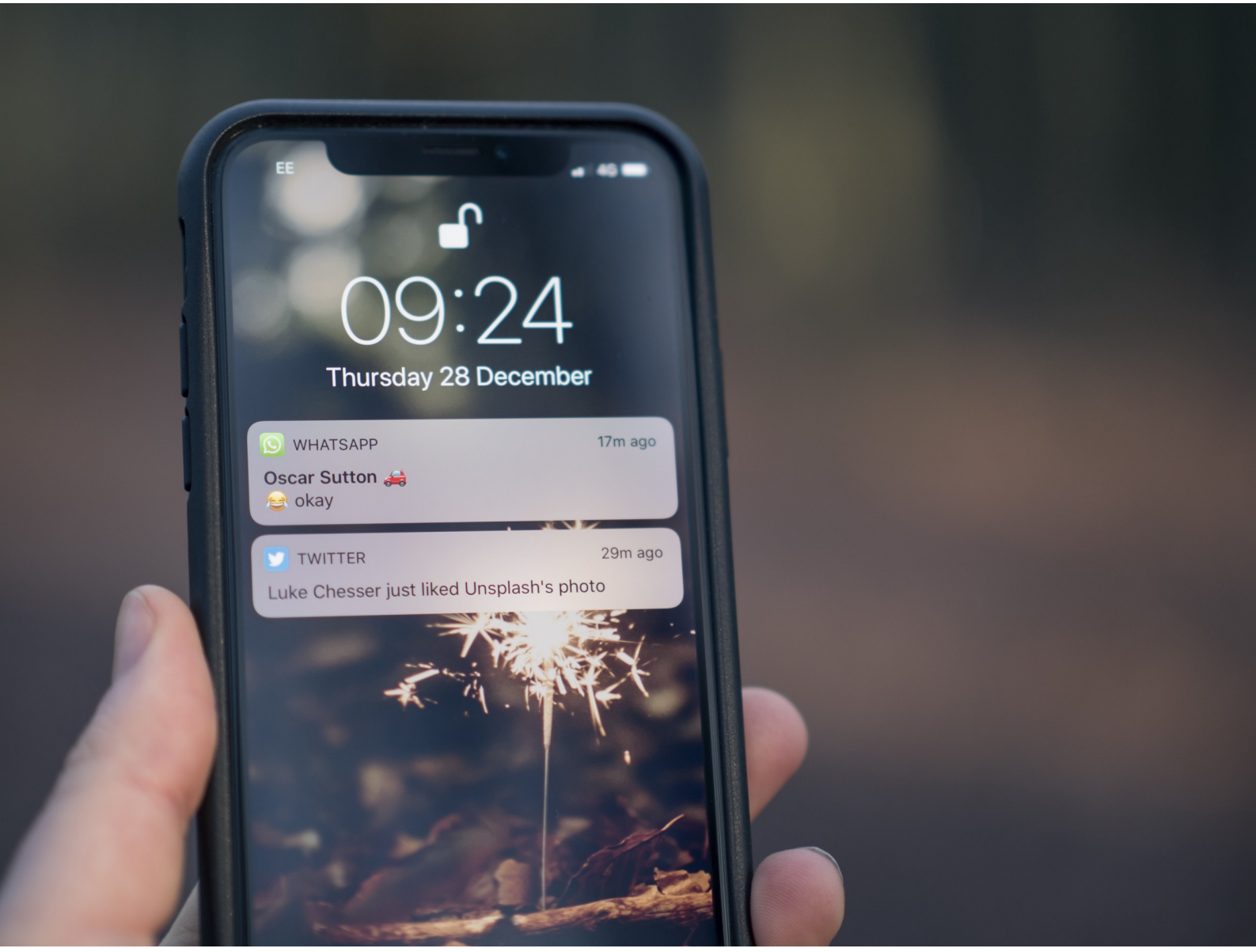
Creating a toast service with Angular CDK

Adrian Fâciu Follow
Nov 21, 2018 · 14 min read



"person holding phone" by Jamie Street on Unsplash

OK, not that kind of toast message, but something close 😃 . Prepare yourself, this will be a long read, starting from scratch up to a full-blown, *almost,* **production-ready** toast service.

