Having fun with Structural Directives in Angular 🏎

Christian Janker Follow
Feb 5 · 5 min read

Or: Unconventional use of structural directives

Photo by Stephanie Cook on Unsplash

This post aims to show undocumented or inconvenient usage of angular directives. They can help in numerous ways to keep your view declarative and to move side effects to the outer boundaries of your application. As a result, you'll get more readable and testable source code.

I have to admit, that Igor Minar himself does not really recommend this approach. See here. Though, if we don't call the `createEmbeddedView` method on every change of an input variable or within a loop and if we keep the context object alive I would argue that it is still an acceptable one. That said, let's start having fun :)

1 What are Structural Directives?

Short: A directive that is placed on a template element and is providing possibilities to structure the DOM with it. The most well known structural directives provided by Angular are ***ngIf** and ***ngFor**. We can recognize a structural directive by the * in front of its selector name. This is syntactic sugar provided by Angular in order to make them more convenient to use. The following two snippets are basically the same:

```
<div *ngIf="showGreeting">
   <p>Welcome, {{username}}</p>
</div>
```

is de-sugared into:

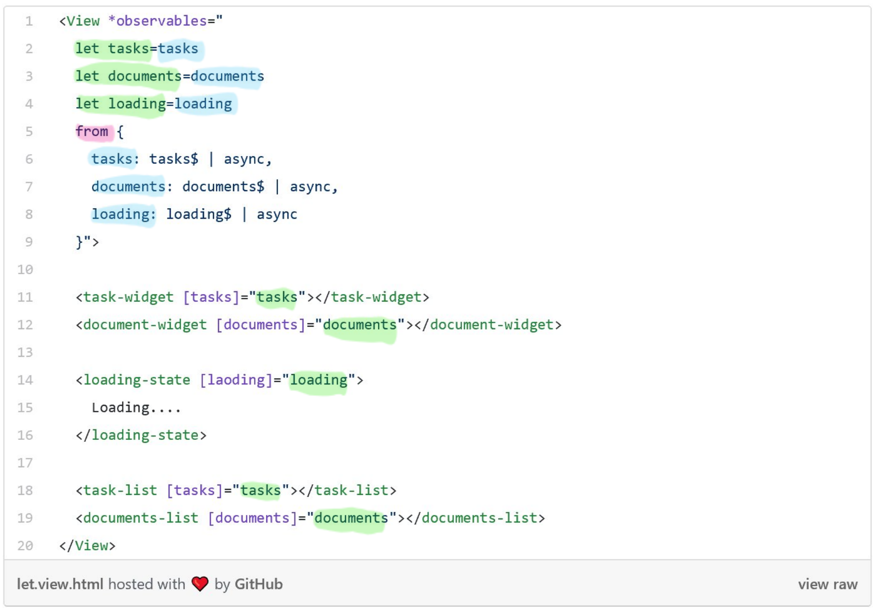
```
<ng-template [ngIf]="showGreeting">
   <div>
      <p>Welcome, {{username}}</p>
   </div>
</ng-template>
```

So, effectively a structural directive is just a directive placed on a template. When it is executed it triggers when and how the given template is rendered.

The syntactic sugar hides the existence of the template from the developer and therefore enables them to write very declarative and well understandable abstractions for their business use cases. The following directives are samples of some application specific abstractions.

2 The Observables Directive

This is an already known and used approach of handling observables **subscriptions** in Angular via structural directives. The advantage of it is that you don't have to subscribe nor to unsubscribe from observables manually. This is done automatically by the `async` pipe operator. This is already possible with the `*ngIf` directive but more often you don't need the conditional check.
This directive comes in handy especially when there would be multiple subscriptions to the same observable in one template and it works perfectly fine within a component with `OnPush` change detection. Example:



```html
<View *observables="
  let tasks=tasks
  let documents=documents
  let loading=loading
  from {
    tasks: tasks$ | async,
    documents: documents$ | async,
    loading: loading$ | async
  }">

  <task-widget [tasks]="tasks"></task-widget>
  <document-widget [documents]="documents"></document-widget>

  <loading-state [laoding]="loading">
    Loading....
  </loading-state>

  <task-list [tasks]="tasks"></task-list>
  <documents-list [documents]="documents"></documents-list>
</View>
```

let.view.html hosted with ❤ by GitHub view raw

Highlighted in green are the template input variables, which access the blue highlighted context object from the directive. The context object is defined within the directive and in our example it corresponds with the object behind the pink **from** keyword.

The directive defines a custom language with a `from` keyword after which an object is expected. In the above the `task$,` `documents$` and `loading$` are observables provided by the corresponding component of the template. We implicitly subscribe to them through the `async` pipe and feed the values into the context object of the directive. The context can then be accessed and referenced again through template input variables within the template, e.g. `let tasks=tasks`. The first part `let tasks` defines the template input variable, followed by the assignment of the variable `tasks` from the directives context object. How the context object looks like is defined in the `from` part.
Here is the source of the directive:

```
import {Directive, TemplateRef, ViewContainerRef, OnIn

export class ObservablesContext {
  [key: string]: any;
}

@Directive({
  selector: '[observables]'
})
export class ObservablesDirective implements OnInit {

  private _context = new ObservablesContext();

  // You can create a custom domain specific language
  // which maps to an Input.
  // "observablesFrom" for example can then be used li
  @Input()
  set observablesFrom(value: any) {
    Object.assign(this._context, value);
```

A very important fact to mention is that the call to `createEmbeddedView` is only done once. It’s only the context that is being changed during runtime.

