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)

Experts 2022

Google Developer Expert in Web technologies

/ Angular / Google Maps

Angular JS addict since 2011

Web consultant / trainer @ 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?

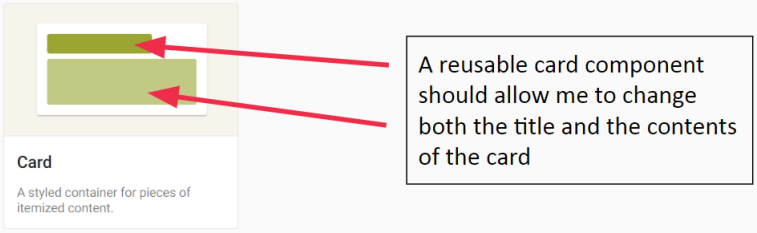
These slides have several links to examples and exercises so here's the link to access them:

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

Why do we need Content Projection?

Why Content Projection?

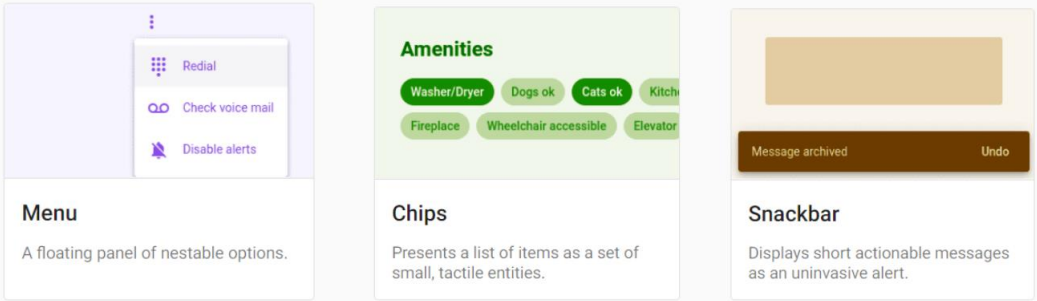
- In order to have truly reusable components, it is important that such components are built in a way that is **as generic as possible**
- This means that such components should be customizable
- The component should not be tied to any business logic



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?

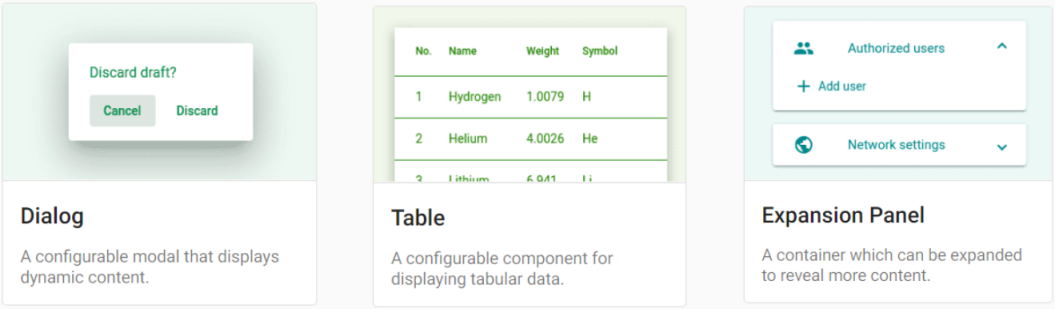
- In many cases, we can achieve component reusability using **inputs** because all we customize is text or icons, or a list of such texts or icons:



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 image…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:

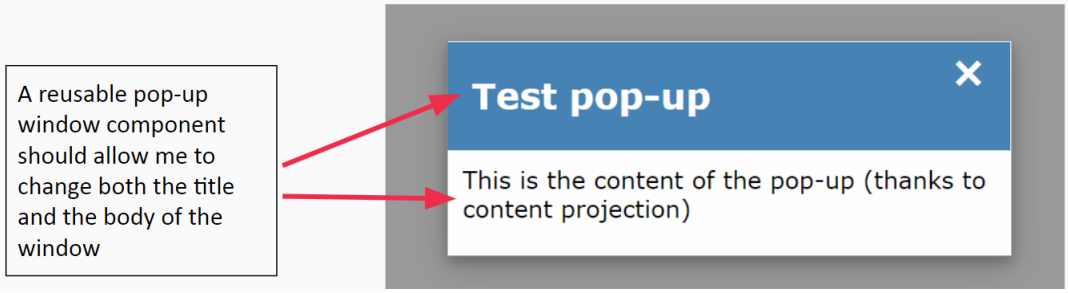


More complex components require more stuff like content projection.

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

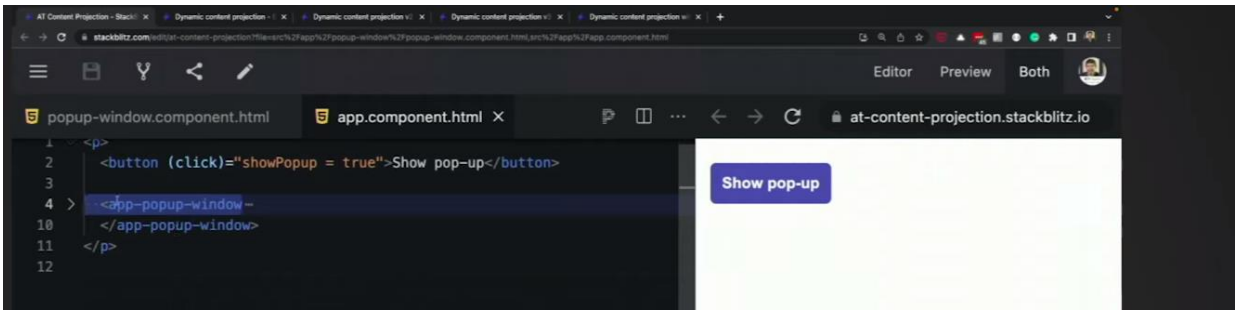
Content gets projected into the template of the component:

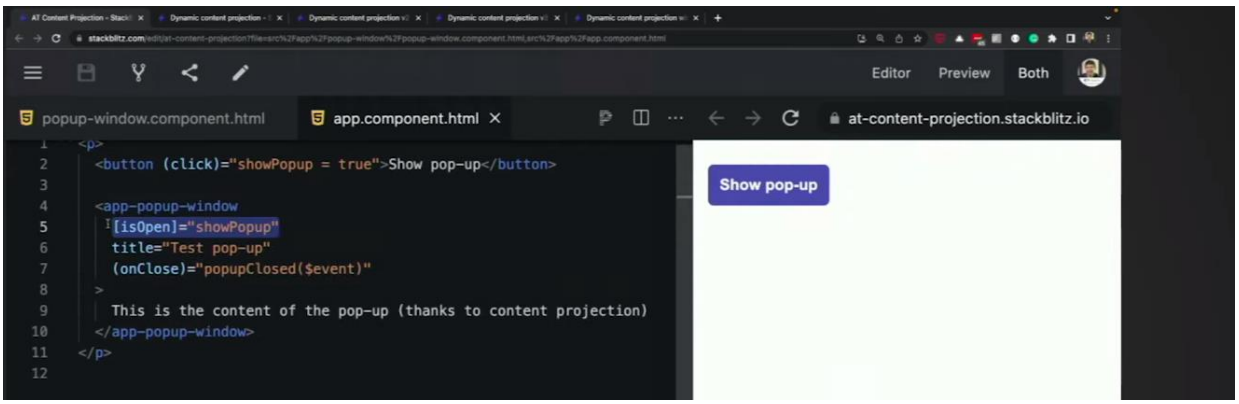
```
<div class="modalW-content">
  <div class="modalW-header">
    <span class="close" (click)="closePopup()">&times;</span>
    <h2>{{ title }}</h2>
  </div>
  <div class="modalW-body">
    <ng-content></ng-content>
  </div>
</div>
```

