenti 1-2-3 per implementare la transclusion, ma niente di più. Potremmo anche forzare il fatto che dentro <*level-1-component*> ci può stare solo un componente di tipo <*level-2-component*> con questa sintassi:

<ng-container select="level-2-component"><ng-content>

Conclusioni

In questo capitolo abbiamo visto come progettare l'architettura di un'applicazione relativamente alla parte dei componenti. Nei prossimi capitoli ci occuperemo di approfondire altri aspetti per ottenere il design ottimo in un'applicazione Angular avanzata.