Open in app Get started

Published in Angular In Depth

Christian Lüdemann Follow
Nov 19, 2018 · 6 min read · Listen

Save

Creating Reusable Angular Components — How To Avoid the Painful Trap Most Go In

> **AngularInDepth is moving away from Medium. More recent articles are hosted on the new platform inDepth.dev. Thanks for being part of indepth movement!**

To reuse where it is possible is a natural instinct in all aspects of life. The idea that some know-how to efficiently create reusable components that will be flexible enough for your use cases while being worth the effort of making it reusable.

When having multiple teams working together it makes sense to reuse components between projects. There are two main ways to create reusable components in Angular:

1. Pass **inputs to the component**, passing the necessary data to the component used for rendering and configuring the component. This normally involves iterating over the provided data and follow a convention for how to render the data.

2. Use **transclusion/content projection** to pass a template to the reusable component and **show the templateRefs inside the component** using the **ngTemplateOutlet** directive or **ng-content**

The choice of which technique to reuse components you should use is determined by the desired flexibility. If you have a simple reusable component that doesn't need to be very flexible, simply using inputs will do. An example of this is a simple questionnaire that should be dynamically rendered using json data from an API.

On the other hand, this becomes a pain when you need to pass lots of inputs to the component to provide the necessary data to the component. For example, you might need a fieldsDefinitions and actionDefinitions input for determining the fields and different actions in a list and inside the reusable component, you then need to have a lot of logic to render the input data. I have seen cases here where this has escalated to creating a dedicated DSL for rendering components. This gets very painful as the amount of input keeps growing, as well as the complexity of the reusable component as it should handle more edge cases, in the end **making the component harder to reuse** than if it were simply **copied and modified** to serve the purpose.

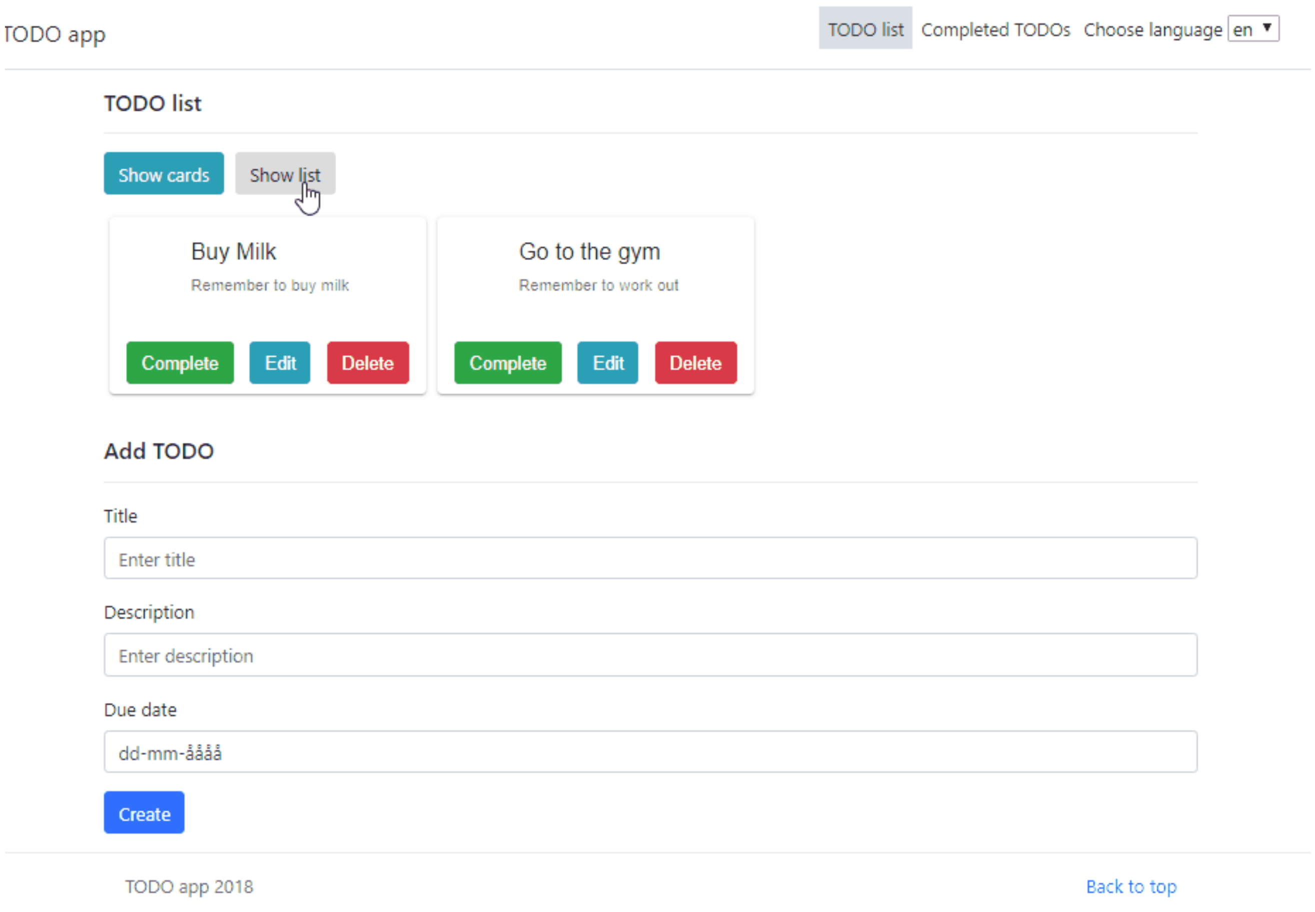
What you want to do instead is allow for an external template to plug into the component using either templateref or ng-content (check my plugin architecture post for an example of this with ng-content). In summary, use **template projection** when **more flexibility is needed** for the reusable component.