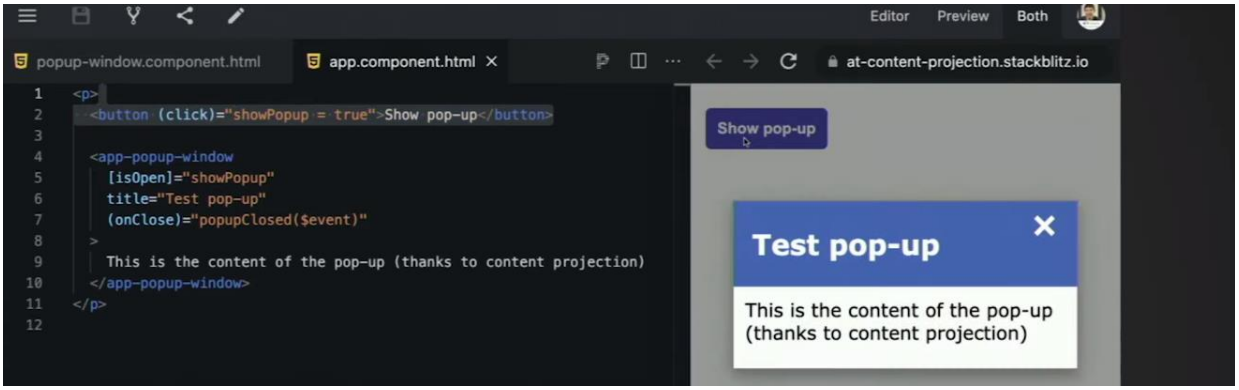
popup-window.component.html

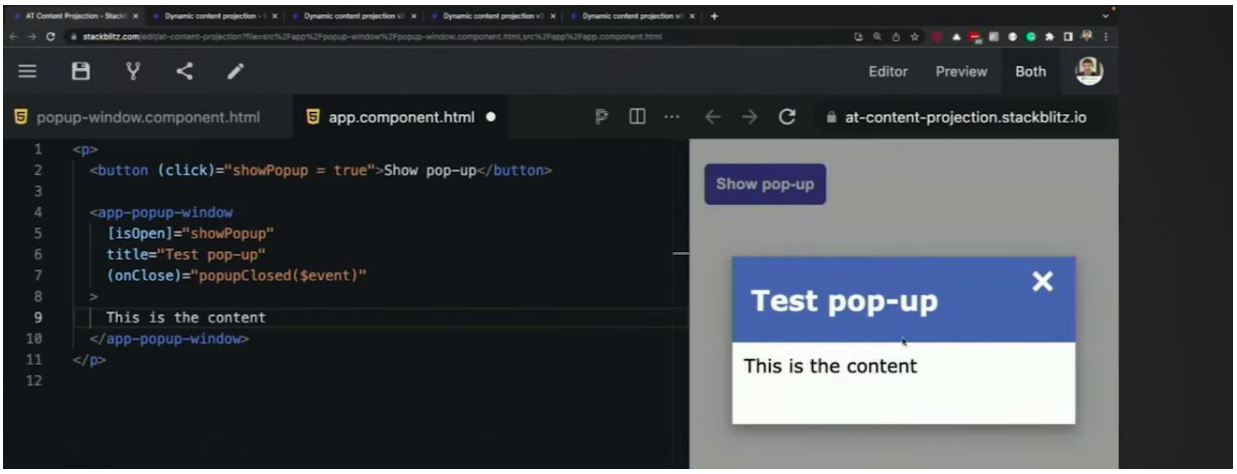
DEMO

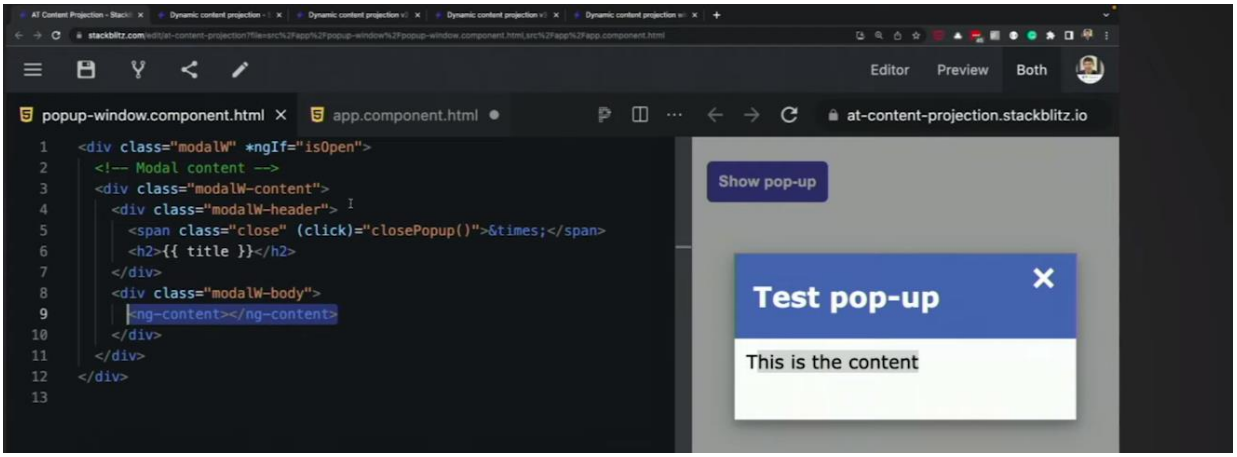
We have a parent component that displays the popup window. We then pass some text (a template or component can also be passed in) in the content section of the component. We then use the ng-content directive to project the template of the child component.









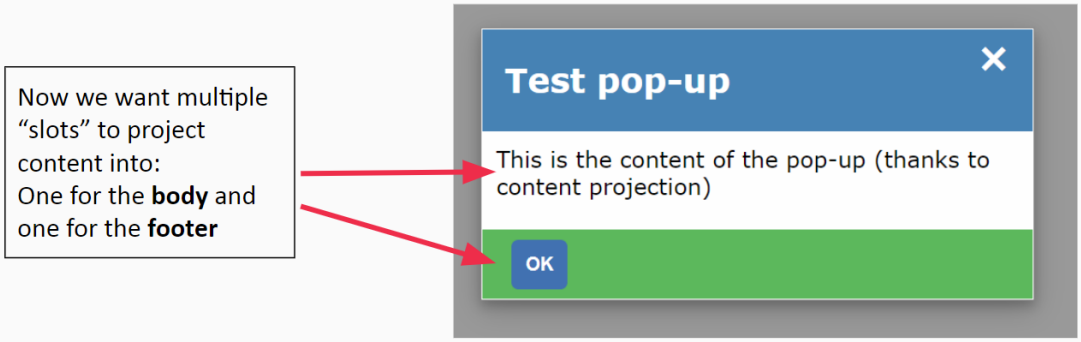


Option #2
Multi-slot content projection

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:

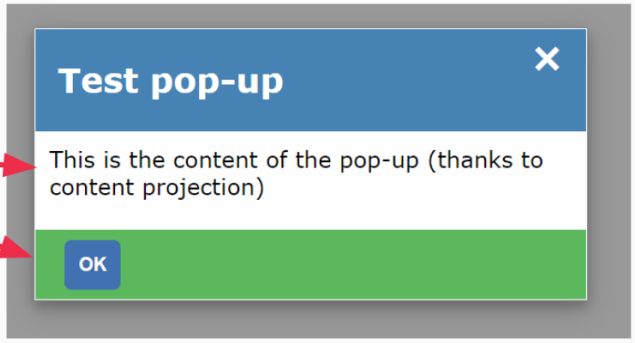
```
<p attributeSelector>I'll go in the first ng-content</p>
<div class="cssClassSelector">Me, second ng-content</div>
<div class="content" first>And I'm the third</div>
```

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 want multiple "slots" to project content into:
One for the **body** and one for the **footer**



