

# Become a Content Projection Artist

## ALAIN CHAUTARD

In this workshop, we'll see how content projection works in Angular, and we will illustrate the many different use cases of content projection to build truly generic components such as tabs, cards, and pop-ups that can be customized for several different use cases. Topics will include multi-slot content projection as well as passing template references to components.

### About me

Alain Chautard (or just Al)



Google Developer Expert in Web technologies

/ Angular / Google Maps

Angular JS addict since 2011

Web consultant / trainer @ [angulartraining.com](http://angulartraining.com)

Organizer of the Sacramento Angular Meetup group / GDG Sacramento



We will see different techniques of creating reusable components in angular using content projection.

### Quick Poll

- Who is using Angular Material or another component library to build Angular apps?
- Who is building or planning on building their own library of reusable components?
- Who is new (or new-ish) to Angular and/or web development?

These slides have several links to examples and exercises so here's the link to access them:

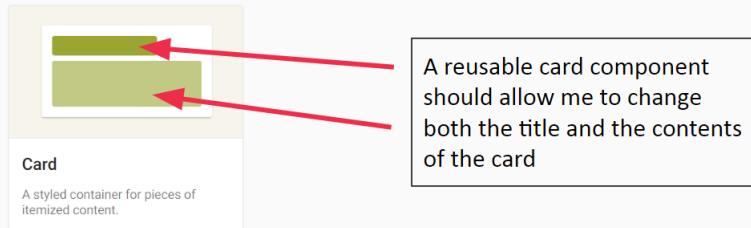
<https://bit.ly/at-cp-workshop>



# Why do we need Content Projection?

## Why Content Projection?

- In order to have truly reusable components, it is important that such components are built in a way that is **as generic as possible**
- This means that such components should be customizable
- The component should not be tied to any business logic



We have a Card component that we should be able to put different things like a title, description, image, etc. The Card component has to be really generic and not tied to any business logic at all.

## Why not just Inputs?

- In many cases, we can achieve component reusability using **inputs** because all we customize is text or icons, or a list of such texts or icons:

**Menu**  
A floating panel of nestable options.

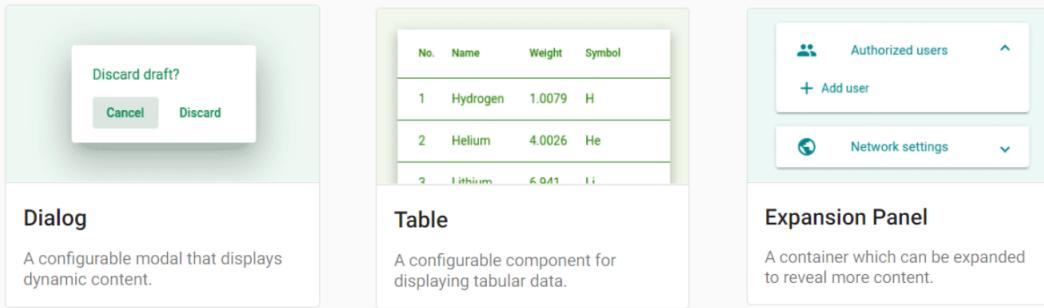
**Amenities**  
Presents a list of items as a set of small, tactile entities.

**Snackbar**  
Displays short actionable messages as an unobtrusive alert.

Recall that we can use inputs to pass data (title, description, image URL) into a component in angular, then just display it in the component. Yes, inputs work for simple scenarios like a menu, snack bar, chips, where all we need to pass as a parameter to the component are strings or images...we can't use inputs when passing complex stuff like angular components, directives, pipes etc.

When we need to customize more than just text...

- **Content projection** comes to the rescue when we need to customize a lot of content in the reusable component:

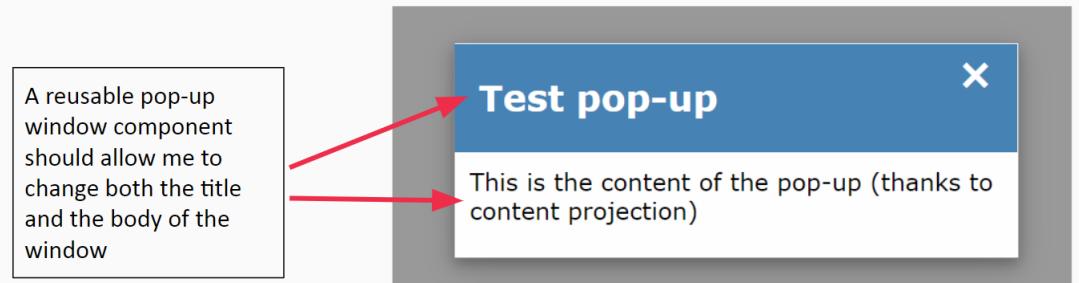


More complex components require more stuff like content projection.

## Option #1 Basic Content Projection

Let's take a look at a pop-up window component (dialog)

- Our next examples are going to illustrate a pop-up window/dialog component that can render any content passed from a parent component:



We want the title and the body of the component should be anything we want, this helps reusability

## We pass content from the parent component:

```
<app-popup-window [isOpen]="showPopup" title="Test pop-up" (onClose)="popupClosed($event)">
  This is the content of the pop-up (thanks to content projection)
</app-popup-window>
```

[app.component.html](#)

Content gets projected into the template of the component:

```
<div class="modalW-content">
  <div class="modalW-header">
    <span class="close" (click)="closePopup()">&times;</span>
    <h2>{{ title }}</h2>
  </div>
  <div class="modalW-body">
    <ng-content></ng-content>
  </div>
</div>
```

[popup-window.component.html](#)



We have a parent component that displays the popup window. We then pass some text (a template or component can also be passed in) in the content section of the component. We then use the ng-content directive to project the template of the child component.

The screenshot shows the StackBlitz editor interface. On the left, the code for `app.component.html` is displayed:

```
<p>
  <button (click)="showPopup = true">Show pop-up</button>
</p>
```

On the right, the browser preview shows a single button labeled "Show pop-up".

The screenshot shows the StackBlitz editor interface. On the left, the code for `popup-window.component.html` is displayed:

```
<p>
  <button (click)="showPopup = true">Show pop-up</button>
</p>
<app-popup-window
  [isOpen]="showPopup"
  title="Test pop-up"
  (onClose)="popupClosed($event)"
>
  This is the content of the pop-up (thanks to content projection)
</app-popup-window>
```

On the right, the browser preview shows a single button labeled "Show pop-up".

The screenshot shows the StackBlitz editor interface. On the left, the code for both `app.component.html` and `popup-window.component.html` is displayed:

```
<p>
  <button (click)="showPopup = true">Show pop-up</button>
</p>
<app-popup-window
  [isOpen]="showPopup"
  title="Test pop-up"
  (onClose)="popupClosed($event)"
>
  This is the content of the pop-up (thanks to content projection)
</app-popup-window>
```

On the right, the browser preview shows a modal window titled "Test pop-up" with the content "This is the content of the pop-up (thanks to content projection)".

A screenshot of a browser window showing a Stackblitz editor and preview. The editor shows two files: `popup-window.component.html` and `app.component.html`. The `app.component.html` file contains:

```
<p>
  <button (click)="showPopup = true">Show pop-up</button>

  <app-popup-window
    [isOpen]="showPopup"
    title="Test pop-up"
    (onClose)="popupClosed($event)"
  >
    This is the content
  </app-popup-window>
</p>
```

The preview shows a button labeled "Show pop-up". When clicked, a modal window titled "Test pop-up" appears with the content "This is the content".

A screenshot of a browser window showing a Stackblitz editor and preview. The editor shows two files: `popup-window.component.html` and `app.component.html`. The `app.component.html` file contains:

```
<div class="modalW" *ngIf="isOpen">
  <!-- Modal content -->
  <div class="modalW-content">
    <div class="modalW-header">
      <span class="close" (click)="closePopup()">&times;</span>
      <h2>{{ title }}</h2>
    </div>
    <div class="modalW-body">
      <ng-content></ng-content>
    </div>
  </div>
</div>
```

The preview shows a button labeled "Show pop-up". When clicked, a modal window titled "Test pop-up" appears with the content "This is the content".

