Angular 7 Animations.

Getting familiar with animations in Angular.

Onejohi Follow
Jan 13, 2019 · 5 min read

We've come a long way from seeing something almost familiar to the gif above on our screens to our current gorgeous GUIs that give us the power to construct some serious motion illusions, animations.

A webpage is constructed using HTML code and styled using CSS rules while JavaScript handles interactions. When designing complex web applications, you might want to change a button's look depending on a particular state of the application, say, when a user is following or isn't following another. Other times you might want to create a toggle button, where a user can switch between two states or multiple states and you'd want to visually notify the user of this state change. This means you'll have to change an element's style, like a button from red to blue, or enlarge the element to dominate space.

Angular makes it easy for you to provide the illusion of motion through user interaction and also in response to changes in the application state. This lets you integrate your animation logic with the rest of the code for easier control.

Setup.

Before you can use animations, you need to import a few animation-specific modules to your root application.

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
3
4  @NgModule({
5    imports: [ BrowserModule, BrowserAnimationsModule ],
6    // other declarations
7  })
8  export class AppModule { }
```

app.module.ts hosted with ❤ by **GitHub** **view raw**

Once you're done with your imports, you can now start using animations. Let's simulate a button that toggles between two states. When the button is active, it should be green and when it's inactive, it should be red.

In our component.ts file, we should create a toggle function that toggles a state between active and inactive.

```
1  export class AnimationPage {
2    state: String = 'inactive'
3    constructor() {}
4
5    toggleState() {
6      this.state = this.state === 'active' ? 'inactive' : 'active'
7    }
8  }
```

animations.component.ts hosted with ❤ by **GitHub** **view raw**

Now, that we have our toggle state set. You can add `console.log(this.state);` after line 6 to see the state change when you attach the event to a particular button on the template using `<button (click)='toggleState()'> Change State </button>` .

Let's now build a simple animation that transitions this button between two states driven by the value of the state.

```
import { Component, Input } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';

@Component({
  selector: 'animation-el',
  templateUrl: 'animated-button.html',
  animations: [
    trigger('buttonState', [
      state('inactive', style({
        backgroundColor: 'red'
       })),
      state('active', style({
        backgroundColor: 'green'
       })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
  ]
})

export class AnimationPage { ... // refer to code above. }
```

animation.component.ts hosted with ❤ by **GitHub** **view raw**

As you can see, we import trigger, state, style, animate and transition to create a simple animation where the background color is changed from red to green in a transition that lasts 100ms. Please note that it's `backgroundColor` not `background-color` as used in CSS. You will have to be careful when naming attributes that have a dash in CSS as dashes aren't allowed in JS, thereby `fontFamily`, `textAlign` is the way to go about it.

Clicking the button still changes the state, but the style is not applied to the button as we haven't attached the trigger to the button. So in our `'animated-button.html` we'll use the `[@triggerName]` syntax to attach the animation to one or more buttons.

```
<button [@buttonState]="state" (click)="toggleState()"> Change State </button>

<!-- Multiple buttons can be applied to as well -->
<button [@buttonState]="state" (click)="toggleState()"> Change State 2</button>
```

```
<button [@buttonState]="state" (click)="toggleState()"> Change State 3</button>
<button [@buttonState]="state" (click)="toggleState()"> Change State 4</button>
```

animated-button.html hosted with ❤ by **GitHub** **view raw**

To apply a custom state to each button, create instances of multiple buttons where each can have it's own state thereby `toggleState()` will only affect one particular element. For instance, let's say you have a list of users, each user should have a property of `state` which should be of type `String` .

Our template code will thereby change to.

```
<li>
  <button *ngFor="let user of users" (click)="toggleState()" [@buttonState]="user.state">
</li>
```

animated-button2.html hosted with ❤ by **GitHub** **view raw**

States and transitions.

Angular animations are centered around states and transitions. An animation state can be anything, a string or boolean that defines a particular state. It could be a simple attribute or based on a computed value based on a method. You could compute to test if a person is older than 18 based on the year his birthday data is showing to determine if a particular button should be disabled or enabled. 😉

> *These* `state` *definitions specify the* end styles *of each state. They are applied to the element once it has transitioned to that state, and stay* as long as it remains in that state. *In effect, you're defining what styles the element has in different states.*

After you define states, you can define transitions between the states to control the timing of switching between one state to the other.

If multiple transitions have the same timing, you can combine them into the same transition definition.