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>
```

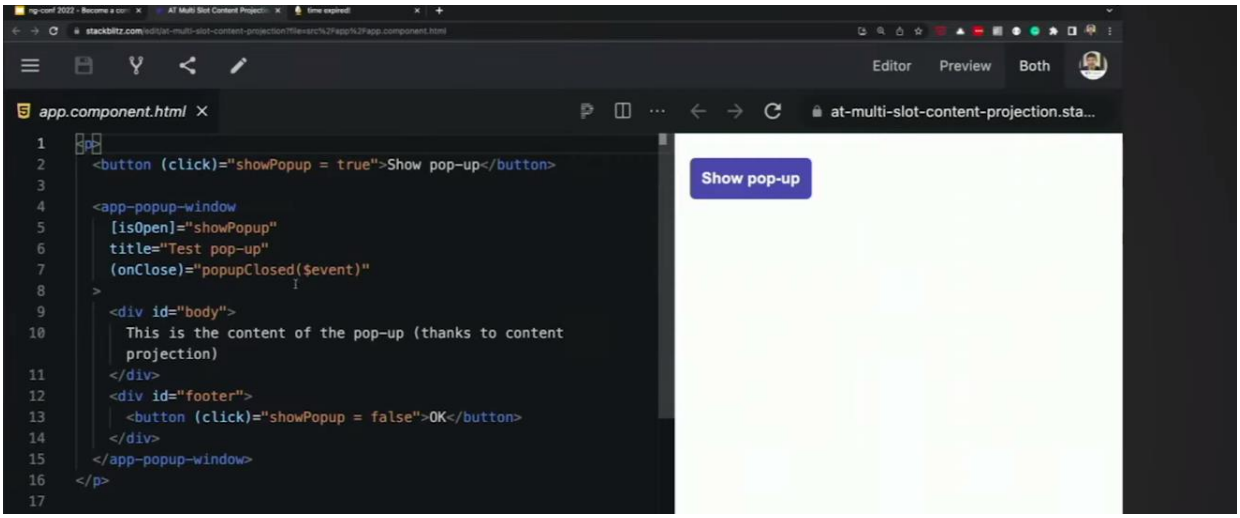
app.component.html

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

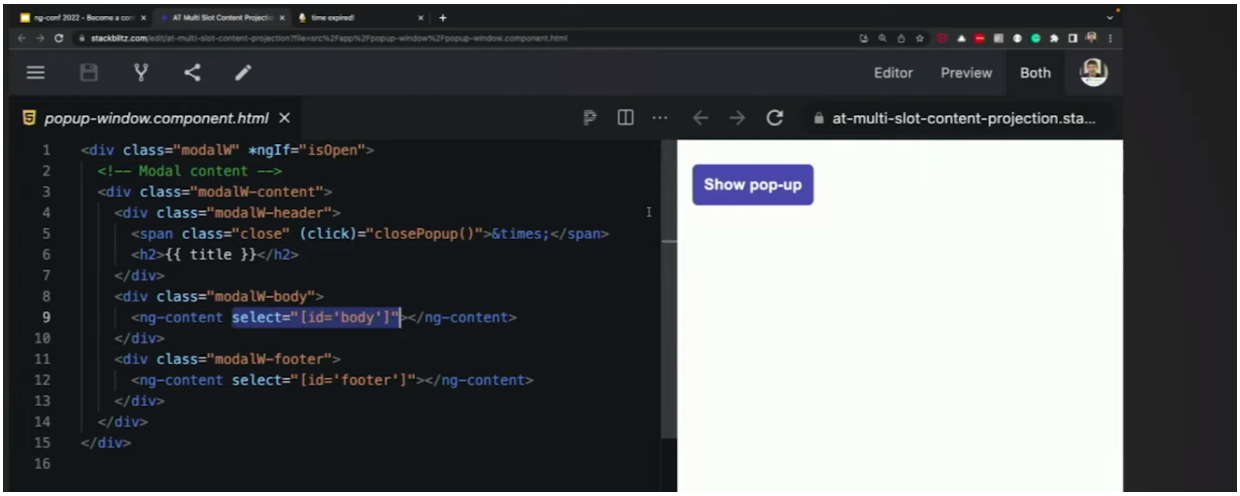
In the parent component, we now have 2 different blocks that we want to project into the child component. We then have 2 ng-content in the child component that gets the 2 blocks based on the selectors.


app.component.html
Editor
Preview
Both
at-multi-slot-content-projection.sta...
Show pop-up
<button (click)="showPopup = true">Show pop-up</button>
<app-popup-window
[isOpen]="showPopup"
title="Test pop-up"
(onClose)="popupClosed($event)"
>

This is the content of the pop-up (thanks to content projection)

<button (click)="showPopup = false">OK</button>

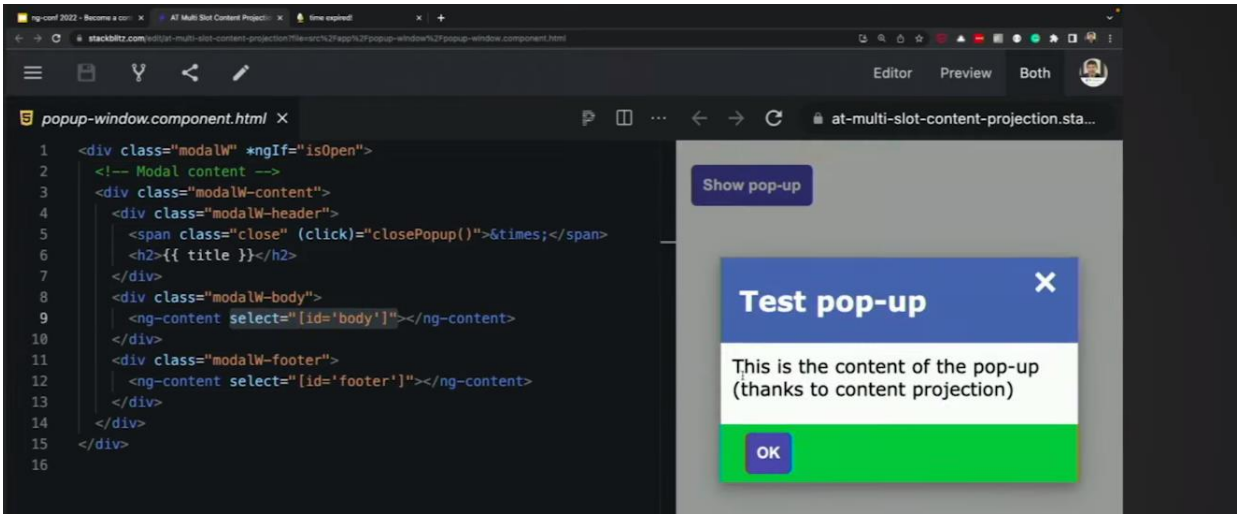
</app-popup-window>
</p>


popup-window.component.html
Editor
Preview
Both
at-multi-slot-content-projection.sta...
Show pop-up
<div class="modalW" *ngIf="isOpen">
<!-- Modal content -->

<span class="close" (click)="closePopup()">×
<h2>{{ title }}</h2>

<ng-content select="[id='body']"></ng-content>

<ng-content select="[id='footer']"></ng-content>


popup-window.component.html
Editor
Preview
Both
at-multi-slot-content-projection.sta...
Show pop-up
Test pop-up
This is the content of the pop-up (thanks to content projection)
OK
<div class="modalW" *ngIf="isOpen">
<!-- Modal content -->

<span class="close" (click)="closePopup()">×
<h2>{{ title }}</h2>

<ng-content select="[id='body']"></ng-content>

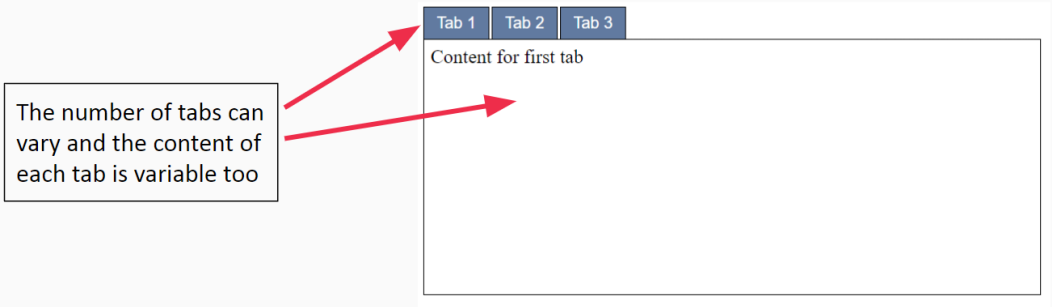
<ng-content select="[id='footer']"></ng-content>

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 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:

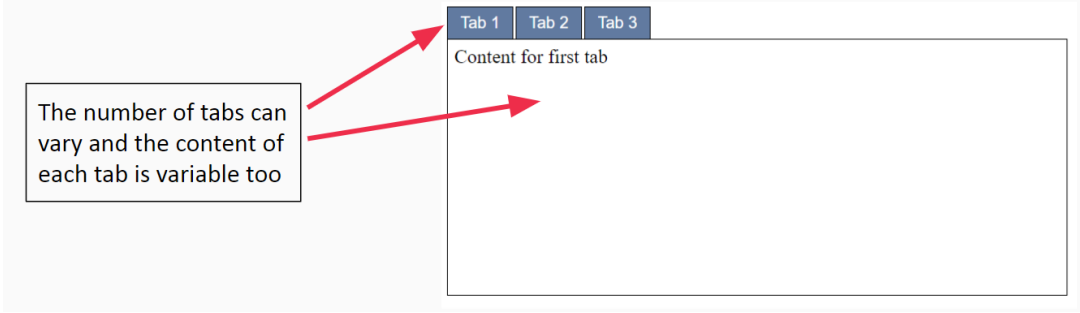
```
<ng-template tab>Content for first tab</ng-template>
```

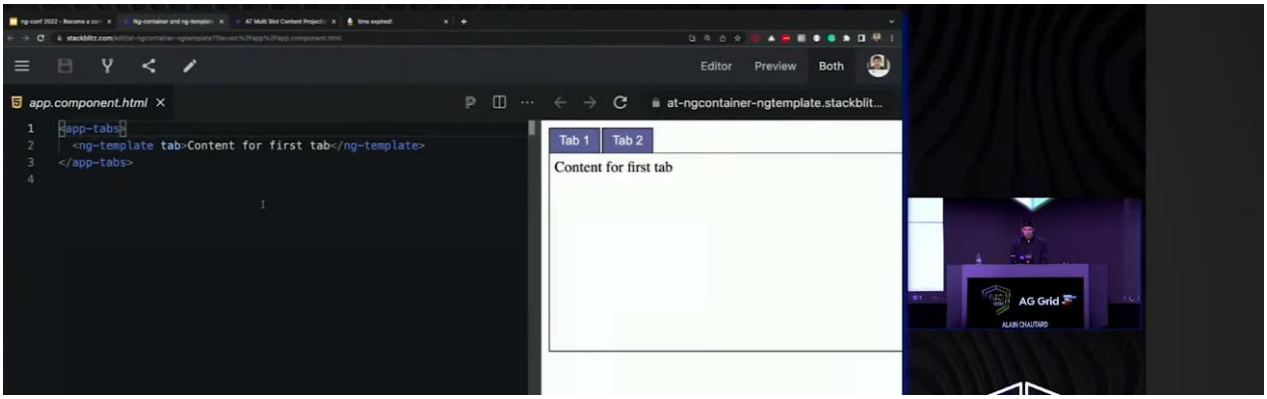
By default, an >ng-template> itself is not visible or rendered in the DOM, it is only rendered when angular needs it to display in a container.

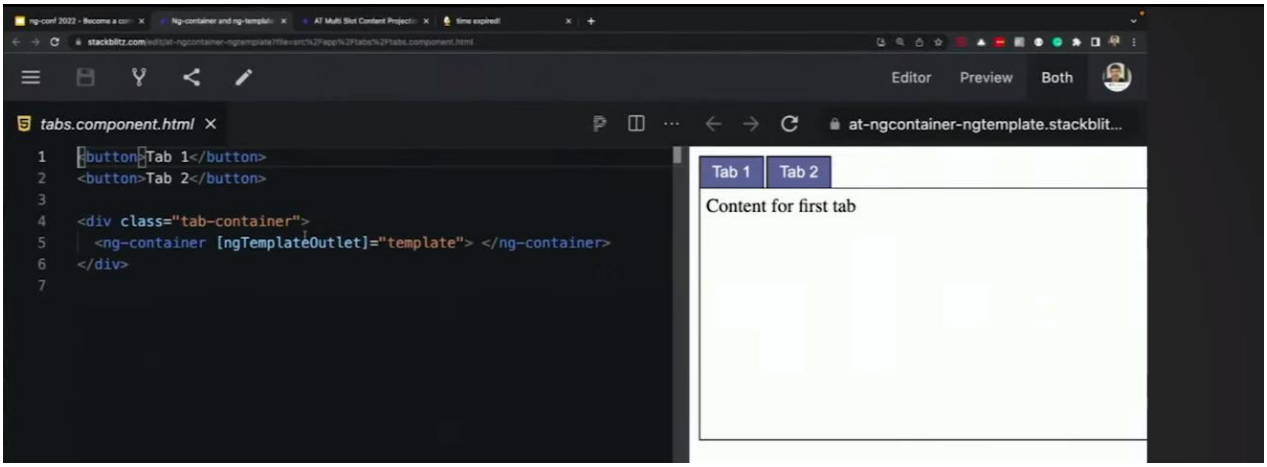
Exercise #2 - ng-container, ng-template, and more!

- 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>;
}
```