## Option #2 Multi-slot content projection

But use-cases are more complex than the above scenario and most of the time we need to pass more than one thing to the child component but many different things like a body, a header, a footer, etc. We can use **multi-slot** content projection using multiple `ng-content` as slots as below.

## What is multi-slot content projection?

- It's a technique to pass **more than one piece of content** to the child component. In our example, now we want to customize both the body of the dialog and its footer:



## How to do multi-slot content projection?

- We need multiple `<ng-content>` that each have their own custom CSS selector to pick the right template from the HTML content projected by the parent component:

```
<ng-content select="[attributeSelector]"></ng-content>
<ng-content select=".cssClassSelector"></ng-content>
<ng-content select="div.content[first]"></ng-content>
```

- Then the parent component can customize the projected content accordingly:

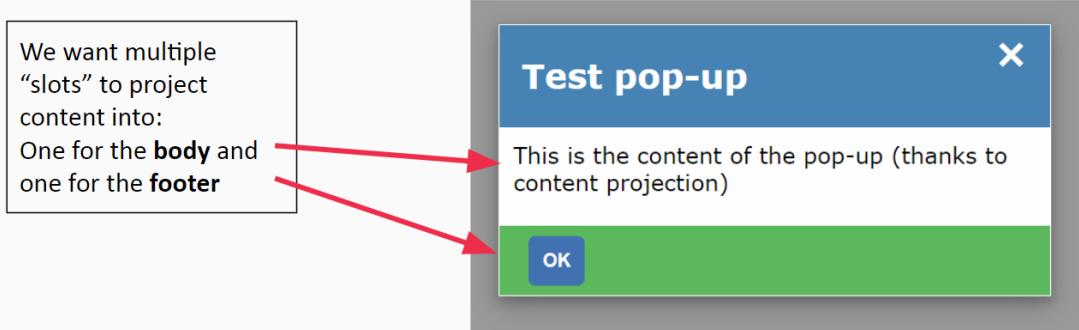
```
<p attributeSelector>I'll go in the first ng-content</p>
<div class="cssClassSelector">Me, second ng-content</div>
<div class="content" first>And I'm the third</div>
```

Each `ng-content` will have a query/selector that helps it decide which element to get from the parent component based on the CSS selector we pass as a parameter.

## Exercise #1 - Multi-slot content projection



- Let's change the code at <https://stackblitz.com/edit/at-content-projection> to make our pop-up window have two different slots:



We pass multiple elements from the parent component:

```
<app-popup-window [isOpen] = "showPopup" title = "Test pop-up" (onClose) = "popupClosed($event)">
  <div body>
    This is the content of the pop-up (thanks to content projection)
  </div>
  <div footer>
    <button (click) = "showPopup = false">OK</button>
  </div>
</app-popup-window>
```

[app.component.html](#)

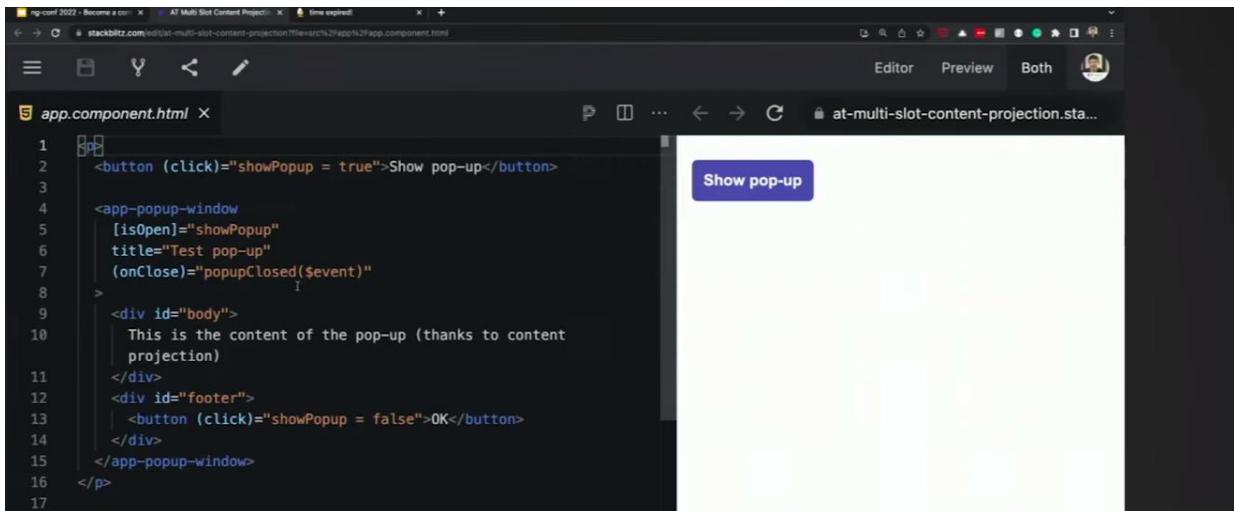
Such content gets projected using CSS selector queries:

```
<div class = "modalW-body">
  <ng-content select = "[body]"></ng-content>
</div>
<div class = "modalW-footer">
  <ng-content select = "[footer]"></ng-content>
</div>
```

[popup-window.component.html](#)

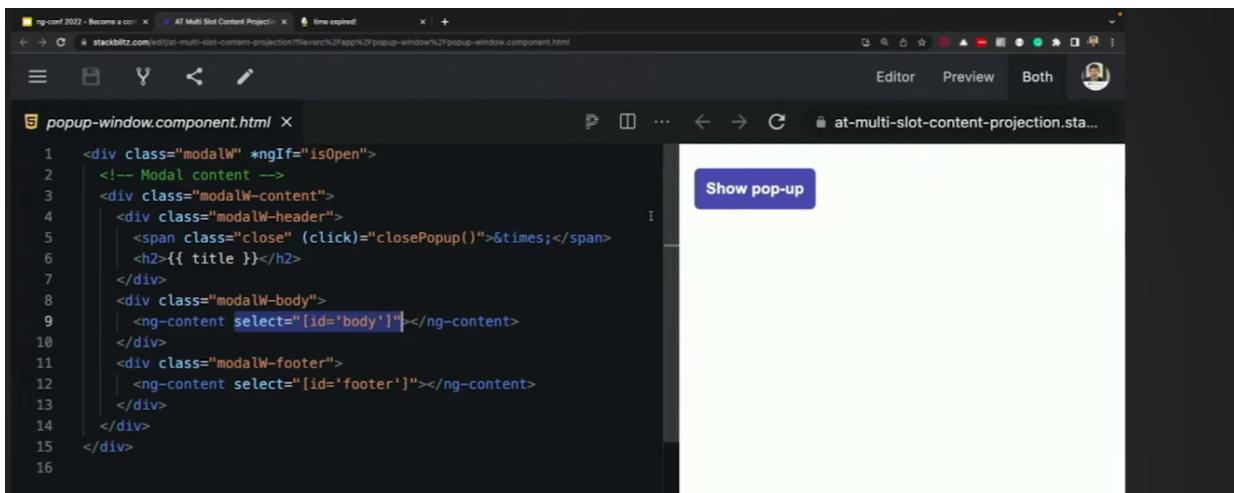


In the parent component, we now have 2 different blocks that we want to project into the child component. We then have 2 ng-content in the child component that gets the 2 blocks based on the selectors.