Should you use template reference or ng-content?

There is a subtle difference between using templateRef vs. using ng-content because of component containing the ng-content. Also for a child component being instantiated with ng-content, the constructor and init hooks will also be **invoked regardless of if the child component has been rendered** in the DOM. For that reason, passing the template projection as templateRef is the most maintainable and performant, as the lifecycle hooks are only getting called if the templateRef have actually been rendered in the DOM and because it gets destroyed with the component instantiating the templateRef.

Creating a reusable card/list view component

To illustrate how you should keep components reusable and maintainable we are going to create a reusable card list component, which can toggle between cards and a list. This is based on my Angular todo app demo, a simple TODO management application:



transclusion, that is passing template references to the reusable component. This is going to cause **slightly more duplication** but **easier use and maintenance** of the reusable component. The point is: less code duplication is not beneficial it if it makes the code **harder to use and maintain**.

We want to create a card-list component that takes in a listRef, cardRef and data to be shown and is used like this:

```html
<div class="todo-list-wrapper">
  <div class="mx-auto col-10">
    <h5>{{'todo-list' | translate}}</h5>
    <hr>

    <app-cards-list [listRef]="todoListRef" [cardRef]="todoItemCardRef" [data]="todo
  </div>

  <hr>

  <app-add-todo [currentTODO]="currentTODO"></app-add-todo>
</div>

<ng-template #todoItemCardRef let-todo="data">
  <app-todo-item-card [todoItem]="todo" (todoDelete)="deleteTodo($event)" (todoEdit)
</ng-template>

<ng-template #todoListRef let-todos="data">
  <ul class="list-group mb-3">
    <app-todo-item-list-row *ngFor="let todo of todos" [todoItem]="todo" (todoDelete
  </ul>
</ng-template>
```

todo-list.component.html hosted with ❤ by **GitHub** view raw

Note how simple the interface of the cards-list component is because we simply utilize templateRefs and map data using `let-todos="data"` which will map data to todos when we are passing data to a templateRef with `ngTemplateOutletContext`.