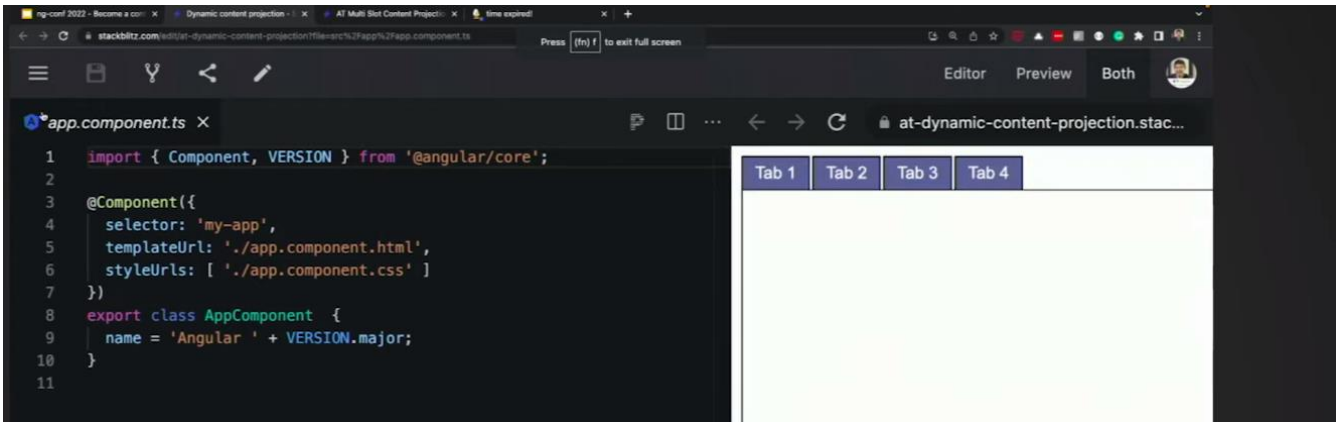
tabs.components.ts

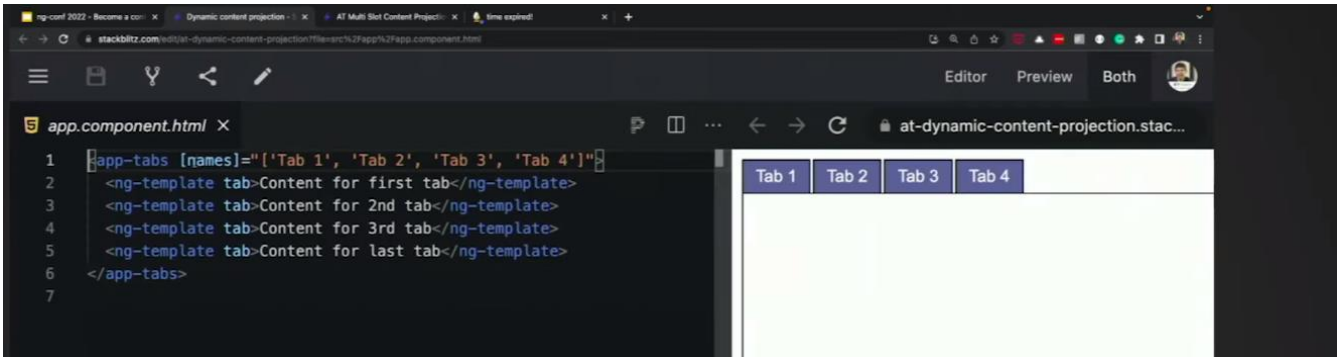
The @ContentChildren directive will query the content to find all our templates as a QueryList.

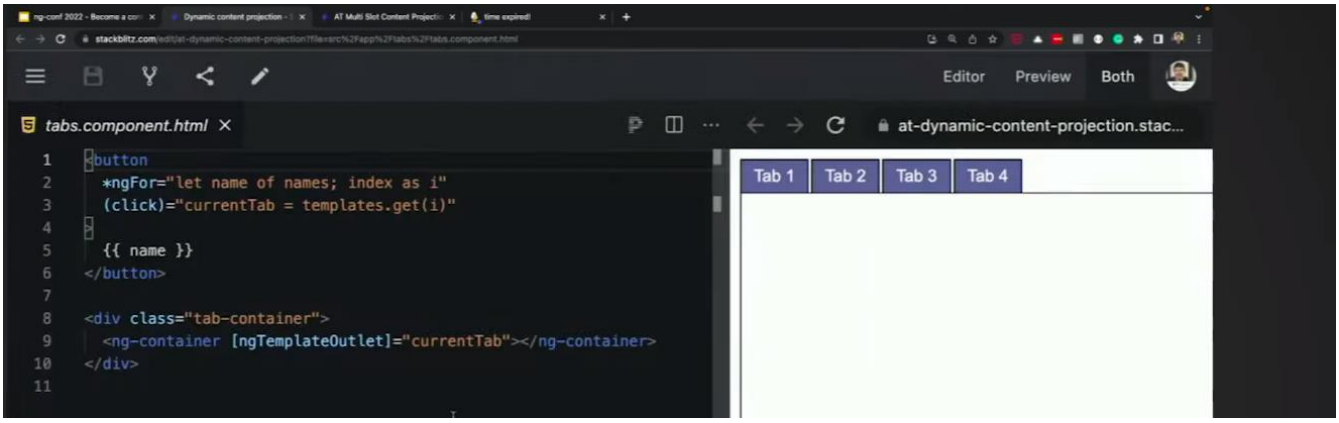
When a tab is clicked, we switch the current template using our template list:

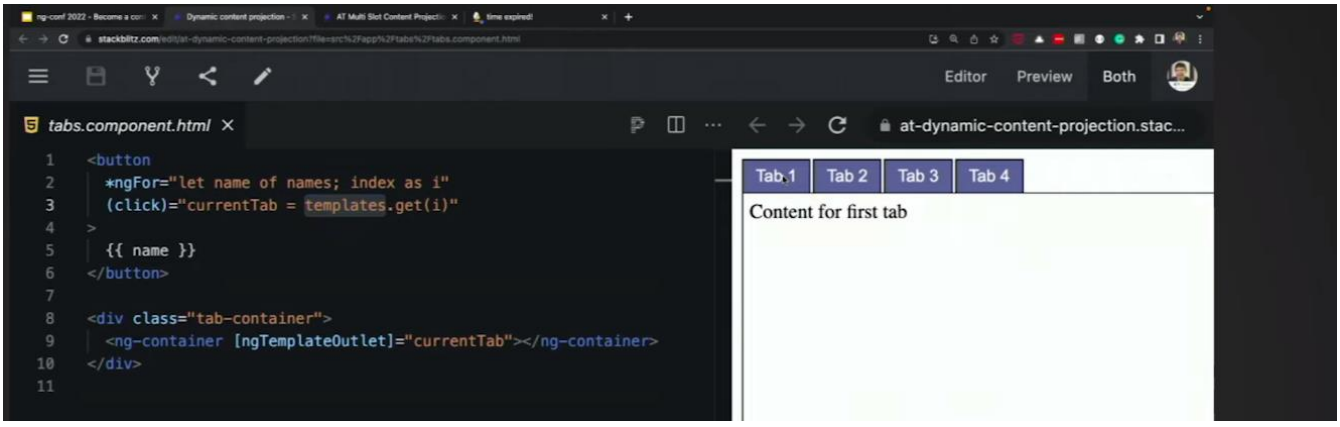
```
<button  *ngFor="let name of names; index as i"
    (click)="currentTab = templates.get(i)">
  {{ name }}
</button>

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

tabs.component.html

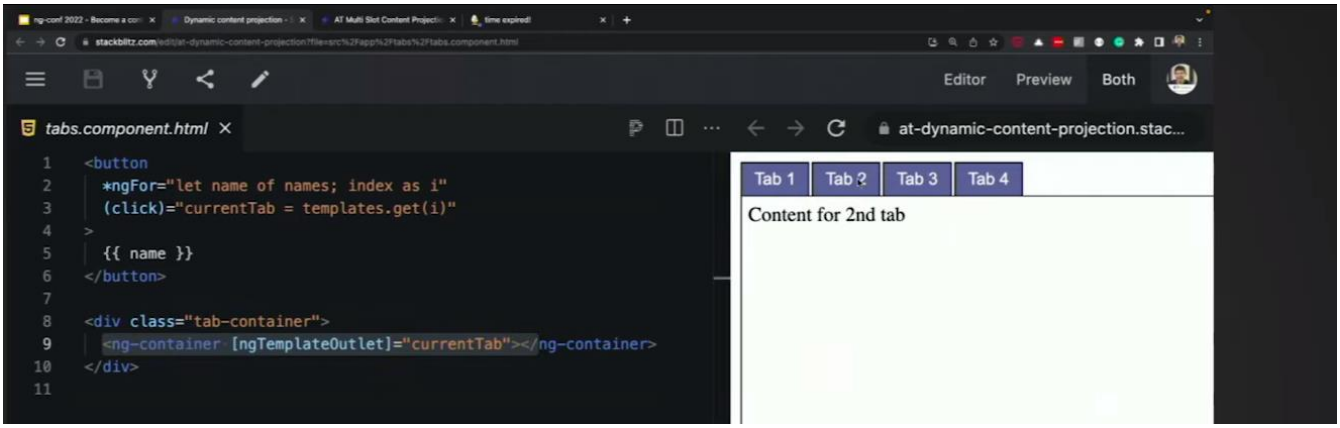


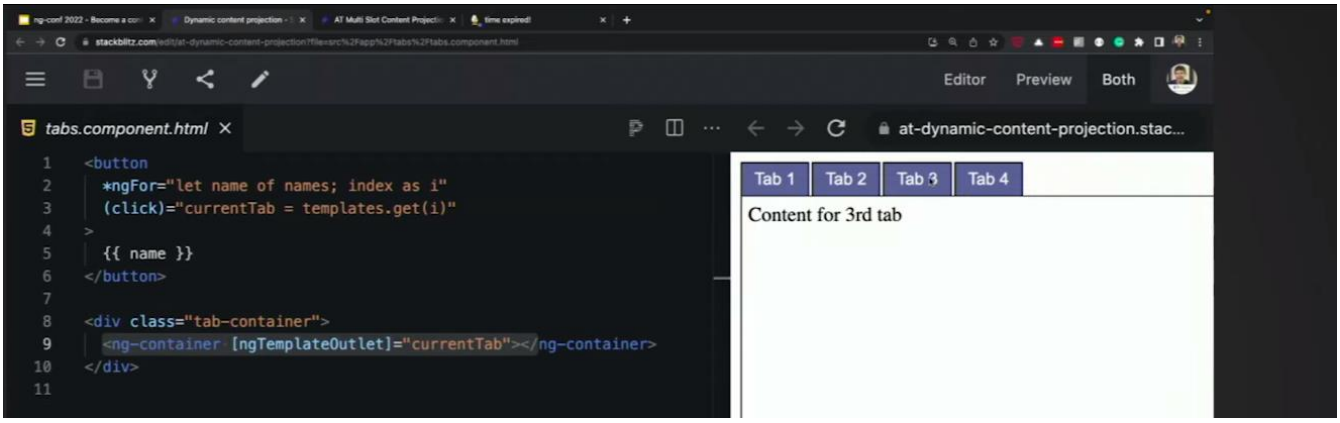


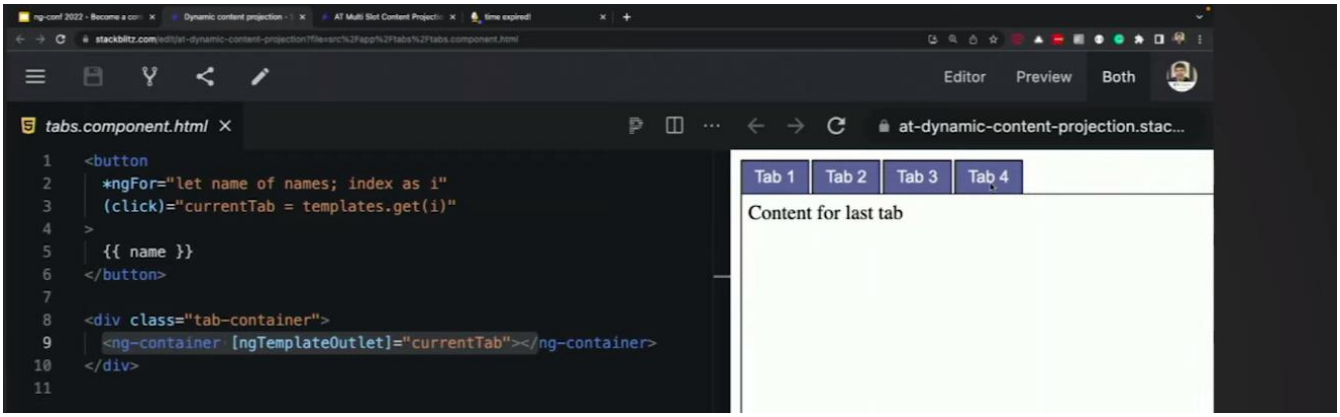


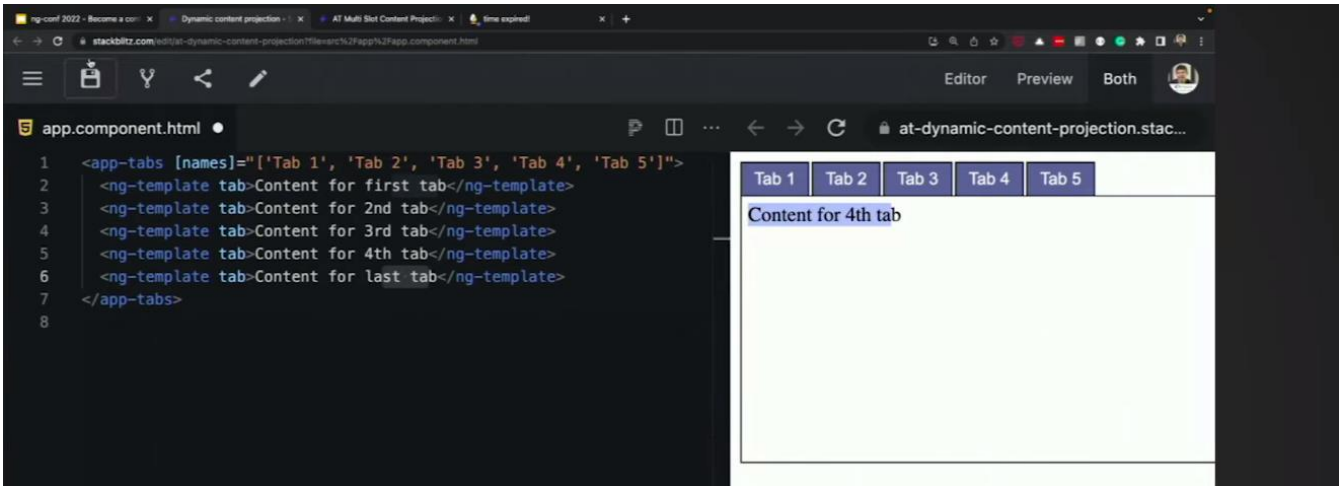


Clicking in the Tab 1 means that it will get the content for the first ng-template and set the value inside the container for display.









The 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;
}
```

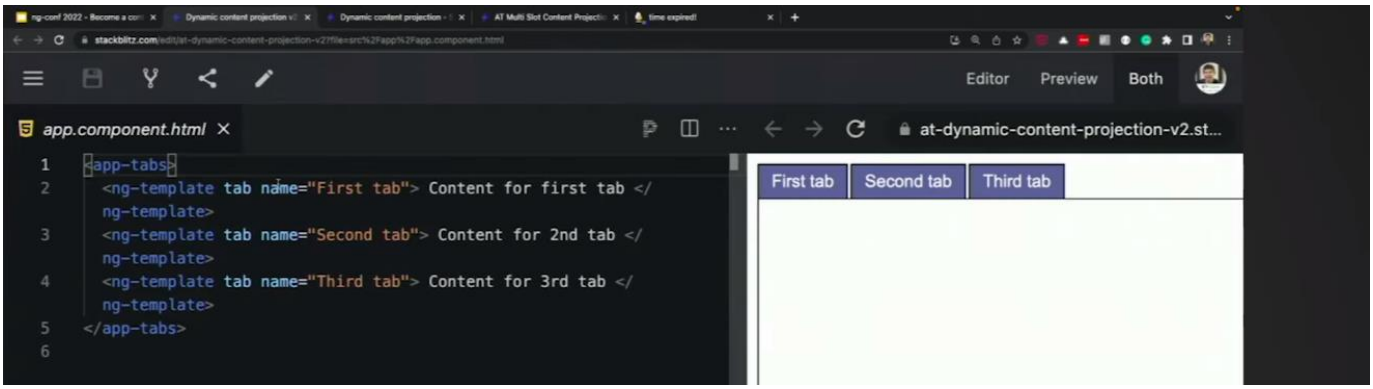
tab.directive.ts

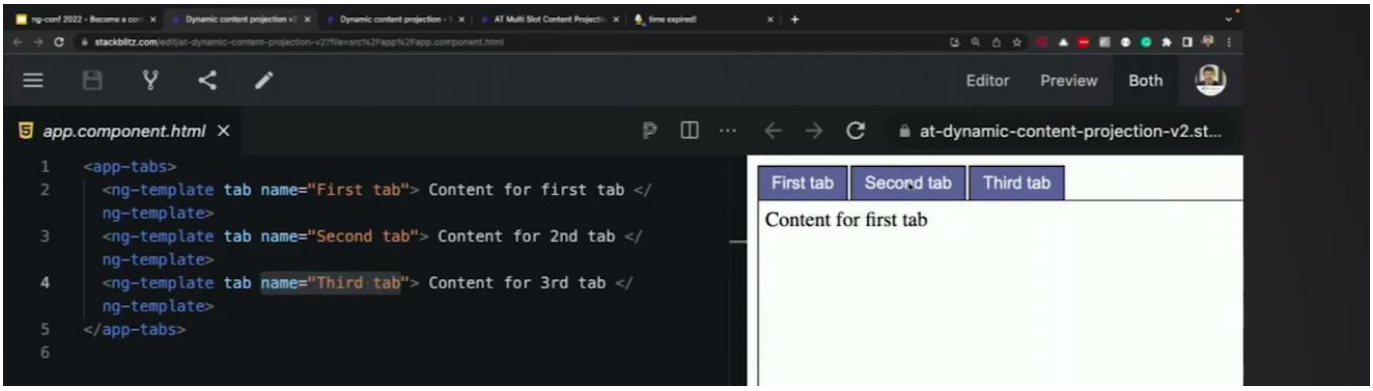
```
<button
  *ngFor="let tab of tabs; index as i"
  (click)="currentTab = templates.get(i)">
  {{ tab.name }}
</button>
```