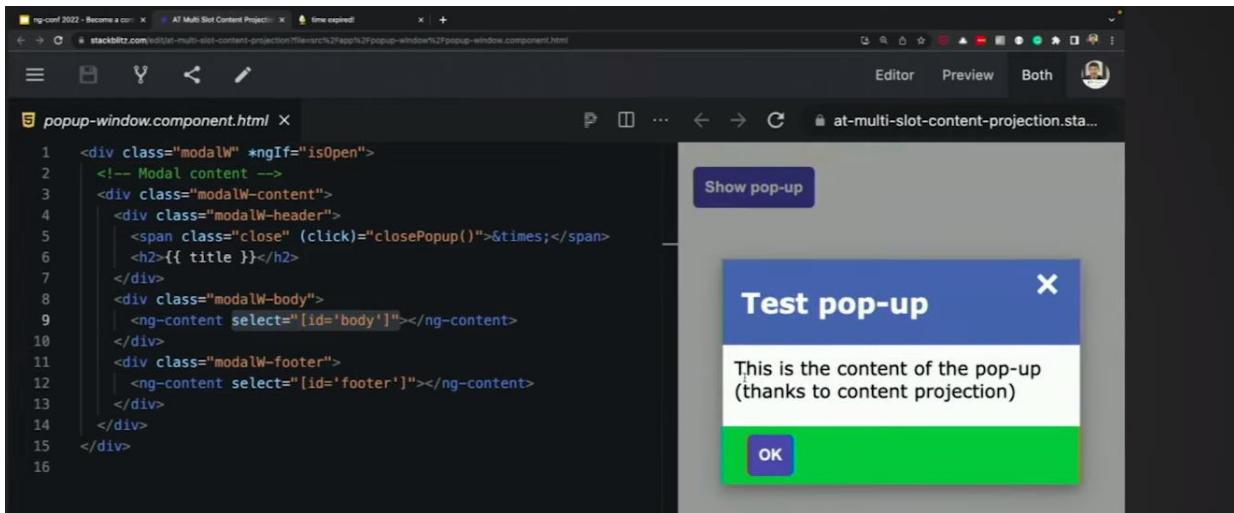
app.component.html

```
1 <button (click)="showPopup = true">Show pop-up</button>
2
3 <app-popup-window
4   [isOpen]="showPopup"
5   title="Test pop-up"
6   (onClose)="popupClosed($event)">
7   >
8     <div id="body">
9       This is the content of the pop-up (thanks to content
10      projection)
11    </div>
12    <div id="footer">
13      <button (click)="showPopup = false">OK</button>
14    </div>
15  </app-popup-window>
16 </p>
17
```



popup-window.component.html

```
1 <div class="modalW" *ngIf="isOpen">
2   <!-- Modal content -->
3   <div class="modalW-content">
4     <div class="modalW-header">
5       <span class="close" (click)="closePopup()">&times;</span>
6       <h2>{{ title }}</h2>
7     </div>
8     <div class="modalW-body">
9       <ng-content select="#body"></ng-content>
10    </div>
11    <div class="modalW-footer">
12      <ng-content select="#footer"></ng-content>
13    </div>
14  </div>
15 </div>
16
```



popup-window.component.html

```
1 <div class="modalW" *ngIf="isOpen">
2   <!-- Modal content -->
3   <div class="modalW-content">
4     <div class="modalW-header">
5       <span class="close" (click)="closePopup()">&times;</span>
6       <h2>{{ title }}</h2>
7     </div>
8     <div class="modalW-body">
9       <ng-content select="#body"></ng-content>
10    </div>
11    <div class="modalW-footer">
12      <ng-content select="#footer"></ng-content>
13    </div>
14  </div>
15 </div>
16
```

# Option #3

## Dynamic content projection with ng-template

Multi-slot content projection above works but you have to know upfront the number of blocks you want to project, it is not dynamic in nature. For that, we can use the ng-template approach here.

### Dynamic content projection with any number of items

- Multi-slot content projection is static: We cannot have a dynamic query to project a variable number of elements.
- In this new example, we work on a generic **tabs** component that can render any number of tabs, each tab displaying projected content:



### Using ng-container and ng-template

- **<ng-container>** is a placeholder element that can be used to render a template based on the component logic. We pass a reference to that template using **ngTemplateOutlet**:

```
<ng-container [ngTemplateOutlet]="template"> </ng-container>
```

- **<ng-template>** is an element containing an HTML template to render in a container:

```
<ng-template tab>Content for first tab</ng-template>
```

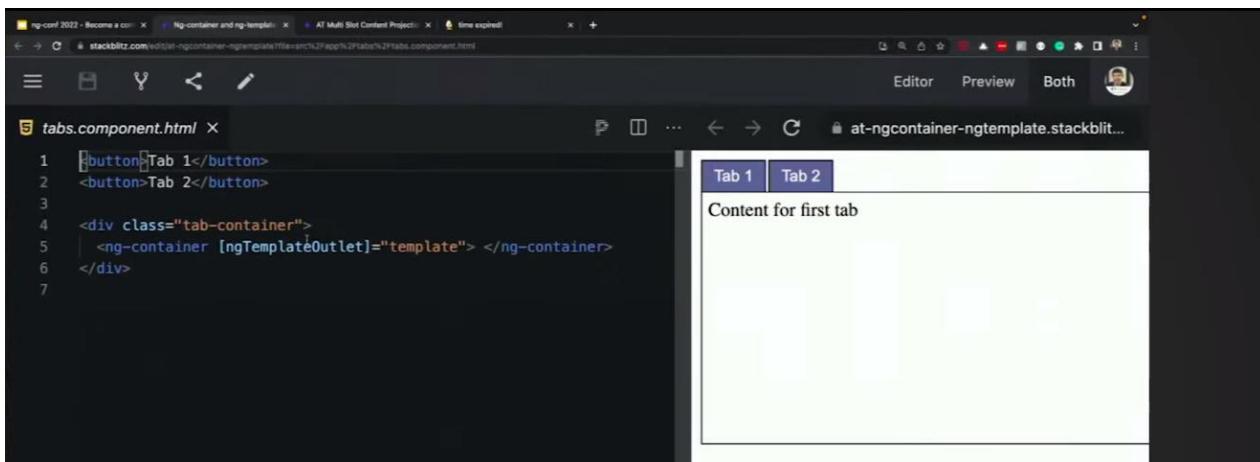
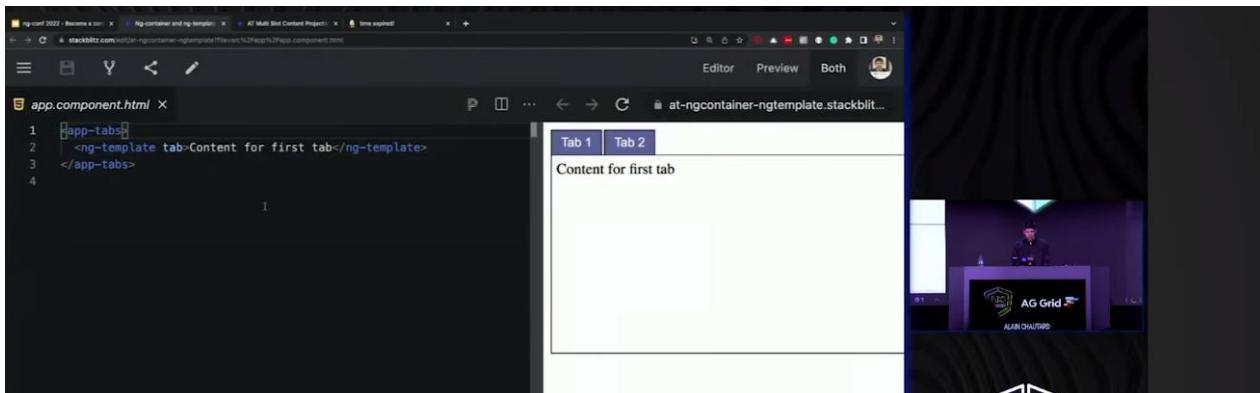
By default, an **>ng-template** itself is not visible or rendered in the DOM, it is only rendered when angular needs it to display in a container.