Angular Material is a great material UI design components library for your Angular applications. All the common parts needed to create components, things like layout, accessibility, common components like grid or tree have been isolated inside the CDK (Component Development Kit).

You can use the CDK to develop custom components. In this post, we will explore how to leverage this to create a toast service that can be used to show toast messages throughout your application.

Setup

First, we need to add CDK as a dependency to our project:

```
yarn add @angular/cdk
```

or if you prefer npm:

```
npm install @angular/cdk
```

Now we can use what we need from the CDK. We'll mainly use the Overlay package that will allow us to open floating panels around the screen which is exactly what we need to show toast messages. Internally this package is using the Portal package that allows us to render dynamic content throughout the application.

Toast component

We can start with the basic thing that we need, meaning a toast component:

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-toast',
5    template: `
6      <div>This is a toast message</div>
7    `
```

This is the simplest component possible, a selector and a `<div>` with some text. Further down, we'll extend it to suit our needs. But for now we'll focus on displaying it on the screen.

First thing after we have a component is to add it to an NgModule. In this case since we'll not use the component directly in a template we'll have to also add it to the `entryComponents` list, otherwise, the compiler will notice it's not used and will remove it from the final bundle.

Since we're good citizens and we can think like we're creating a library we can create a module file just for this toast component:

```
1  import { NgModule } from '@angular/core'
2
3  import { ToastComponent } from './toast.component';
4
5  @NgModule({
6    declarations: [ToastComponent],
7    entryComponents: [ToastComponent]
```

The only thing left to do with this module is add it in the imports array of our main application module.

Toast service

Now that we have a component and a module for our toast, we can start thinking about what we need next. From our application, we want to programmatically show toast messages whenever it's needed. So the best option for this would be a toast service:

```
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class ToastService {
```

We have a simple service with an empty (for now) method *show*. The only notable thing is that we use the new *providedIn* attribute of the Injectable decorator that's available since Angular 6, specifying 'root' as a value. This means that this service will be registered as a

singleton in our application, and that we don't need to add it to the providers array anymore.

This actually means that our services are now *tree-shakeable*. If we have a service defined in our application, but it's not referenced in any other place, it will be removed from the final bundle. This was not the case when we had to add them to the providers array of a module. Angular itself is starting to move its built-in services to this way of registering them and I suggest you start doing it as well.

And now to our show toast method: we'll need to get an instance of the Overlay service, create an instance of our toast component, and display it somewhere on the screen.

We can achieve this with three lines of code:

```
const overlayRef = this.overlay.create();
const toastPortal = new ComponentPortal(ToastComponent);
overlayRef.attach(toastPortal);
```

First, we create an overlay using the *create* method on the overlay service. This will create a container in our application that will host our toast message. We get back *a reference* to this container that we can use to control it.

Secondly, we have to create an instance of our toast component. Since we're trying to dynamically render a component, we can use the portal package to achieve this. We import the **ComponentPortal** class from `@angular/cdk/portal` and we create a new instance passing our toast component class as an argument. This will create an instance of our component and give us back a wrapper that we can use to handle it.

The last thing we need to do in order to show our toast is to get our toast portal and attach it to the overlay container that we previously created.

Now, if we call the show method on our toast service in our app we'll see a div with a text added to our application. But it does not look like a floating container or a toast. Let's quickly fix this.

The default positioning classes used by the overlay container are exported from CDK so that we can use them. Either we import them

or create some custom ones. Since I'm always in favor of reusing functionality, we'll import the existing ones.

We can go to our main **styles.css** file and import the prebuilt overlay css there:

```
@import "~@angular/cdk/overlay-prebuilt.css";
```

If we use scss we can import it too, but we'll also have to call the mixin. Otherwise, nothing will happen:

```
@import "~@angular/cdk/overlay";

@include cdk-overlay();
```

If we call our method, we'll see that the message is floating in our top left corner. If we call the method multiple times, we'll notice that we can see only one message, while previously they would stack one beneath the other. We can see only one because by default now they are placed on top of each other.

In order to make it look more like a toast, we'll add some basic styling (similar to what Bootstrap has) and use some of the icons that Angular Material provides.

Our toast component will look a bit more complex now:

```
<div class="toast">
  <mat-icon>done</mat-icon>
  <div>This is a toast message</div>
  <mat-icon (click)="close()">close</mat-icon>
```

Now we have defined the toast CSS class to make it look a bit more like a toast message:

```
.toast {
  position: relative;
  display: flex;
  justify-content: space-around;
  margin-bottom: 20px;
  padding: 10px 15px 10px 48px;
  width: 290px;
  background: #fff;
  border-width: 1px;
```

I assume we don't always want to show the text "This is a toast" when displaying the message. We'll make this configurable so that each time we call the show method we can pass in what text we want to show.

Since we'll extend this further on, let's create an interface to represent the data we might send:

```
export interface ToastData {
  text: string;
}
```

For now, it's OK if we just have the text there. We also update the calls to our service with something like:

```
toastService.show({ text: 'Everything is ok!' });
```

Next, we have to pass this data to our component, somehow. We're going to achieve this using Angular's dependency injection system. In the toast component class we'll add it to the constructor:

```
export class ToastComponent {
  constructor(readonly data: ToastData) { }
}
```

Then we'll use it in the component's template:

```
<div>{{ data.text }}</div>
```

We also have to make sure we add this data to the component injector so we have to update our toast service a bit. Thankfully, when we create the component portal (for the toast) we can pass in an injector that will be attached to it. Angular CDK provides a PortalInjector class which extends the basic Injector one from Angular so we'll use it.

First, we create a map with the custom tokens we want to add in the injector and then instantiate the PortalInjector:

```
const tokens = new WeakMap();
tokens.set(ToastData, data);

const injector = new PortalInjector(parentInjector, tokens);
```

The first parameter of the portal injector constructor is a *parent injector,* and we can get an instance of it using dependency injection. This is used to retrieve any dependencies that we don't add to the custom tokens and we use in the toast component.

Then we can create our component portal with our injector attached:

```
const toastPortal =
    new ComponentPortal(ToastComponent, null, injector);
```

But if we try it, we'll notice it won't work yet. `ToastData` is an interface which is not a valid object for our map. There are two options here, one would be to make it a *class*, and everything will work. Or, we can create a custom InjectionToken and use that one when we create the map and require the data.

If we want to go for the injection token, we'll have to create it:

```
const TOAST_DATA = new InjectionToken<ToastData>
('TOAST_DATA');
```

Use it in our map:

```
tokens.set(TOAST_DATA, data);
```

And we also update the constructor for the Toast component to use the inject decorator:

```
constructor(@Inject(TOAST_DATA) readonly data: ToastData) {
}
```

Now our toast is configurable and we can pass in whatever text we want.

As a bonus to this, it would be easy enough to configure the type of toast we're showing and maybe display a different image based on that.

We can define our toast type as one of success, info or warning. We left error out because it's not a good UX practice to show errors in the form of a toast.

We can create a TypeScript union type to represent this information:

```
export type ToastType = 'warning' | 'info' | 'success';
```

And extend our ToastData to include it:

```
export class ToastData {
  text: string;
  type: ToastType;
}
```

When we want to show a toast message we also specify the type:

```
toastService.show({ text: 'Everything is ok!', type:
'success'});
```

You can use any icons you want but, in our case, we'll go with the ones provided by the Material library. If we don't have it already we need to make sure we have the `material` module added as a dependency:

```
yarn add @angular/material
```

Then we have to add the icon font to our HTML file by including this line:

```
<link href="https://fonts.googleapis.com/icon?
family=Material+Icons"
      rel="stylesheet">
```

Add the Material icon module to our imports:

```
1  import { NgModule } from '@angular/core'
2  import { OverlayModule } from '@angular/cdk/overlay';
3  import { MatIconModule } from '@angular/material/icon'
4
5  import { ToastComponent } from './toast.component';
6
7  @NgModule({
8    imports: [OverlayModule, MatIconModule],
9    declarations: [ToastComponent],
```

And finally, we're able to display an icon:

```
<mat-icon>info</mat-icon>
```

The names of the icons we want to show are identical for two of our toast types (warning and info), but when we have a success toast type

we want to show a *done* icon. We can easily fix this by creating a local property in the component and initializing it in the constructor:

```
this.iconType = data.type === 'success' ? 'done' :
data.type;
```

Updated image usage:

```
<mat-icon> {{ iconType }} </mat-icon>
```

OK, we have a toast message. Now how do we get rid of it? We have to implement some self-closing logic and allow it to be closed both from the UI and programmatically.

In order to make sure we can close the toast from the UI we have to add a close icon/button. We can do that by adding it in the template:

```
<mat-icon (click)="close()">close</mat-icon>
```

Now really, how do we close the toast? It looks like it's not that complicated at all. We show the toast using an overlay, and when we create the overlay we get an instance of an overlay reference back. We can use this to do several things, including destroying it by calling the dispose method:

```
this.overlayRef.dispose();
```

Similar to how we got an overlay reference instance when creating one, we also want to create a toast reference that we can use and return using our service. The only thing we need in it now is the close method:

```
import { OverlayRef } from '@angular/cdk/overlay';

export class ToastRef {
  constructor(readonly overlay: OverlayRef) { }

  close() {
    this.overlay.dispose();
```

We pass in the overlay reference and only expose the close method that will call dispose. We'll use it in two places from the start. We'll return it to the user from the show method of the toast service. And, we'll pass it to the toast component so it can close itself:

```
import { Injectable, Injector } from '@angular/core';
import { Overlay } from '@angular/cdk/overlay';
import { ComponentPortal, PortalInjector } from '@angu

import { ToastComponent } from './toast.component';
import { ToastData } from './toast-config';
import { ToastRef } from './toast-ref';

@Injectable({
  providedIn: 'root'
})
export class ToastService {
  constructor(private overlay: Overlay, private parent

  showToast(data: ToastData) {
    const overlayRef = this.overlay.create();

    const toastRef = new ToastRef(overlayRef);
    const injector = this.getInjector(data, toastRef,
    const toastPortal = new ComponentPortal(ToastCompo

    overlayRef.attach(toastPortal);

```

Instantiate it by passing the overlay reference, return it from the method and add it to the tokens when we create the injector.

In the toast component, we implement the close method. When the user clicks the close icon we actually do something and use a setTimeout to automatically hide the toast after an interval:

```
import { Component, OnInit, OnDestroy } from '@angular

import { ToastData } from './toast-config';
import { ToastRef } from './toast-ref';

@Component({
  selector: 'app-toast',
  templateUrl: './toast.component.html',
})
export class ToastComponent implements OnInit, OnDestr
  iconType: string;

  private intervalId: number;

  constructor(readonly data: ToastData, readonly ref:
    this.iconType = data.type === 'success' ? 'done' :
  }

  ngOnInit() {
    this.intervalId = setTimeout(() => this.close(), 5
```

Awesome! Let's make some changes to correctly show toasts one beneath the other.

Multiple toast messages

Since we want to show the next toast relative to the last one that was shown, it will be useful to remember the last one created. So let's create a private variable *lastToast* and store the reference to the last toast created.

And now that we have this reference we need to add another method on it that will allow us to get the position of the toast message on the screen:

```
getPosition() {
    return
this.overlay.overlayElement.getBoundingClientRect();
}
```

We have an instance of the overlay class and it has the `overlayElement` property that is the actual HTML element. From that, we can use the getBoundingClientRect method to get the actual

size and position of the element. We will use this to know where the last element is displayed on the screen and show the next one in the correct position.

Let's configure the toast service to show toasts where we want. When calling the create method on the overlay service we can also pass in a config object.

Conveniently the overlay service has a position method for us to use that will expose in a fluent API style all the options that we can set. For example, if we want to show our toasts on the right side of the screen, relative to the global window we can do something like:

```
const positionStrategy =
this.overlay.position().global().right();
```

And then pass this in when creating the overlay:

```
const overlayRef = this.overlay.create({
    positionStrategy,
});
```

In order to stack the toast messages below each other, we have to add a top position and compute it. We'll use the `getPosition` method we've added in the toast reference class for this:

```
getPosition() {
  const position = this.lastToast ?
     this.lastToast.getPosition().bottom : 0;

return position + 'px';
}
```

If there is a last toast, we get the bottom position from that one. If not, we use 0 or a default value. Since the position API expects a CSS style value we have to add `'px'` and convert the whole result to a string.

Adding this to the initial position creation strategy:

```
const position = this.overlay.position()
  .global()
  .right()
  .top(this.getPosition())
```

Cool! All the toast messages show up nicely and in order. 👍

Dynamic content

For sure there will be some cases where a simple text message is not enough. Maybe we want to show an anchor tag or even a button, or something more complex in our toast. For this, we'll add support for templates.

First, we extend the `ToastData` object that we passed into our `show` method inside the toast service so that we can either pass in a text that we want to show or a template reference, optionally with a context object:

```
export class ToastData {
  type: ToastType;
  text?: string;
  template?: TemplateRef<any>;
  templateContext?: {};
```

We marked all three of text, template, and templateContext as options so we can pass in either of them. One might want to treat the case when nothing is passed in though. 😄

With this in place, we have to update the template of our toast message to account for this new template reference that might be used. In order to obtain this, we'll use the ng-template component provided by Angular:

```
<ng-container *ngIf="data.text; else templateRef">
  {{ data.message }}
</ng-container>

<ng-template #templateRef>
  <ng-container *ngTemplateOutlet="data.template; conte
```

We have a container that we show only if we have a text property. If not, we just show the template also providing the context.

And that's it. Now we can also pass template references to our toast service.

Animations

That's all good and nice, but we can make our toast message even more awesome. How? By adding animations! 😃

The Angular animations API is easy to use and you can quickly create nice effects with it. For a detailed introduction check out the official docs.

In our case, we'll add two simple fade in and fade out animations for our toast messages.

In order for this to work you need to import the BrowserAnimationsModule and use it in your application:

```
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
```

```
@NgModule({
  imports: [ BrowserAnimationsModule ]
)}
export class CoreModule {}
```

Then, we can go on and create some animations. I find it best to isolate different things in different files so we can create a file where we specify our animations:

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

export const toastAnimations: {
    readonly fadeToast: AnimationTriggerMetadata;
} = {
    fadeToast: trigger('fadeAnimation', [
        state('in', style({ opacity: 1 })),
        transition('void => *', [style({ opacity: 0 })
        transition(
```

If we ignore the imports, which are almost half of the code, the rest is not that complicated.

We have a `toastAnimations` object with a `fadeToast` property. We use the trigger method to specify the name of this animation and some metadata about it.

Firstly, we specify the style that we want to have in the default state, mainly that we want opacity to be 1. Then we specify the two transitions that we care about. Basically, when the element will be shown which can be represented with the `void => *` expression. And, one when the state will change from *default* to *closing*. Now we could have tried to use the reverse and express the transition as `* => void` which is also perfectly valid. Although, it would not have helped us much since the element would have already been removed, so there would not be anything to animate.

Inside the `animate` methods we specify the duration of the animation, but since we want to make this configurable we use something similar to interpolation to specify a value:

```
animate('{{ fadeIn }}ms')
```

Lastly, we have defined a union type to specify the state in which our animation can be: default or closing.

Inside our toast component we have to add the animation to the `animations` array of the component decorator and declare an `animationState` property to hold the current value:

```
import { toastAnimations, ToastAnimationState } from
'./toast-animation';

...

animations: [toastAnimations.fadeToast],

...

animationState: ToastAnimationState = 'default';
```

With this in place, we can update the template with the animation:

```
<div class="toast"
     [@fadeAnimation]="{value: animationState, params:
        { fadeIn: 1000, fadeOut: 1000 }}">
  <mat-icon>{{ iconType }}</mat-icon>
  <div>{{ data.text }}</div>
  <mat-icon (click)="close()">close</mat-icon>
```

We have to use the trigger name that we defined for our animation, in our case *fadeAnimation* and bind an object to it, similar to property binding. Inside the object we specify two properties: the value that is bound to our component property that holds the animations state, and a params property where we specify configurable values that we use inside the animation.

The only thing left to do is to update our code so that the fade out animation is also visible. Right now we just remove the element from the screen, we'll have to manually update the state property for this.