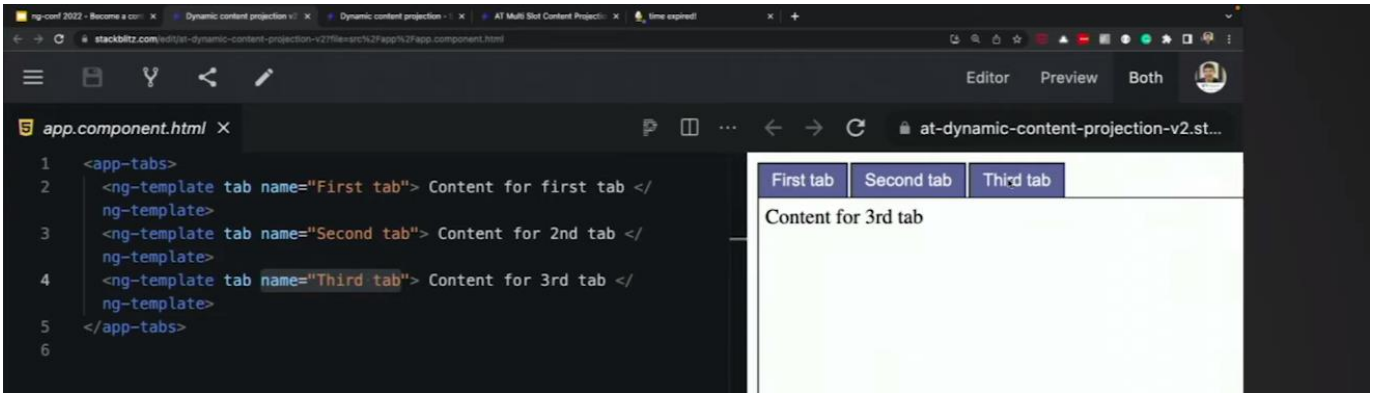
tabs.component.html

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

  @ContentChildren(TabDirective)
  tabs: QueryList<TabDirective>;

  currentTab: TemplateRef<any>;
}
```

tabs.component.ts







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:

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

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

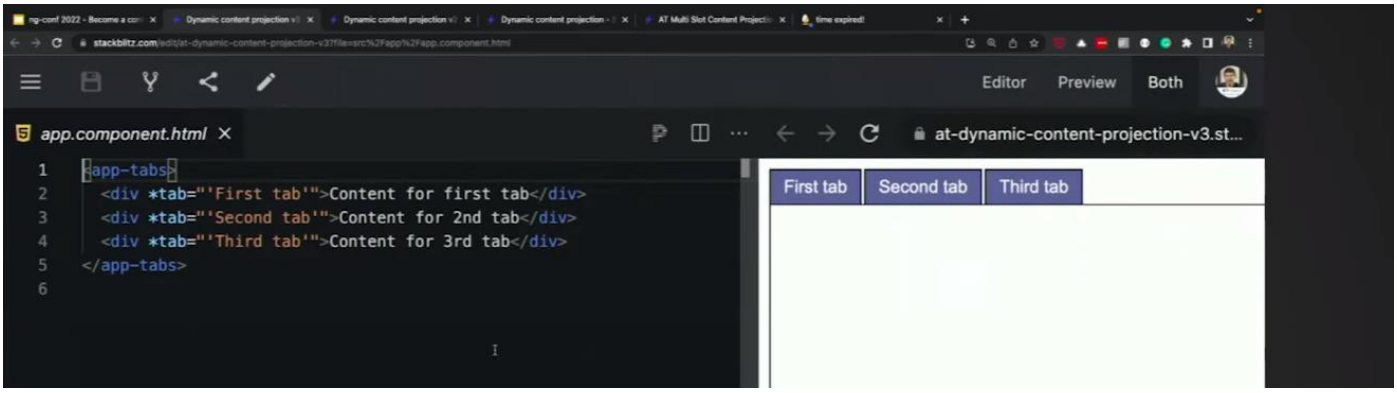
tab.directive.ts

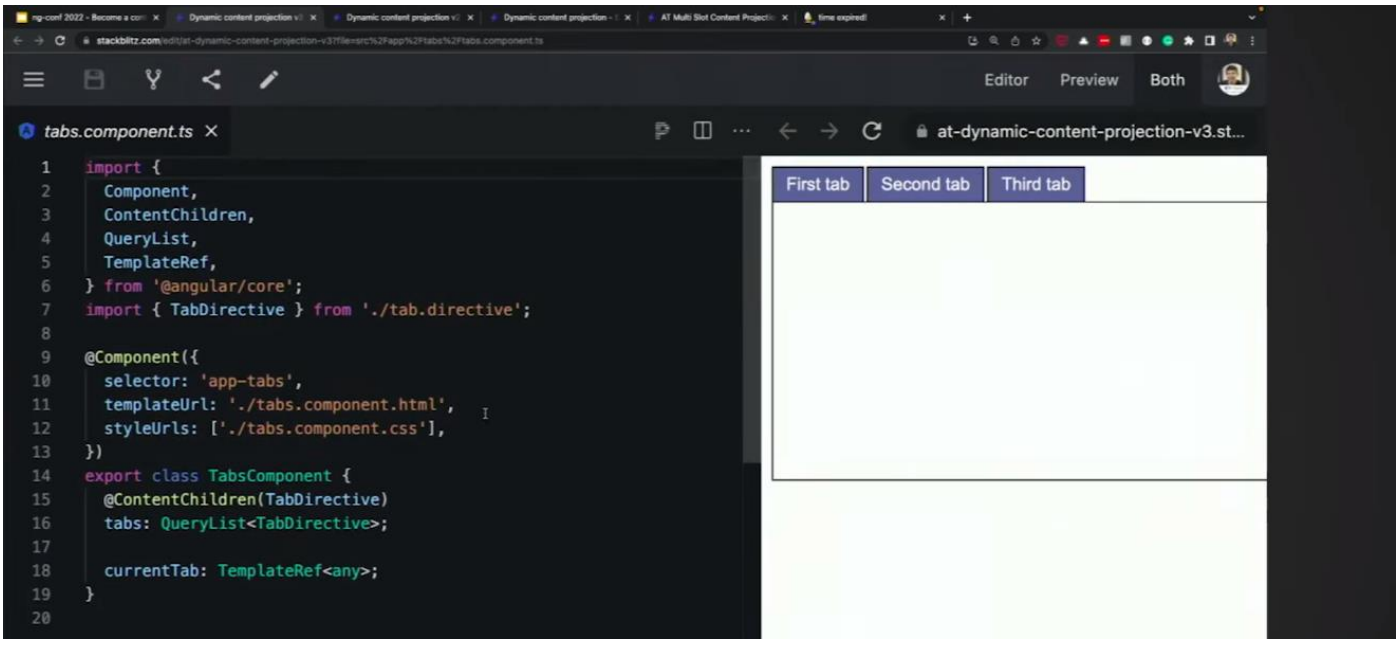