## Exercise #2 - ng-container, ng-template, and more!



- Let's look at and understand the code at  
<https://stackblitz.com/edit/at-ngcontainer-ngtemplate>

Then we want to change that code to make our **tabs** component fully dynamic and generic:



We pass multiple templates from the parent component:

```
<app-tabs [names]=["Tab 1", "Tab 2", "Tab 3"]>
  <ng-template tab>Content for first tab</ng-template>
  <ng-template tab>Content for 2nd tab</ng-template>
  <ng-template tab>Content for 3rd tab</ng-template>
</app-tabs>
```

[app.component.html](#)

**Tab** is a marker directive used for querying these templates:

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

@Directive({
  selector: '[tab]',
})
export class TabDirective {}
```

[tab.directive.ts](#)

Templates are retrieved with **@ContentChildren** using the **TabDirective** to select the right elements:

```
@Component({
  selector: 'app-tabs',
  templateUrl: './tabs.component.html',
  styleUrls: ['./tabs.component.css'],
})
export class TabsComponent {

  index = 0;

  @Input()
  names: string[] = [];

  @ContentChildren(TabDirective, { read: TemplateRef })
  templates: QueryList<any>;
```

[**currentTab: TemplateRef<any>;**]



[tabs.components.ts](#)

The **@ContentChildren** directive will query the content to find all our templates as a **QueryList**.

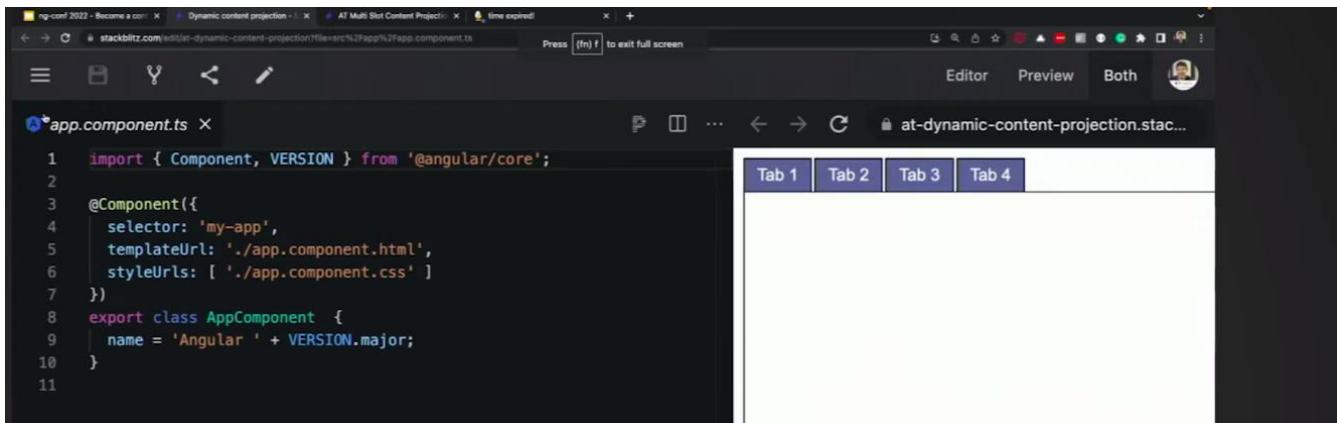
When a tab is clicked, we switch the current template using our template list:

```
<button *ngFor="let name of names; index as i"
  (click)="currentTab = templates.get(i)">
  {{ name }}
</button>

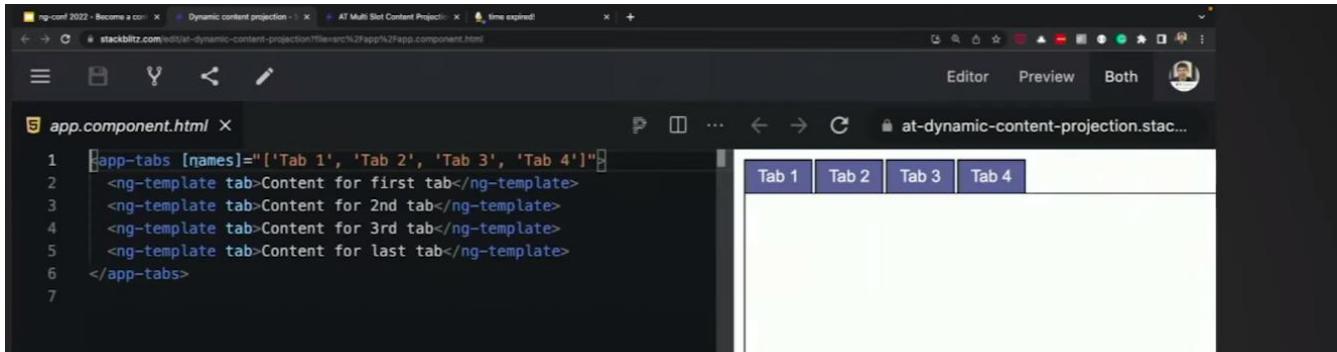
<div class="tab-container">
  <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
</div>
```

[tabs.component.html](#)

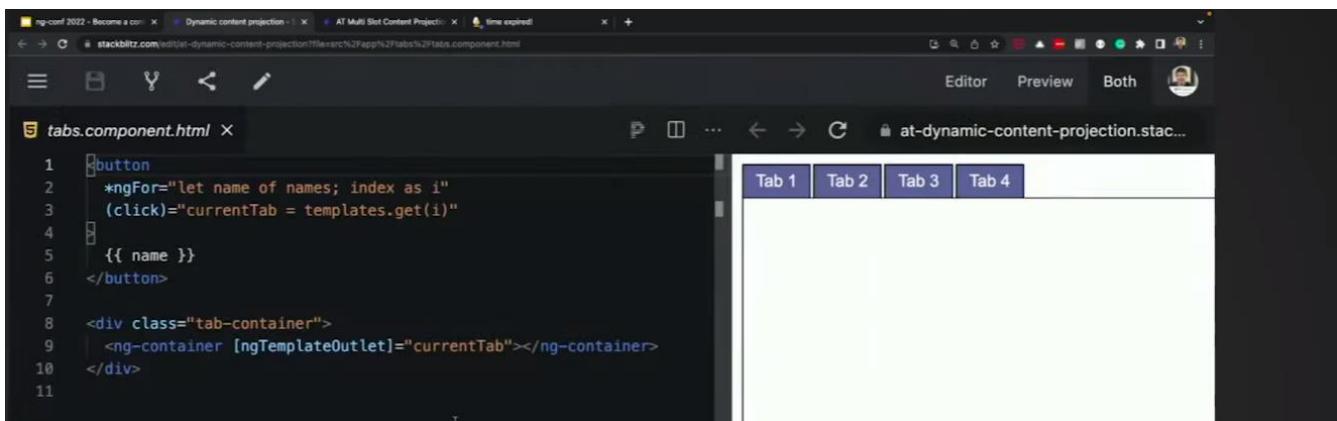




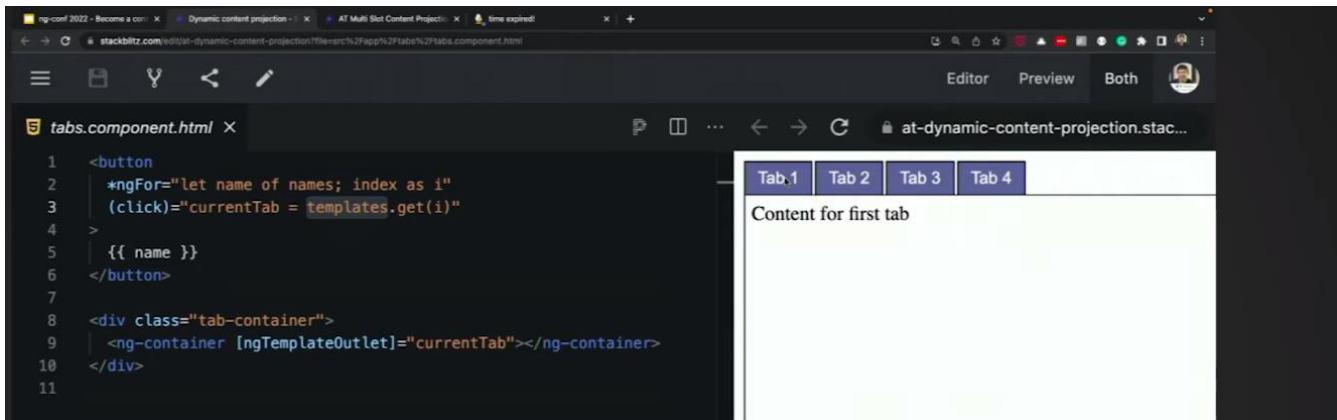
```
1 import { Component, VERSION } from '@angular/core';
2
3 @Component({
4   selector: 'my-app',
5   templateUrl: './app.component.html',
6   styleUrls: [ './app.component.css' ]
7 })
8 export class AppComponent {
9   name = 'Angular ' + VERSION.major;
10 }
11
```