Please note that the `View` component does only project its content and giving the directive a more meaningful name through its selector name.

3 Route Params Directive

This structural directive allows you to read parameters from the currently active route and pass it via inputs into your container component. Therefore your container component does not have to depend on the **ActiveRoute** directly, gaining a nice separation of concerns. Assume we have the following route configured:

```
const routes: Route[] = [
  { path: 'commits/:usernameParam', component: CommitsV
];
```

Then we can use the route params directive to extract the username parameter and pass it to the `<commit-list-view>` component:

```
<!-- commitsView.html-->
<Route *params="let username=usernameParam">
    <commit-list-view [user]="username"></commit-list-v
</Route>
```

Please note again that the purpose of the component `Route` is just to form a nice declarative language.

In the implementation of the directive, we subscribe to the current activated route and its route params. On every change we sync the params to the context object of our structural directive:

```
import {Directive, OnInit, TemplateRef, ViewContainerR
import {ActivatedRoute} from '@angular/router';
import {Subscription} from 'rxjs';

export class ParamsContext {
  [key: string]: any;
}

@Directive({
  selector: '[params]'
})
export class ParamsDirective implements OnInit, OnDest

  context = new ParamsContext();

  private _routeParamsSubscription: Subscription;

  constructor(private template: TemplateRef<ParamsCont
              private viewContainer: ViewContainerRef,
              private route: ActivatedRoute) {}

  ngOnInit() {
    this.viewContainer.createEmbeddedView(this.templat
```

4 Route Configuration Directive

With structural directives, we can do all sorts of crazy stuff. If we can read the route params declaratively, can we possibly also configure a route itself?
Yes we can, but this is a dangerous one and the implementation should be considered as a proof of concept to be able to configure your routes declaratively.

The Angular Router allows it to change its configuration at runtime, so we can extend it within a structural directive with a given template.

```
<!-- within: app.component.html -->
<router-outlet></router-outlet>

<Route *path="'foo/:usernameParam' let username=usernam
  <commits-container [username]="username" #cc>
    <commit-list [commits]="cc.commits$ | async"></comm
```

In the example above we have configured the route `foo/:usernameParam` . As with the route params directive we can access these params through the directives context object. It is important to note that this configuration is made inside of the **app.component.html** and therefore is only rendered once. Otherwise, we would configure the same route over and over again.

Let’s take a look at the implementation:

```
import {Component, Directive, Input, OnInit, TemplateR
import {ActivatedRoute, Router, Route} from '@angular/
import {take} from 'rxjs/operators';

export class RouteContext {
  [key: string]: any;
}

@Directive({
  selector: '[path]'
})
export class RouteConfigurationDirective implements On

  @Input('path') path: string;

  @Input('pathMatch') match: string;
  @Input('pathOutlet') outlet: string;

  constructor(private template: TemplateRef<RouteConte
    private router: Router) {}

  ngOnInit() {
    const config: Route = {
      path: this.path,
      component: RouterRenderComponent, // placeholder
      data: { template: this.template } // pass the te
    };

    // 'prefix' or 'full'
    if (this.match) {
      config.pathMatch = this.match;
    }

    // adress a named router outlet
    if (this.outlet) {
      config.outlet = this.outlet;
    }

    this.router.config.push(config);
  }
}

@Component({
  selector: 'router-component'
```

In the directive we inject the router and extend it with the given configuration. The RouteRenderComponent is just a placeholder component that takes the template from the routes data and renders it.
What is missing: There should be some logic to recognize already configured routes and child routes are not supported too.

5 Fetch Directive

This directive allows us to fetch data via HTTP. In this showcase only GET requests are supported. Let’s take a look at its usage:

```
<Route *params="let username=usernameParam">
  <Fetch *url="let commits from commitsUrl(username) ma
      <commit-list [commits]="commits"></commit-list>
  </Fetch>
```

In this example we use the route params directive to access the current username route param, pass it into the **commitsUrl** method and finally **map** the response to a structure we can display with the `<commit-list>` component.

```
import {HttpClient} from '@angular/common/http';
import {ComponentFactoryResolver, Directive, Input, On
import {LoadingComponent} from '../loading-spinner/loa

/**
 *
 * <Fetch *url="let commits from 'https://api.github.c
 *    <commits-list commits="commits"><commits-list>
 * </Fetch>
 */

export class UrlContext {
  $implicit: any;
}

@Directive({
  selector: '[url]'
})
export class FetchUrlDirective implements OnInit {

  context = new UrlContext();
  url: string;

  @Input()
  set urlFrom(value: string) {
    this.url = value;
    this.fetch(this.url);
  }

  @Input('urlMap') mapFn = response => response; // Fu

  @Input('urlLoadingComponent') loadingComponent: Type

  constructor(private template: TemplateRef<UrlContext
              private componentFactoryResolver: Compon
              private viewContainer: ViewContainerRef,
              private httpClient: HttpClient) {}

  ngOnInit() {}
```

Summary

Other possible directives in my head:
Connect-Redux directive

User-Role directive
Specific service call directive

I hope you enjoyed this journey as much as I did. I guess not ;) but that’s perfectly okay. My goal was to explore the possibilities of structural directives a little bit in order to learn more about them.

You can take a look at the source here. I've also managed to write some tests with Spectator. You can take a look at them here.

Have a pleasant day and keep on rocking 🤟🏾
You can follow me on Twitter 🙋‍♂️