```ts
export class TabsComponent {
  @ContentChildren(TabDirective)
  tabs: QueryList<TabDirective>;

  currentTab: TemplateRef<any>;
}
```

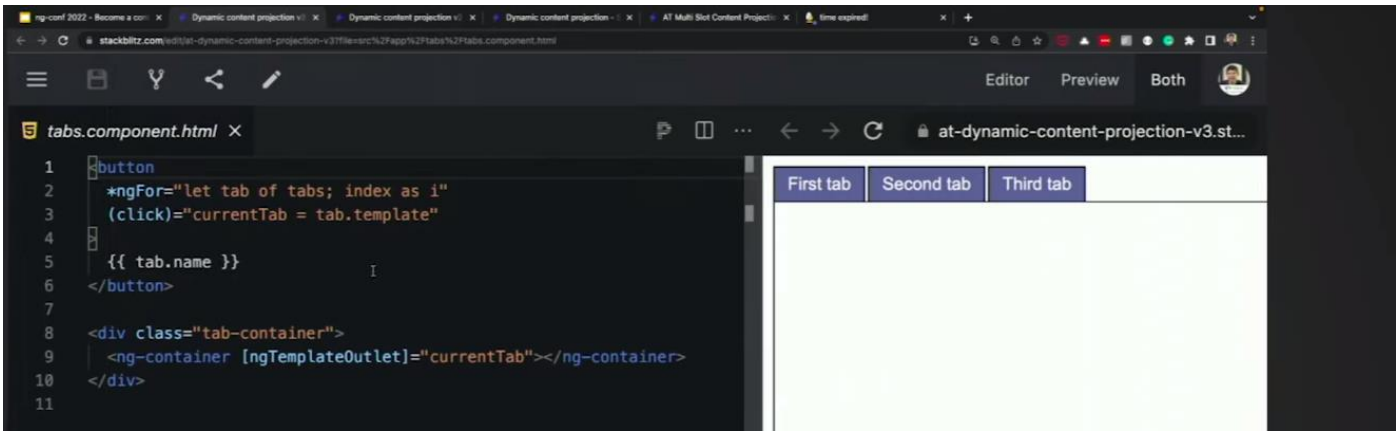
tabs.component.ts

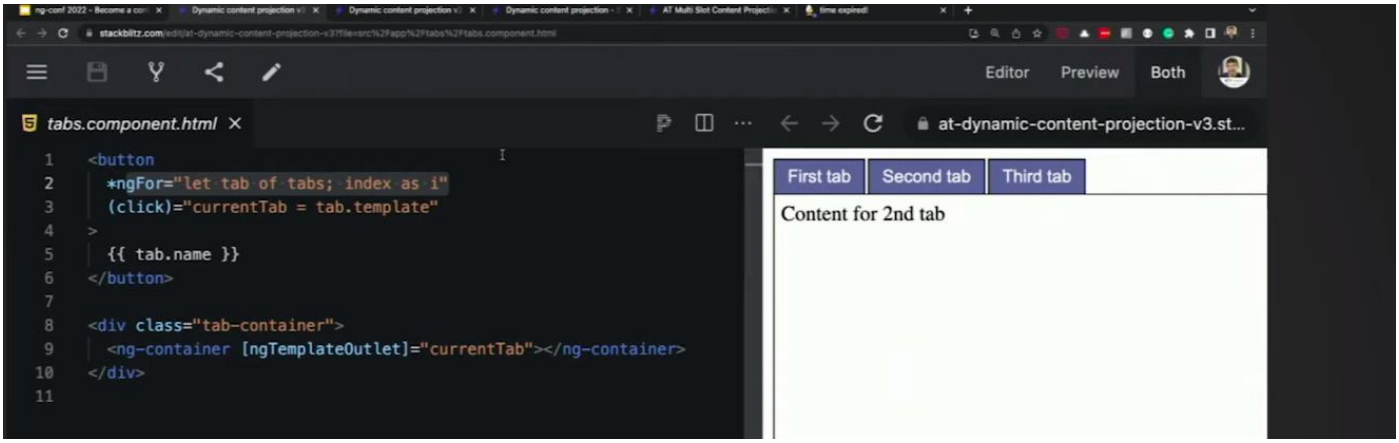
```html
<button *ngFor="let tab of tabs; index as i"
        (click)="currentTab = tab.template">
  {{ tab.name }}
</button>
```

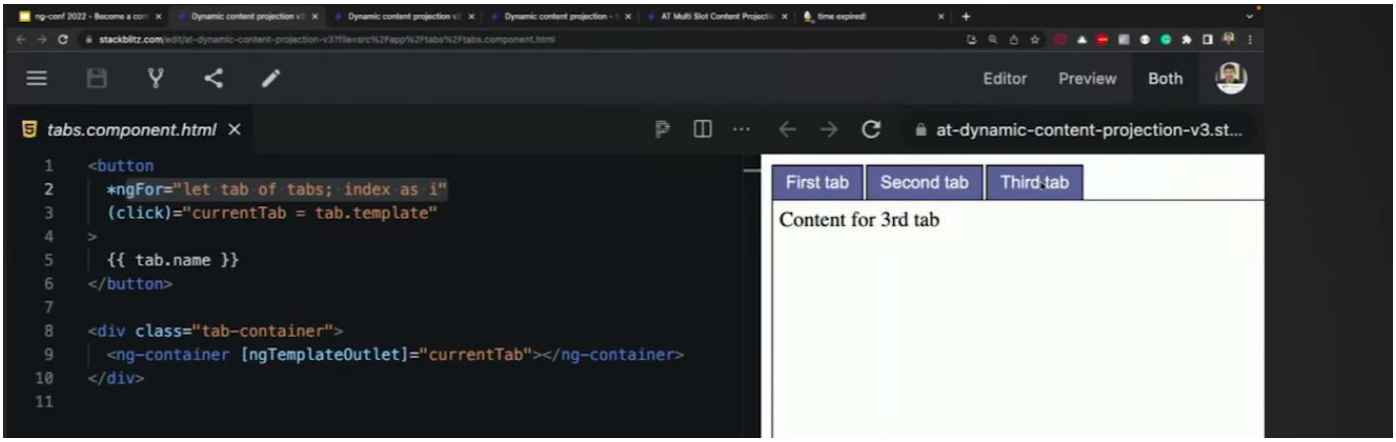
tabs.component.html









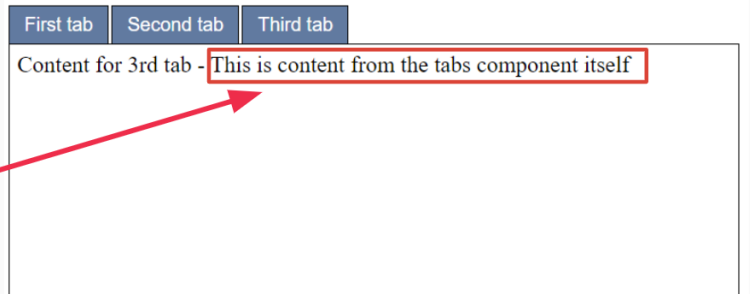


Option #5
Accessing context from the host component

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!

The content of a tab could be using some data from the host component itself



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>
```