When our component was initialized we created a timer that will dispose the toast. What we want to do is start the animation instead, something like:

```
ngOnInit() {
  this.intervalId =
```

```
        setTimeout(() => this.animationState = 'closing', 5000);
    }
```

And now we handle the animation done event to actually close the toast. Add the event handler inside the HTML template:

```
(@fadeAnimation.done)="onFadeFinished($event)"
```

Create the function that will take care of this:

```
1       onFadeFinished(event: AnimationEvent) {
2           const { toState } = event;
3           const isFadeOut = (toState as ToastAnimationSta
4           const itFinished = this.animationState === 'clo
5
6           if (isFadeOut && itFinished) {
7               this.close();
```

As a bonus, you could reset the time whenever the user moves the mouse on a toast message. 😉

Global configuration

Maybe we want to configure the margin from the top/right where we show the toast message, or the fade in fade out animations time. We should do this in a simple and consistent way throughout our application.

A perfect place to set a global config would be the NgModule we created for our toast. We create a *forRoot* method and we allow the consumers to specify a config object.

Firstly we should define an interface for this toast config object, something like:

```
export interface ToastConfig {
    position?: {
        top: number;
        right: number;
    };
    animation?: {
        fadeOut: number;
```

Also we should provide some default values that we can use when setting things up in case nothing is specified:

```
export const defaultToastConfig: ToastConfig = {
    position: {
        top: 20,
        right: 20,
    },
    animation: {
        fadeOut: 2500,
```

Now back to our module file, we can implement the `forRoot` method that will accept an argument of type `ToastConfig` and add it to the `providers` array so we can use it in our service:

```
  public static forRoot(config = defaultToastConfig):
      return {
          ngModule: ToastModule,
          providers: [
              {
                  provide: TOAST_CONFIG_TOKEN,
                  useValue: { ...defaultToastConfig, .
              },
```

If we don't pass anything, we have the default value for the config. If we pass something, it might be just a part of the config that we want to overwrite, for example the positioning. So when we provide the value we destructure the default config and then overwrite any properties that were specified by the user.

Once we have this in the providers, we can inject it in the toast service constructor and inside the toast component constructor so we can use the values:

```
@Inject(TOAST_CONFIG_TOKEN) readonly toastConfig: ToastConfig
```

Then we can update the position methods to use the top and right properties from the config, and the animation inside the toast component template:

```
[@fadeAnimation]="{value: animationState,
  params: {
    fadeIn: toastConfig.animation.fadeIn,
    fadeOut: toastConfig.animation.fadeOut
  }
}"
```

This way we have a nicely setup global configuration for our toast service. This can easily be extended to add any other properties we want and will allow this service to be used inside different projects that might have slightly different needs.

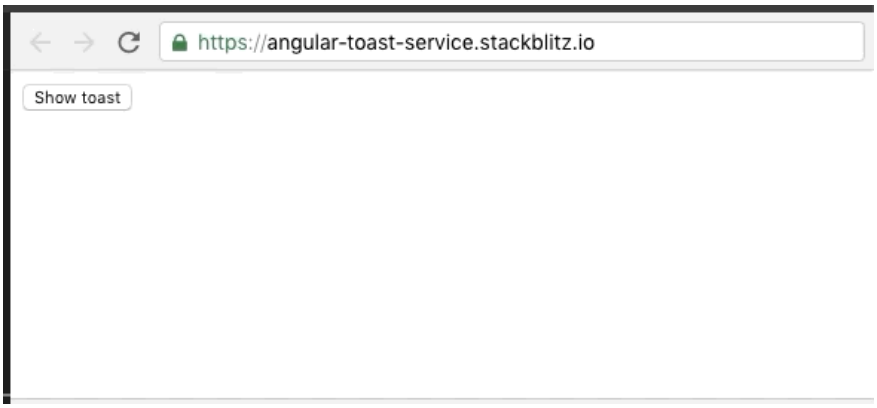
. . .

As a different and probably cleaner alternative, one can created **a single overlay** and a **container component** that will be displayed in the overlay. All the toast messages can be added inside this container component and the template will be an **ngIf** directive going over them. This can make working with positioning and animations easier. Thanks to Alex Okrushko for the feedback and suggestion.

. . .

Hope this helped and that you managed to read all the way to here 😉

This is how our simple toast messages look and behave:



You can see the **code** on StackBlitz.