```
1 <app-tabs [names]=["Tab 1", "Tab 2", "Tab 3", "Tab 4"]>
2   <ng-template tab>Content for first tab</ng-template>
3   <ng-template tab>Content for 2nd tab</ng-template>
4   <ng-template tab>Content for 3rd tab</ng-template>
5   <ng-template tab>Content for last tab</ng-template>
6 </app-tabs>
7
```



```
1 <button
2   *ngFor="let name of names; index as i"
3   (click)="currentTab = templates.get(i)"
4 >
5   {{ name }}
6 </button>
7
8 <div class="tab-container">
9   <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
11
```



```
1 <button
2   *ngFor="let name of names; index as i"
3   (click)="currentTab = templates.get(i)"
4 >
5   {{ name }}
6 </button>
7
8 <div class="tab-container">
9   <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
11
```

Clicking in the Tab 1 means that it will get the content for the first ng-template and set the value inside the container for display.

The screenshot shows a browser window with two tabs open. The left tab is titled 'tabs.component.html' and contains the following code:

```
1  <button
2  | *ngFor="let name of names; index as i"
3  | (click)="currentTab = templates.get(i)"
4  |
5  | {{ name }}
6  </button>
7
8  <div class="tab-container">
9  | <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
```

The right tab is titled 'at-dynamic-content-projection.stackblitz.com' and shows the preview of the component. It features a tab bar with four tabs labeled 'Tab 1', 'Tab 2', 'Tab 3', and 'Tab 4'. Below the tabs, the content for 'Tab 2' is displayed, which is 'Content for 2nd tab'.

The screenshot shows a browser window with two tabs open. The left tab is titled 'tabs.component.html' and contains the same code as the previous screenshot.

```
1  <button
2  | *ngFor="let name of names; index as i"
3  | (click)="currentTab = templates.get(i)"
4  |
5  | {{ name }}
6  </button>
7
8  <div class="tab-container">
9  | <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
```

The right tab is titled 'at-dynamic-content-projection.stackblitz.com' and shows the preview of the component. It features a tab bar with four tabs labeled 'Tab 1', 'Tab 2', 'Tab 3', and 'Tab 4'. Below the tabs, the content for 'Tab 3' is displayed, which is 'Content for 3rd tab'.

The screenshot shows a browser window with two tabs open. The left tab is titled 'tabs.component.html' and contains the same code as the previous screenshots.

```
1  <button
2  | *ngFor="let name of names; index as i"
3  | (click)="currentTab = templates.get(i)"
4  |
5  | {{ name }}
6  </button>
7
8  <div class="tab-container">
9  | <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
```

The right tab is titled 'at-dynamic-content-projection.stackblitz.com' and shows the preview of the component. It features a tab bar with four tabs labeled 'Tab 1', 'Tab 2', 'Tab 3', and 'Tab 4'. Below the tabs, the content for 'Tab 5' is displayed, which is 'Content for last tab'.

The screenshot shows a browser window with two tabs open. The left tab is titled 'app.component.html' and contains the following code:

```
1  <app-tabs [names]="['Tab 1', 'Tab 2', 'Tab 3', 'Tab 4', 'Tab 5']">
2  | <ng-template tab>Content for first tab</ng-template>
3  | <ng-template tab>Content for 2nd tab</ng-template>
4  | <ng-template tab>Content for 3rd tab</ng-template>
5  | <ng-template tab>Content for 4th tab</ng-template>
6  | <ng-template tab>Content for last tab</ng-template>
7  </app-tabs>
```

The right tab is titled 'at-dynamic-content-projection.stackblitz.com' and shows the preview of the component. It features a tab bar with five tabs labeled 'Tab 1', 'Tab 2', 'Tab 3', 'Tab 4', and 'Tab 5'. Below the tabs, the content for 'Tab 4' is displayed, which is 'Content for 4th tab'.

The number of titles in the **names** list needs to match the number of ng-templates

# Option #4

## Syntax improvements using Angular tricks

### Syntax improvements using Angular tricks

- Since we're using a marker directive as a way to query the templates to be used in our content projection, why not use an `@Input` with that directive to pass the name of the tab as a parameter?

Before:

```
<app-tabs [names]="['Tab 1', 'Tab 2', 'Tab 3', 'Tab 4']">
  <ng-template tab>Content for first tab</ng-template>
  <ng-template tab>Content for 2nd tab</ng-template>
  <ng-template tab>Content for 3rd tab</ng-template>
</app-tabs>
```

After:

```
<app-tabs>
  <ng-template tab name="First tab">Content for first tab</ng-template>
  <ng-template tab name="Second tab">Content for 2nd tab</ng-template>
  <ng-template tab name="Third tab">Content for 3rd tab</ng-template>
</app-tabs>
```

Note that we do not have to pass all the tab names as an input, we can pass the name directly on the ng-template.

### Updated code:

```
@Directive({
  selector: '[tab]',
})
export class TabDirective {
  @Input()
  name: string;
}
```

[tab.directive.ts](#)

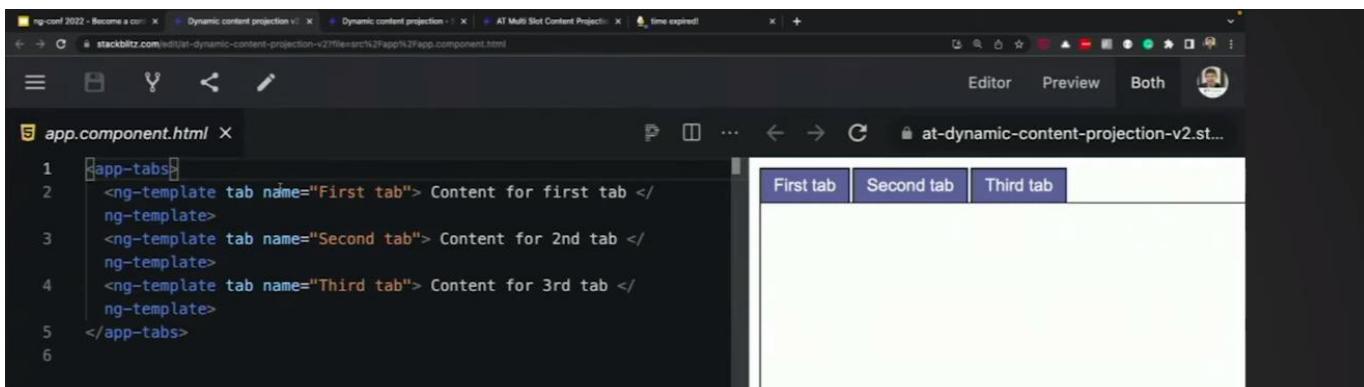
```
<button
  *ngFor="let tab of tabs; index as i"
  (click)="currentTab = templates.get(i)">
  {{ tab.name }}
</button>
```

[tabs.component.html](#)

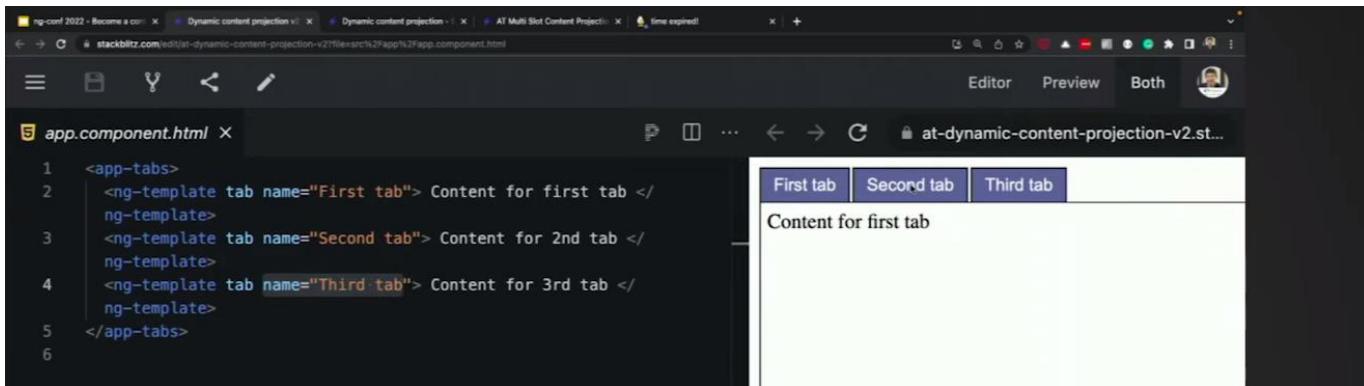
```
export class TabsComponent {
  @ContentChildren(TabDirective, { read: TemplateRef })
  templates: QueryList<TemplateRef<any>>;
  @ContentChildren(TabDirective)
  tabs: QueryList<TabDirective>;
  currentTab: TemplateRef<any>;
}
```

[tabs.component.ts](#)

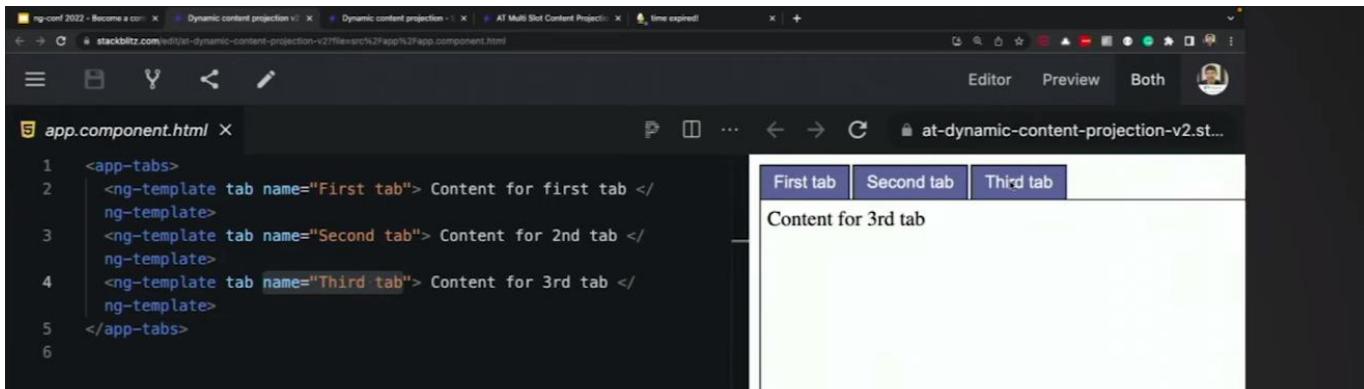