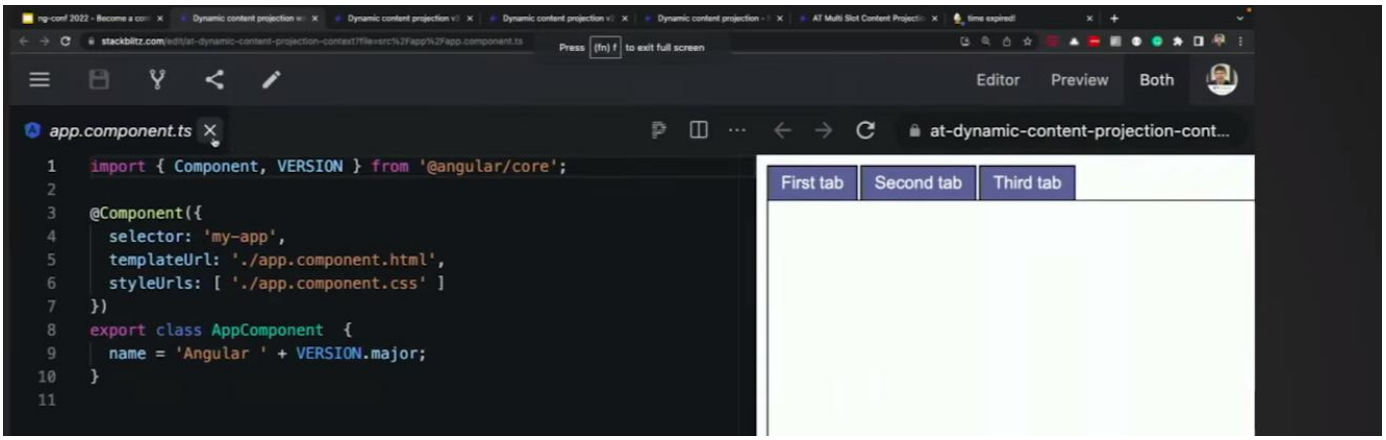
DEMO

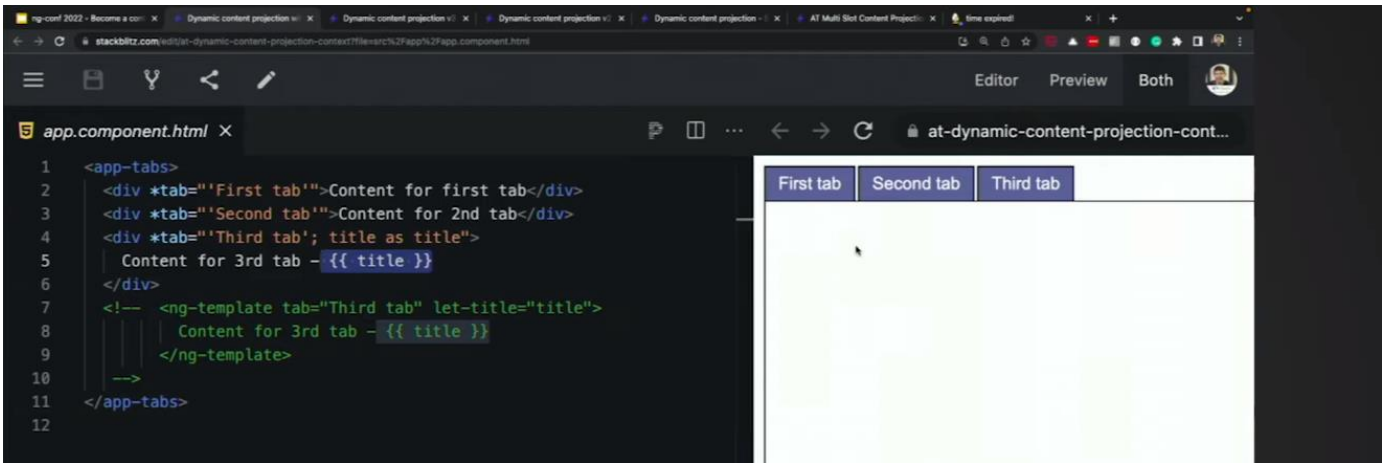
tabs.component.ts

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

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

  currentTab: TemplateRef<any>;
}
```

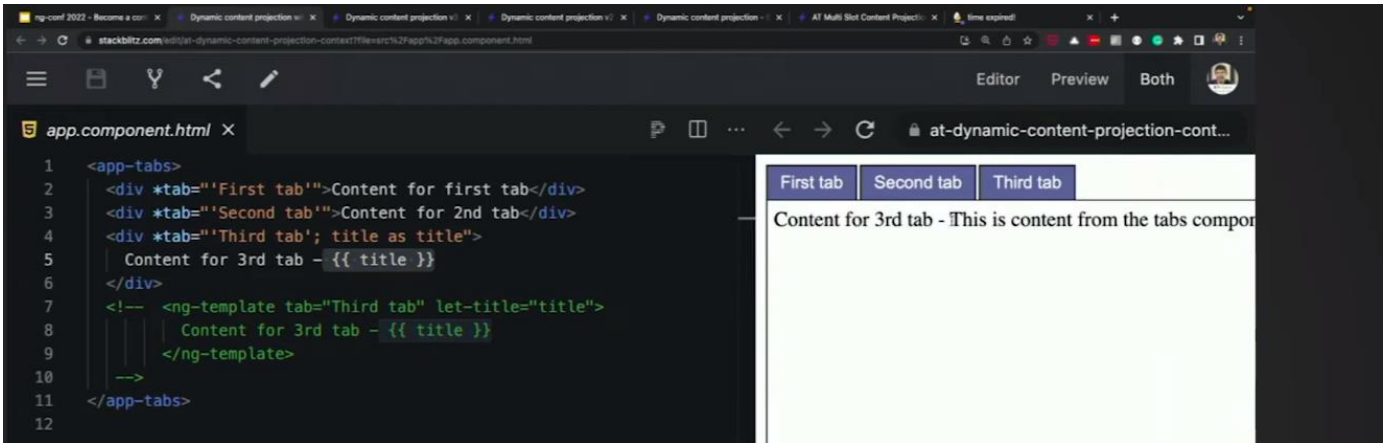
tabs.component.html

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

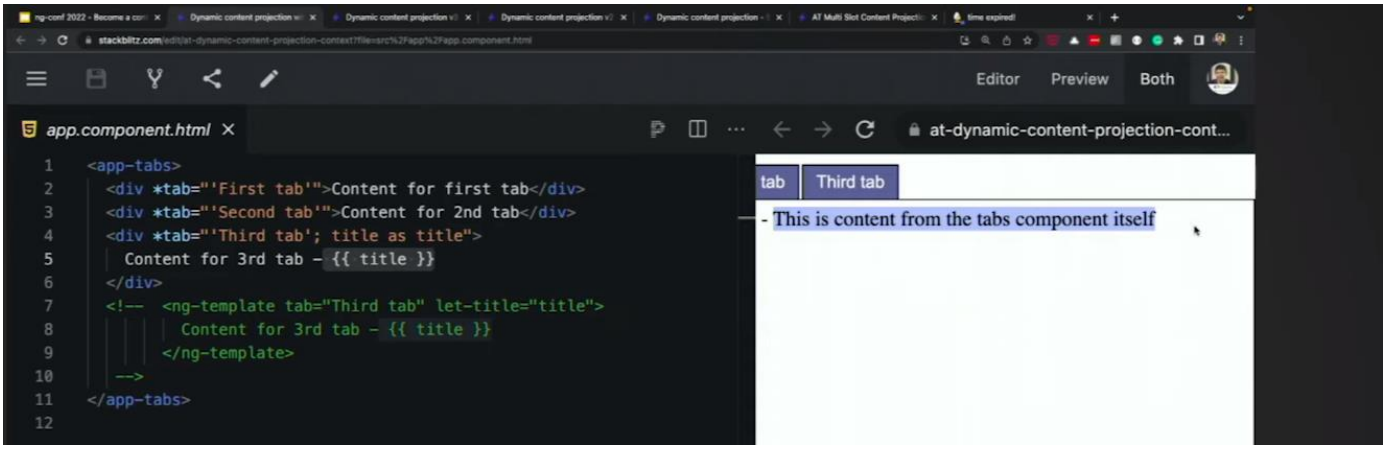



Editor
Preview
Both
app.component.html
at-dynamic-content-projection-cont...
<app-tabs>
<div *tab="'First tab'">Content for first tab
<div *tab="'Second tab'">Content for 2nd tab
<div *tab="'Third tab'; title as title">
Content for 3rd tab - {{ title }}

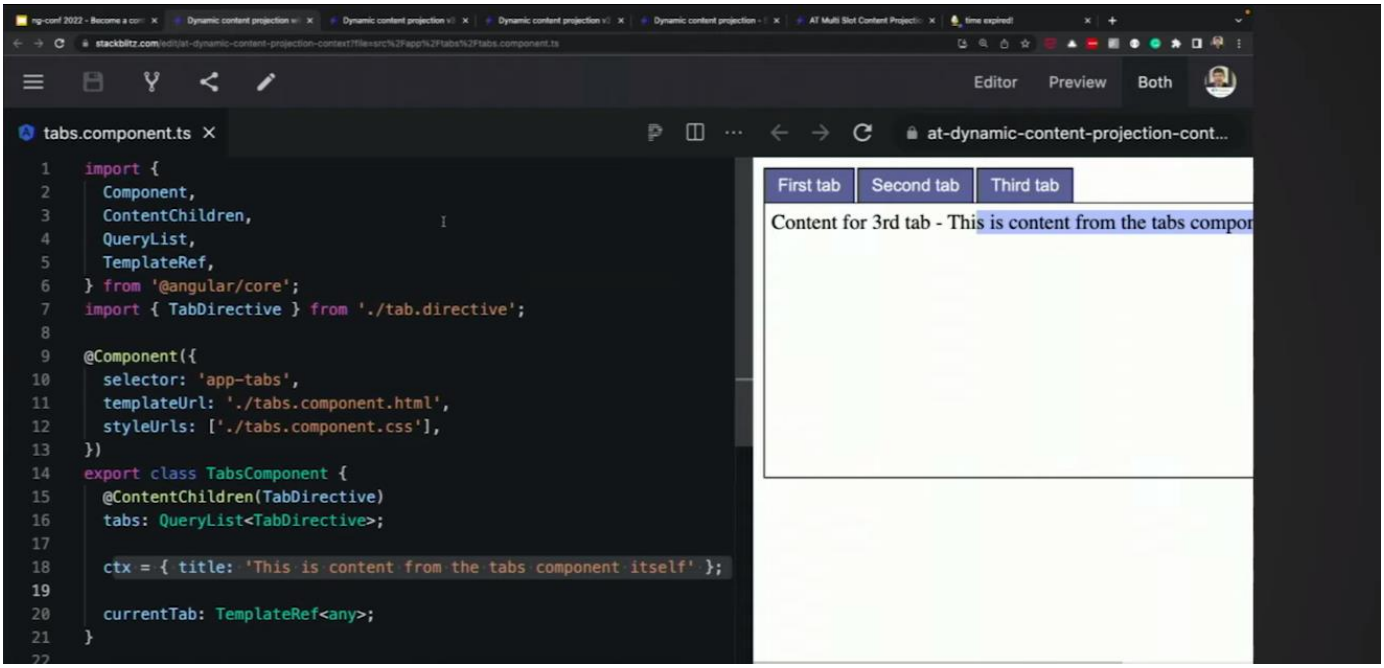
<!-- <ng-template tab="Third tab" let-title="title">
Content for 3rd tab - {{ title }}
</ng-template>
-->
</app-tabs>
First tab
Second tab
Third tab


Editor
Preview
Both
app.component.html
at-dynamic-content-projection-cont...
<app-tabs>
<div *tab="'First tab'">Content for first tab
<div *tab="'Second tab'">Content for 2nd tab
<div *tab="'Third tab'; title as title">
Content for 3rd tab - {{ title }}

<!-- <ng-template tab="Third tab" let-title="title">
Content for 3rd tab - {{ title }}
</ng-template>
-->
</app-tabs>
First tab
Second tab
Third tab
Content for 3rd tab - This is content from the tabs compor


Editor
Preview
Both
app.component.html
at-dynamic-content-projection-cont...
<app-tabs>
<div *tab="'First tab'">Content for first tab
<div *tab="'Second tab'">Content for 2nd tab
<div *tab="'Third tab'; title as title">
Content for 3rd tab - {{ title }}

<!-- <ng-template tab="Third tab" let-title="title">
Content for 3rd tab - {{ title }}
</ng-template>
-->
</app-tabs>
tab
Third 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

Twitter: @AlainChautard

https://blog.angulartraining.com

Q&A

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

Email: 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