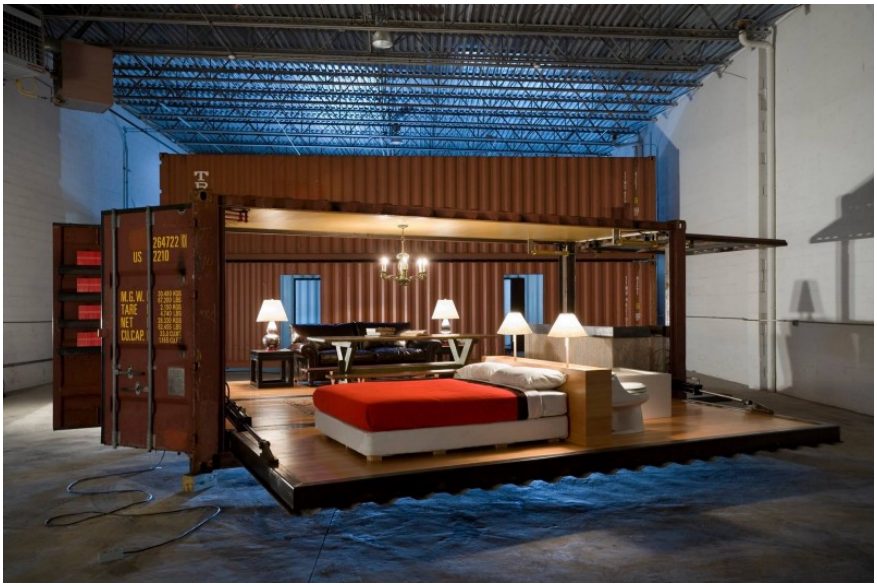
Here is how to get ViewContainerRef before @ViewChild query is evaluated

Max Koretskyi aka Wizard Follow
Jul 7, 2017 · 5 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

In one of my recent article on dynamic component instantiation Here is what you need to know about dynamic components in Angular I've shown the way how to add a child component to the parent component view dynamically. All dynamic components are inserted into a specific place in the template using `ViewContainerRef` reference. This reference is usually obtained by specifying some template reference variable in the parent component template and then using queries like `ViewChild` inside the component to get it.

Here is the quick refresher. Suppose we have our parent `App` component and we want to add a child `A` component into the specific place in the template. Here is how we do that.

A component

We're creating `A` component

```
@Component({
  selector: 'a-comp',
  template: `
      <span>I am A component</span>
  `,
})
export class AComponent {
}
```

App root module

And then register it with in the `declarations` and `entryComponents:`

```
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, AComponent],
  entryComponents: [AComponent],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

App component

And then in the parent `App` component we put the code that creates `A` component instance and inserts it

```
@Component({
  moduleId: module.id,
  selector: 'my-app',
  template: `
      <h1>I am parent App component</h1>
      <div class="insert-a-component-inside">
          <ng-container #vc></ng-container>
      </div>
  `,
})
export class AppComponent {
  @ViewChild('vc', {read: ViewContainerRef}) vc:
ViewContainerRef;
```

```
  constructor(private r: ComponentFactoryResolver) {}

  ngAfterViewInit() {
    const factory =
this.r.resolveComponentFactory(AComponent);
    this.vc.createComponent(factory);
  }
}
```

Here is the working plunker. If that's something you don't understand, I suggest you read the article I mentioned in the beginning.

This is all well and good but there's one limitation with that approach. We have to wait until the `ViewChild` query is evaluated and that happens during change detection. We can access the reference only after `ngAfterViewInit` lifecycle hook. But what if we don't want to wait until Angular runs change detection and want to have a complete component view before change detection? As it turns out we can do that using a directive instead of template reference and `ViewChild` query.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

. . .

Using directive instead of ViewChild query

Every directive can inject a reference to the `ViewContainerRef` into the constructor. This will be a reference to a view container associated with and anchored to the directive host element. So let's implement such a directive:

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

@Directive({
  selector: '[app-component-container]',
})
```

```
export class AppComponentContainer {
  constructor(vc: ViewContainerRef) {
    vc.constructor.name === "ViewContainerRef_"; // true
  }
}
```

I've added the check in the constructor to ensure that view container is available when the directive is instantiated. Now we need to use it in the `App` component template instead `#vc` template reference:

```
<div class="insert-a-component-inside">
    <ng-container app-component-container></ng-container>
</div>
```

If you run it you will see that it works fine. Great, we now know how a directive can access the view container before change detection. Now it somehow needs to pass it to the component. How can we do that? Well, a directive can inject a parent component and call a method on the component directly. However, there's a limitation in that the directive has to know the name of the parent component or use the approach described here.

A better alternative is to use a service shared between a component and its child directives and communicate through it! We can implement that service on the component directly to make it local. I'll also use a custom string token for simplicity:

```
const AppComponentService= {
  createListeners: [],
  destroyListeners: [],
  onContainerCreated(fn) {
    this.createListeners.push(fn);
  },
  onContainerDestroyed(fn) {
    this.destroyListeners.push(fn);
  },
  registerContainer(container) {
    this.createListeners.forEach((fn) => {
      fn(container);
    })
  },
  destroyContainer(container) {
    this.destroyListeners.forEach((fn) => {
      fn(container);
    })
  }
};
```

```
@Component({
  providers: [
    {
      provide: 'app-component-service',
      useValue: AppComponentService
    }
  ],
  ...
})
export class AppComponent {
```

This service simply implements primitive pub/subscribe pattern and notifies subscribes when the container is registered.

Now we can inject that service into the `AppComponentContainer` directive and register the view container:

```
export class AppComponentContainer {
  constructor(vc: ViewContainerRef, @Inject('app-component-
service') shared) {
    shared.registerContainer(vc);
  }
}
```

And the only thing that is left is to listen in the `App` component when the container is registered and use it to create a component dynamically:

```
export class AppComponent {
  vc: ViewContainerRef;

  constructor(private r: ComponentFactoryResolver,
@Inject('app-component-service') shared) {
    shared.onContainerCreated((container) => {
      this.vc = container;
      const factory =
this.r.resolveComponentFactory(AComponent);
      this.vc.createComponent(factory);
    });

    shared.onContainerDestroyed(() => {
      this.vc = undefined;
    })
  }
}
```

Here is the plunker. And that's it. You can see that we no longer need a `ViewChild` query. And if you add a `ngOnInit` lifecycle hook you will see that the `A` component is rendered before it's triggered.

. . .

RouterOutlet

If this approach seems hackish to you it is not. We need to look no further than the sources of Angular `router-outlet` directive. This directive injects `viewContainerRef` in the constructor and uses the shared service `parentContexts` to register itself and a view container within router configuration:

```
export class RouterOutlet implements OnDestroy, OnInit {
  ...
  private name: string;

  constructor(parentContexts, private location:
ViewContainerRef) {
    this.name = name || PRIMARY_OUTLET;
    parentContexts.onChildOutletCreated(this.name, this);
    ...
  }
```

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.