```
app.component.html
1 <app-tabs>
2   <ng-template tab name="First tab"> Content for first tab </ng-template>
3   <ng-template tab name="Second tab"> Content for 2nd tab </ng-template>
4   <ng-template tab name="Third tab"> Content for 3rd tab </ng-template>
5 </app-tabs>
```



```
app.component.html
1 <app-tabs>
2   <ng-template tab name="First tab"> Content for first tab </ng-template>
3   <ng-template tab name="Second tab"> Content for 2nd tab </ng-template>
4   <ng-template tab name="Third tab"> Content for 3rd tab </ng-template>
5 </app-tabs>
```



```
app.component.html
1 <app-tabs>
2   <ng-template tab name="First tab"> Content for first tab </ng-template>
3   <ng-template tab name="Second tab"> Content for 2nd tab </ng-template>
4   <ng-template tab name="Third tab"> Content for 3rd tab </ng-template>
5 </app-tabs>
```

Let's simplify even further!

- What if we removed **ng-template** from our code?
- We could remove one **@ContentChildren** query in the tabs component
- We could also “tie” the **@Input** more explicitly to the directive:

**Before:**

```
<app-tabs>
  <ng-template tab name="First tab">Content for first tab</ng-template>
  <ng-template tab name="Second tab">Content for 2nd tab</ng-template>
  <ng-template tab name="Third tab">Content for 3rd tab</ng-template>
</app-tabs>
```

**After:**

```
<app-tabs>
  <div *tab="First tab">Content for first tab</div>
  <div *tab="Second tab">Content for 2nd tab</div>
  <div *tab="Third tab">Content for 3rd tab</div>
</app-tabs>
```

## Shorthand vs. Expanded syntax for template-based directives:

Shorthand syntax:

```
<div *ngIf="condition">Content to render</div>
```



Expanded syntax:

```
<ng-template [ngIf]="condition">  
    <div>Content to render</div>  
</ng-template>
```



The two syntax above are actually the same.

### Exercise #3 - Let's improve the tabs component API



- Start from:  
<https://stackblitz.com/edit/at-dynamic-content-projection-v2>
- Remove **ng-template** from our code
- Remove one **@ContentChildren** query in the tabs component
- Try to “tie” the **@Input** more explicitly to the directive:

**Before:**

```
<app-tabs>  
    <ng-template tab name="First tab">Content for first tab</ng-template>  
    <ng-template tab name="Second tab">Content for 2nd tab</ng-template>  
    <ng-template tab name="Third tab">Content for 3rd tab</ng-template>  
</app-tabs>
```

**After:**

```
<app-tabs>  
    <div *tab="'First tab'">Content for first tab</div>  
    <div *tab="'Second tab'">Content for 2nd tab</div>  
    <div *tab="'Third tab'">Content for 3rd tab</div>  
</app-tabs>
```

## Updated code:

```
@Directive({
  selector: '[tab]',
})
export class TabDirective {
  @Input('tab')
  name: string;

  constructor(public template: TemplateRef<any>) {}
}
```

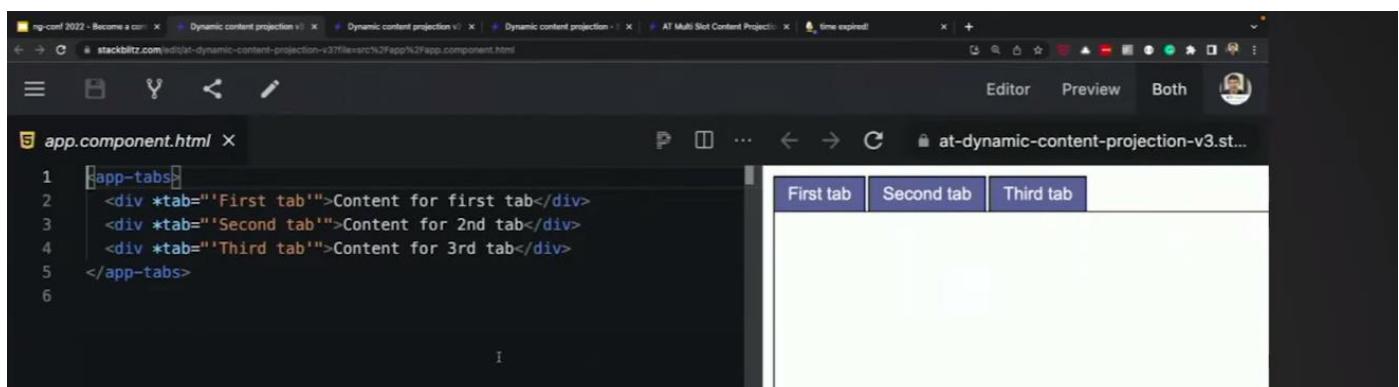
 tab.directive.ts