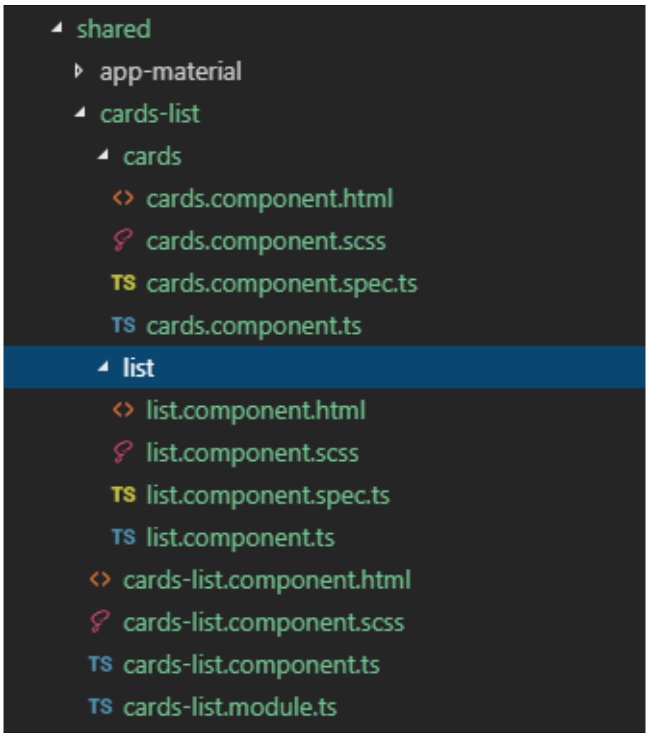
We are then going to create the card-list component.

Open the terminal, go to shared folder and type:

```
ng g c cards
```

```
ng g c list
```

We should now have:



```
shared
  app-material
  cards-list
    cards
      cards.component.html
      cards.component.scss
      cards.component.spec.ts
      cards.component.ts
    list
      list.component.html
      list.component.scss
      list.component.spec.ts
      list.component.ts
    cards-list.component.html
    cards-list.component.scss
    cards-list.component.ts
    cards-list.module.ts
```

Since the only input we are working with here is template refs and data, to be shown in the template refs, we are going to have very simple presentation components. The list component looks like this:

```html
<ng-container [ngTemplateOutlet]="listRef" [ngTemplateOutletContext]="{data: data}"><
```

list.component.html hosted with ❤ by **GitHub** **view raw**

To render this it only takes in a listRef and the data to render the list.

The cards component template is slightly different because it is iterating over each item (todo item in this case), and are rendering them using `ngTemplateOutlet` and is setting the data for the ngTemplateOutlet with `ngTemplateOutletContext` . It is setting the data which in our templateRef is passed to the todo data using `let-todo="data"` .

```html
    <ng-container *ngIf="cardRef; else noCard" [ngTemplateOutlet]="cardRef" [ngTempl
    </ng-container>
  </div>
</div>

<ng-template #noContent>
  <div class="no-data">
    {{'taskCards.noData' | translate}}
  </div>
</ng-template>
<ng-template #noCard>
  <div class="no-data">
    {{'taskCards.noCardRef' | translate}}
  </div>
</ng-template>
```

cards.component.html hosted with ❤ by **GitHub** **view raw**

Some styling is applied to these cards to make them wrap nicely:

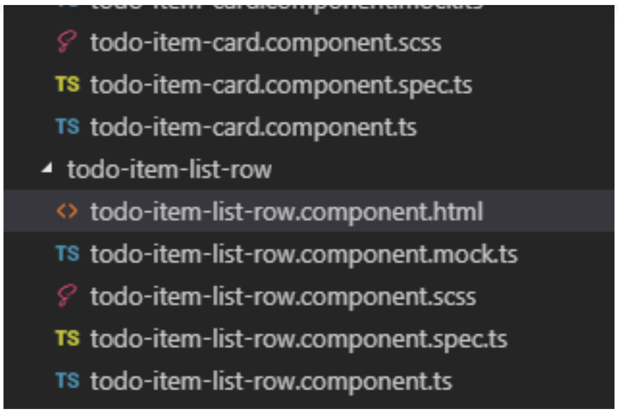
```scss
.cards-wrapper {
  display: flex;
  flex-wrap: wrap;
  flex-direction: row;
}

.card-item {
  min-width: 280px;
  margin: 5px;
  flex-basis: 280px;
}
```

cards.component.scss hosted with ❤ by **GitHub** **view raw**

Now we can display the card-list component and easily change the cards or list by simply changing the template ref provided.