```
transition('inative => active, active => inactive',
  animate('100ms ease-out'))
```

Or use the shorter hand syntax if two transitions have the same timing `<=>`

```
transition('inative <=> active', animate('100ms ease-out'))
```

You can also add styles during transitions, but they're not kept around after the transition finishes. You define the styles inline with the transition.

```
transition('inactive => active', [
  style({
    backgroundColor: 'yellow',
    transform: 'scale(1.3)'
  }),
  animate('80ms ease-in', style({
    backgroundColor: 'pink',
    transform: 'scale(1)'
  }))
])
```

inline-transition.ts hosted with ❤ by **GitHub** **view raw**

Wildcards. (*)

Wildcards are used to mostly mean all, and mostly it's the asterisk (*) used as the symbol to indicate a wildcard. A wildcard matches all states. Therefore `active => *` is applied when an element changes state from active to **anything else**. `* <=> *` would be applied when any change between two states take place.

There's also a special state called the `void` state that can apply to any animation. It applies when an element is not attached to a view because it has not been added or because the element has been removed. It's very useful to animate elements that are entering and leaving the view. Here is an example where you an use `void => *` and `* => void` syntax to animate elements in and out of the view.

```
animations: [
  trigger('flyInOut', [
    state('in', style({transform: 'translateX(0)'})),
    transition('void => *', [
      style({transform: 'translateX(-100%)'}),
```

```
      animate(100)
    ]),
    transition('* => void', [
      animate(100, style({transform: 'translateX(100%)'}))
    ])
  ])
]
```

in-out-view.ts hosted with ❤ by GitHub view raw

Here's a slightly complex animation I did that transitions between inactive user enter: `void => inactive` , active user enter: `void => active` , inactive user leave: `inactive => void` and, active user leave: `active => void` to give me fine control over each of these actions.

```
animations: [
  trigger('userState', [
    state('inactive', style({transform: 'translateX(0) scale(1)'})),
    state('active',   style({transform: 'translateX(0) scale(1.1)'})),
    transition('inactive => active', animate('100ms ease-in')),
    transition('active => inactive', animate('100ms ease-out')),
    transition('void => inactive', [
      style({transform: 'translateX(-100%) scale(1)'}),
      animate(100)
    ]),
    transition('inactive => void', [
      animate(100, style({transform: 'translateX(100%) scale(1)'}))
    ]),
    transition('void => active', [
      style({transform: 'translateX(0) scale(0)'}),
      animate(200)
    ]),
    transition('active => void', [
      animate(200, style({transform: 'translateX(0) scale(0)'}))
    ])
  ])
]
```

custom-animation.ts hosted with ❤ by GitHub view raw

In special cases, you might not know how large a particular object can be before animating it in. For instance, let's assume you're animating images from unknown

sources, you can use the wildcard approach to define its height during transitions or in a particular state.

```
state('active', style({ height: '*' }))
```

Conclusion.

You can use angular animations to create complex applications that react to certain events, states that could be global or specific to instances of objects inside the view. This gives you a lot of power to use animations to interact with users by changing element looks to visually define states of particular objects in view or different application states. This makes it much simpler to create this interaction and improve the overall UX (User eXperience) of your application.

To keep things simple, we'll talk more about some advanced Angular animations in the next article next week like Animation Timing where you can tune every animated transition, the delay and easing functions. I'll also cover multi-step animations with keyframes, parallel animation groups, and animation callbacks.

In the meantime, I hope you've gained some knowledge about animations and how you can use Angular to make some pretty cool motions. Leave a few claps, thank you for your time! 😀.

CSS Angular Angular Animations Animation