```
export class TabsComponent {
  @ContentChildren(TabDirective)
  tabs: QueryList<TabDirective>;
  currentTab: TemplateRef<any>;
}
```

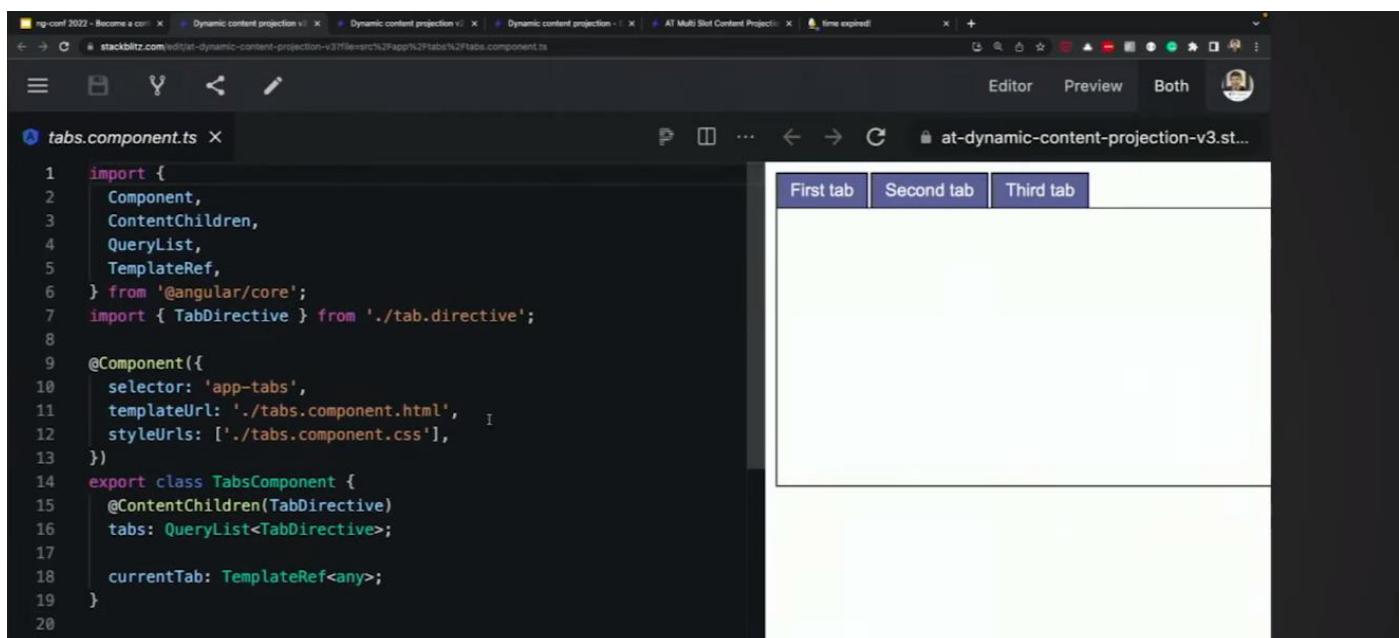
[tabs.component.ts](#)

```
<button *ngFor="let tab of tabs; index as i"
         (click)="currentTab = tab.template">
  {{ tab.name }}
</button>
```

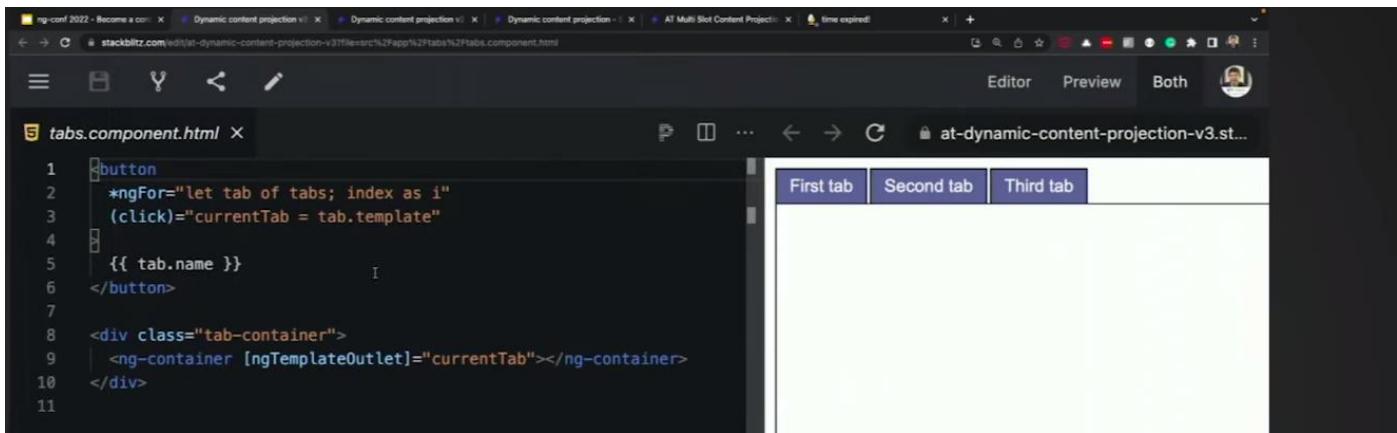
[tabs.component.html](#)



The screenshot shows a browser window with three tabs labeled "First tab", "Second tab", and "Third tab". The tabs are displayed horizontally at the top of the screen. The content area below the tabs is currently empty, indicating that no content has been dynamically projected into it yet.

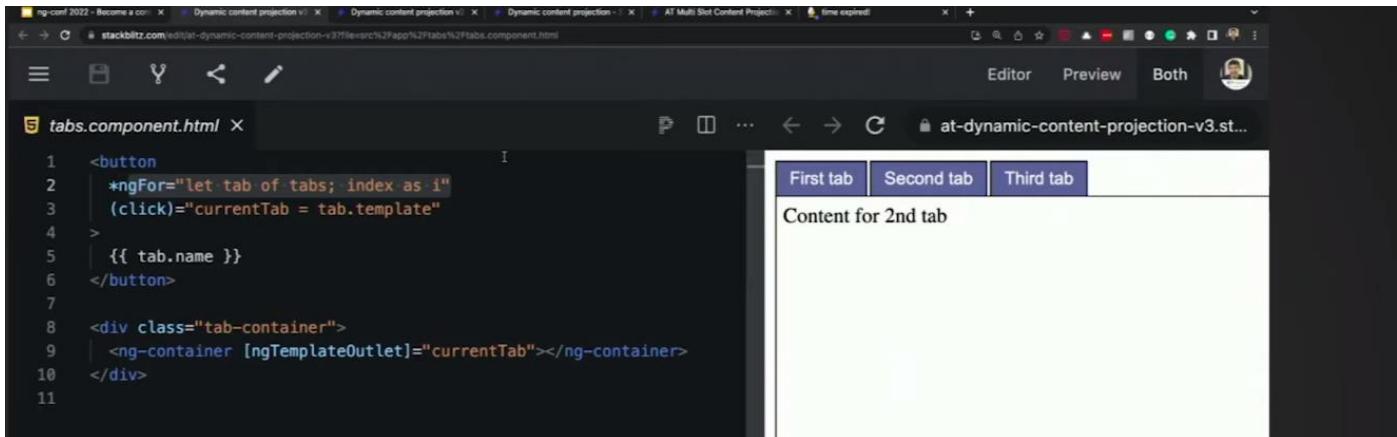


The screenshot shows a browser window with three tabs labeled "First tab", "Second tab", and "Third tab". The tabs are displayed horizontally at the top of the screen. The content area below the tabs is currently empty, indicating that no content has been dynamically projected into it yet.

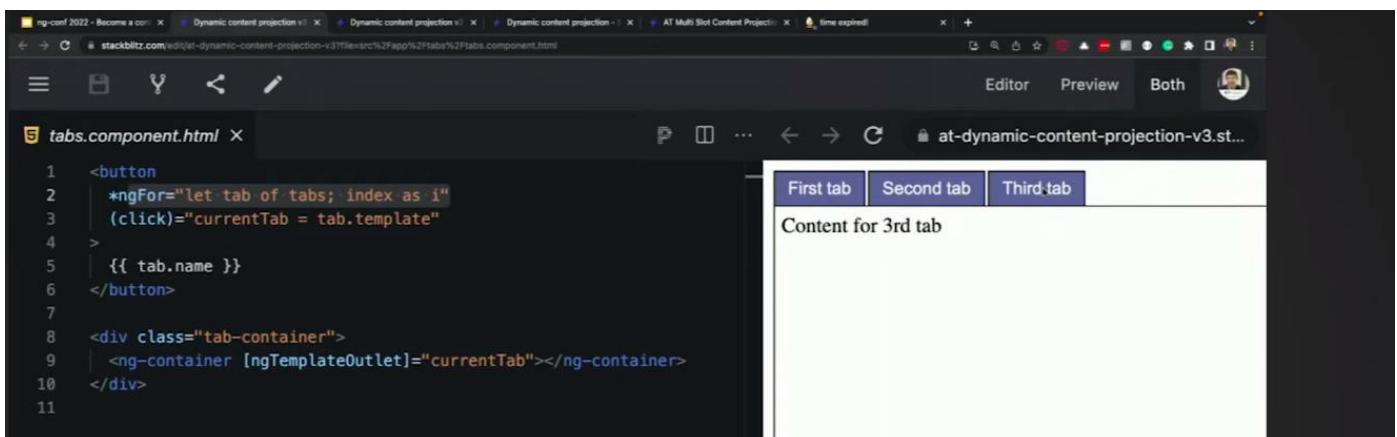


tabs.component.html