The card and list row components are created like presentation/dumb components in the shared folder:


todo-item-card.component.scss
todo-item-card.component.spec.ts
todo-item-card.component.ts
todo-item-list-row
todo-item-list-row.component.html
todo-item-list-row.component.mock.ts
todo-item-list-row.component.scss
todo-item-list-row.component.spec.ts
todo-item-list-row.component.ts

The card component is created with **Angular Material** directives:

```
<mat-card class="todo-card" [ngClass]="this.todoItem.completed ? 'bg-completed' :  '
  <mat-card-header>
    <div mat-card-avatar class="example-header-image"></div>
    <mat-card-title>{{todoItem.title}}</mat-card-title>
    <mat-card-subtitle>{{todoItem.description}}</mat-card-subtitle>
  </mat-card-header>
  <mat-card-content class="card-content">
    <p *ngIf="todoItem.dueDate">
      <small>{{'add-todo.due-date' | translate}}:
        <b>{{todoItem.dueDate}}</b>
      </small>
    </p>
  </mat-card-content>
  <mat-card-actions class="card-actions">
    <button (click)="completeClick()" type="button" class="btn btn-success" aria-lab
      {{'todo-item.complete' | translate}}
    </button>
    <button *ngIf="!readOnlyTODO" (click)="editClick()" type="button" class="btn btn
      {{'todo-item.edit' | translate}}
    </button>
    <button *ngIf="!readOnlyTODO" (click)="deleteClick()" type="button" class="btn b
      {{'todo-item.delete' | translate}}
    </button>
  </mat-card-actions>
</mat-card>
```

The list row is created with Bootstrap (got to spice stuff up):

```
        <h6 class="my-0">{{todoItem.title}}</h6>
        <small class="text-muted">{{todoItem.description}}</small>
        <div *ngIf="todoItem.dueDate">
          <small>{{'add-todo.due-date' | translate}}:
            <b>{{todoItem.dueDate}}</b>
          </small>
        </div>
      </div>

      <div class="align-right btn-group-vertical">
        <button (click)="completeClick()" type="button" class="btn btn-success" aria-l
          {{'todo-item.complete' | translate}}
        </button>
        <button *ngIf="!readOnlyTODO" (click)="editClick()" type="button" class="btn b
          {{'todo-item.edit' | translate}}
        </button>
        <button *ngIf="!readOnlyTODO" (click)="deleteClick()" type="button" class="btn
          {{'todo-item.delete' | translate}}
        </button>
      </div>
    </li>
  </div>
```

These are being used as template references in the reusable component.

The complete demo can be found on my Github.

Conclusion

In this post, we looked at two ways of creating a reusable component. The first way only works for simple components as they will become harder to use and maintain if the complexity grows because of lots of input to be configured for the specific use case and as well as a lot of “duct tape” programming to handle all the different applications of the reusable component. The way of handling this is by using **transclusion instead of using input data and conventions** of how to render this. Transclusion using templateRef **is preferred over ng-content** performance and maintenance wise because it keeps the life cycle in sync with where it is used as well as supporting conditional instantiation.

rows and actions in lists and cards. This quickly becomes tiresome because it is not scaling to more complex usages. What we do instead is we used **templateRefs** for a card and a list, as well as the data to display, and created this reusable component in an easily maintainable way using only three inputs. The lesson of the day is: **reusable components should be easy to use as well as easy to maintain**.

If you liked this post, make sure to follow me on Twitter and give some feedback in the comment section.

Originally published at Christian Lüdemann IT.

Thanks to Tim Deschryver