```
1 <button
2   *ngFor="let tab of tabs; index as i"
3     (click)="currentTab = tab.template"
4   >
5     {{ tab.name }}
6   </button>
7
8 <div class="tab-container">
9   <ng-container [ngTemplateOutlet]="currentTab"></ng-container>
10 </div>
```



Content for 2nd tab

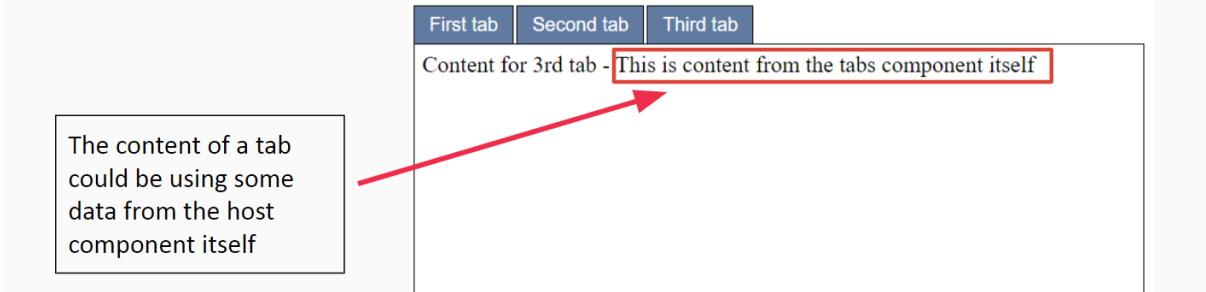


Content for 3rd tab

## Option #5 Accessing context from the host component

## Accessing context from the host component

- Sometimes we want our component to be able to use context from its own class properties, displayed within the projected template!



### Host component context passed to projected template:

app.component.html

```
<app-tabs>
  <div *tab="'First tab'">Content for first tab</div>
  <div *tab="'Second tab'">Content for 2nd tab</div>
  <div *tab="'Third tab'; title as title">
    Content for 3rd tab - {{ title }}
  </div>
```

tabs.component.ts

```
export class TabsComponent {
  @ContentChildren(TabDirective)
  tabs: QueryList<TabDirective>;

  ctx = { title: 'This is some context' };

  currentTab: TemplateRef<any>;
}
```

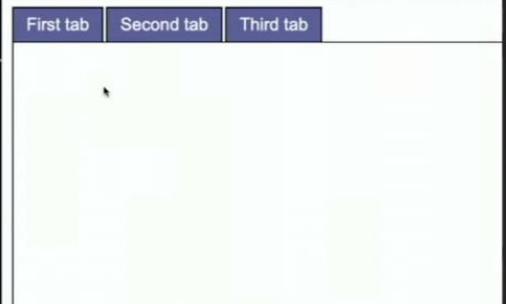
tabs.component.html

```
<div class="tab-container">
  <ng-container *ngTemplateOutlet="currentTab; context: ctx">
  </ng-container>
</div>
```

A blue button labeled 'DEMO' with a play icon is shown, indicating a live demonstration of the code.

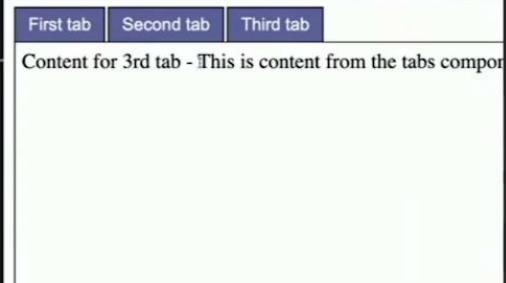
The screenshot shows the Angular IDE with the file `app.component.ts` open. The code defines the `AppComponent` with a selector of `'my-app'`, a template URL of `./app.component.html`, and a style URL of `./app.component.css`. The component has a property `name` set to `'Angular ' + VERSION.major`.

To the right, a browser window displays a tabs component with three tabs: 'First tab', 'Second tab', and 'Third tab'. The 'Third tab' content area is empty, corresponding to the state shown in the IDE.



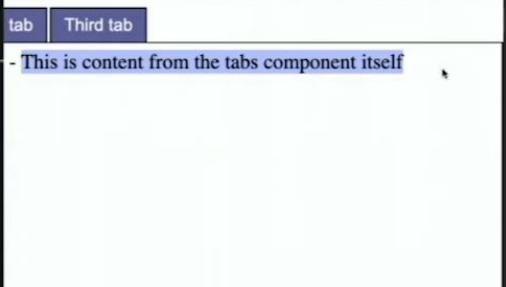
app.component.html

```
1 <app-tabs>
2   <div *tab="First tab">Content for first tab</div>
3   <div *tab="Second tab">Content for 2nd tab</div>
4   <div *tab="Third tab"; title as title>
5     Content for 3rd tab - {{ title }}
6   </div>
7   <!-- <ng-template tab="Third tab" let-title="title">
8     | Content for 3rd tab - {{ title }}
9   </ng-template>
10  -->
11 </app-tabs>
```



app.component.html

```
1 <app-tabs>
2   <div *tab="First tab">Content for first tab</div>
3   <div *tab="Second tab">Content for 2nd tab</div>
4   <div *tab="Third tab"; title as title>
5     Content for 3rd tab - {{ title }}
6   </div>
7   <!-- <ng-template tab="Third tab" let-title="title">
8     | Content for 3rd tab - {{ title }}
9   </ng-template>
10  -->
11 </app-tabs>
```



app.component.html

```
1 <app-tabs>
2   <div *tab="First tab">Content for first tab</div>
3   <div *tab="Second tab">Content for 2nd tab</div>
4   <div *tab="Third tab"; title as title>
5     Content for 3rd tab - {{ title }}
6   </div>
7   <!-- <ng-template tab="Third tab" let-title="title">
8     | Content for 3rd tab - {{ title }}
9   </ng-template>
10  -->
11 </app-tabs>
```

The screenshot shows a browser window with multiple tabs open, including 'ng-conf 2022 - Become a cert...', 'Dynamic content projection ...', 'Dynamic content projection ...', 'Dynamic content projection ...', 'Dynamic content projection ...', 'AT Multi Slot Content Projec...', and 'Time expired'. The active tab is 'tabs.component.ts'.

The code in 'tabs.component.ts' is as follows:

```
1 import { Component, ContentChildren, QueryList, TemplateRef } from '@angular/core';
2 import { TabDirective } from './tab.directive';
3
4 @Component({
5   selector: 'app-tabs',
6   templateUrl: './tabs.component.html',
7   styleUrls: ['./tabs.component.css'],
8 })
9 export class TabsComponent {
10   @ContentChildren(TabDirective)
11   tabs: QueryList<TabDirective>;
12
13   ctx = { title: 'This is content from the tabs component itself' };
14
15   tabBar: TemplateRef<any>;
16 }
17
18
19
20
21 }
```

The preview on the right shows a tabs component with three tabs: 'First tab', 'Second tab', and 'Third tab'. The 'Third tab' is selected, displaying the content: 'Content for 3rd tab - This is content from the tabs component itself'.

## Conclusion

- We have many features and options available when it comes down to content projection with Angular
- All of them work and can make sense: It all depends on your project size and complexity
- These tools and techniques are widely used by component libraries such as Angular Material

Thanks for your  
attention

Slides:  
<https://bit.ly/at-cp-workshop>

Email: [al@interstate21.com](mailto:al@interstate21.com)

Twitter: [@AlainChautard](#)

<https://blog.angulartraining.com>



# Q&A

**Slides:**  
<https://bit.ly/at-cp-workshop>

**Email:** [al@interstate21.com](mailto:al@interstate21.com)

**Twitter:** @AlainChautard

<https://blog.angulartraining.com>



Questions about React, Angular, Cypress,  
or web development in general?

Book a FREE 30-minute call with me:

<https://bit.ly/al-free-consultation>