Angular Context: Easy Data-Binding for Nested Component Trees and the Router Outlet

Levent Arman Özak Follow
Apr 12 · 7 min read

Data-binding in Angular is awesome. Really. Just decorate a public property—or setter for that matter—on a component class with `@Input` and after a quick `<app-child [prop]="parentProp"></app-child>`, voila! The property on the parent component is bound and ready to consume at will. Emitting events are just as simple: Define an `EventEmitter` decorate it with `@Output` and emit it during an internal execution. So simple, so beautiful... Implementing two-way binding is a little more complicated than this, of course. Still, its usage is fantastic: `[(ngModel)]="parentProp"`.

Yet, in time, I discovered some difficulties around this flow. First of all, the `<router-outlet>` does not offer a means to bind data to loaded components or emit events from them to parents. Second, filling a couple of intermediary components with several input properties and event emitters just to pass data through them is not only messy but also time-consuming. Your components get bloated. Besides, you end up writing lots of useless tests. Last, changing the public API of those deeply nested child components tends to become a painful practice. All parents need to be checked carefully for any traces of modified/removed properties, otherwise errors are quite likely.

There are various solutions to some or all of these problems with already available tools. A **service instance injected** in both parent and child is probably the most direct of them. However, it also is one of the most tightly coupled. Another solution may be **content projection**, but it is not always practical. Imagine a multi-layered complex component tree and you will understand what I am implying here. **Dependency injection** grants a third solution, because a parent component can be captured by navigating the DI tree. Nevertheless, the child must know either the type or the interface of the parent in order to find it and that is not always feasible. Oh yes, in order to avoid this, you can provide an injection token from the parent and it will work. Unless, of course, you are using

`ChangeDetectionStrategy.OnPush` on the child, which we all should do, and change the value passed dynamically. **State management** has its own caveat: Since connecting presentational (dumb) components directly to a specific state breaks their reusability, they have to be wrapped with container (smart) components instead and that means additional work and maintenance.

Photo by Mathew Schwartz on Unsplash

. . .

Enter Angular Context

First, a disclaimer: I am the author of **Angular Context (or ngx-context)**, which will be the main subject here on. It is developed with the problematic issues described above in mind and aims to present a solution to most, if not all, of them. The library, as the name indicates, is indeed **inspired by context in React**. Nonetheless, the implementation is completely independent of it and is actually tailored for Angular.

Well, I started looking for a globally applicable method for passing properties and events through components between top-level parents and deeply nested children. My starting point was, not unexpectedly, the amazing **dependency injection** system built in Angular. It is true that there were a few difficulties such as how to capture an unknown parent or how to sync different types of data at first, but I believe the end result is decent and mature enough to share with fellow Angular developers. So, let us take a quick look on what Angular Context can do for you and how.

Installing & Including Angular Context in Your Project

The library is hosted in npm, therefore can be installed by running the code below in your terminal:

```
npm install --save ngx-context
```

After the installation, all you have to do is import the `NgxContextModule` to your root module like this (unrelated meta data is hidden for simplicity):

```
import { NgxContextModule } from 'ngx-context';

@NgModule({
  imports: [ NgxContextModule ]
})
export class AppModule {}
```

One-way Data-binding with Angular Context

Suppose that we have to pass data from `AppComponent` to a progress bar from NgxBootstrap through an imaginary component called `OneWayComponent`. What you do with Angular Context is pretty straightforward: Place a **context provider** on `AppComponent` and a **context consumer** on the progress bar.

```
@Component({
  selector: 'my-app',
  template: `
    <context-provider
      provide="progress progressStriped progressType"
      [contextMap]="{progressType: 'type'}"
    >
      <one-way></one-way>
    </context-provider>
  `,
})
export class AppComponent {
  progress = 70;
  progressStriped = true;
  progressType = 'info';
}
```

app.component.ts hosted with ❤️ by GitHub view raw

One-way Data-binding with Angular Context #1

As you can see, we placed a `ContextProviderComponent` inside the source component, which is `AppComponent` in this case, and wrapped it around any component that may be **directly or indirectly** interested in the provided data, i.e. `OneWayComponent` here. Then, we placed a `ContextConsumerDirective` on the progress bar and after a simple mapping the result looks like this:

One-way Data-binding with Angular Context #1 Result

There are no properties so far on `OneWayComponent`, yet we have been able to render the progress bar correctly. All we had to do is give a basic key-value pair specifying which property name represents what in the context to the `contextMap` attribute. Likewise, any mapping on the provider component was effective on how we consumed the data, i.e. **referring to the mapped name instead of the original**. Now, let us increase the challenge a little bit and try to add

percentage on the bar. There is content projection involved, so we will need to declare a property for that.

```
@Component({
  selector: 'one-way',
  template: `
    <context-consumer consume="progress"></context-con

    <progressbar
      contextConsumer
      [contextMap]="{progress: 'value', progressStripe
    >{{ progress ? progress + '%' : '' }}</progressbar
  `,
```

One-way Data-binding with Angular Context #2

Here, we placed a `ContextConsumerComponent` inside the `OneWayComponent` and were able to consume a single `progress` property provided. We did not have to decorate our new property, because its value is coming from the consumer component and not from a parent. This is how the result looks like:

One-way Data-binding with Angular Context #2 Result

This was not difficult, but how about getting rid of that unnecessary property? We can do it using a `ContextDisposerDirective`, which basically is a structural directive that can be placed on an `<ng-template>` and is used to consume the provided data as template input variables.

```
@Component({
  selector: 'one-way',
  template: `
    <ng-template contextDisposer let-context>
      <progressbar
        [type]="context.type"
        [value]="context.progress"
        [striped]="context.progressStriped"
      >{{ progress ? progress + '%' : '' }}</progressb
```

One-way Data-binding with Angular Context #3

Once again, we could remove the property and the `ContextConsumerComponent` from the `OneWayComponent`. The result will be exactly as before and the middle component is kept clean.

So far so good, but how about when the data changes? Does it keep working? Does the values get updated on the consuming component? Yes, it does.

```
@Component({
  selector: 'my-app',
  template: `
    <context-provider
      provide="progress progressStriped progressType"
      [contextMap]="{progressType: 'type'}"
    >
      <one-way></one-way>
    </context-provider>
  `,
})
export class AppComponent implements OnInit {
  progress = 0;
  progressStriped = true;
  progressType = 'info';
```

One-way Data-binding with Angular Context #4

The `AppComponent` is adjusted to have `0` as initial progress and now has an interval increasing the progress value by `10` every second. You can see the result below:

One-way Data-binding with Angular Context #4 Result

. . .

Two-way Data-binding with Angular Context

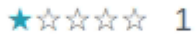
One-way binding, although helpful, is only half of the picture. We are building interactive apps and need to capture user input somehow. In Angular, the recommended way of doing that is via reactive forms, but we will get to that. For now, let us try something else with rating component from NgxBootstrap to see the worst case.

```
@Component({
  selector: 'my-app',
  template: `
    <context-provider provide="rate pre onRating">
      <two-way></two-way>
    </context-provider>

    {{ rate }}
  `,
})
export class AppComponent {
  rate = 1;

  get pre(): number {
    return this.rate;
  }

  onRating = (value: number) => {
    this.rate = value;
  }
}
```

Two-way Data-binding with Angular Context #1

In this scenario, just like the first example, data is bound to the child using the `ContextConsumerDirective`. The tricky part is, in order to make this work, one needs to know and mimic the internals of the rating component. Frankly, that would be a terrible thing to do, not because it is more labor, but rather due to the fact that any future change on the rating component has potential to break the bindings. Therefore, despite being available, **avoid this by all means**.

Take a closer look at `onRating` though. Instead of a method, a public property is called upon. The reason is the lexical scope. I have no intention to explain closures in this post. However, it is worth mentioning that, **event handlers on parents should be arrow functions**, and not regular functions or methods. Otherwise, `this` will not represent the parent component instance and it will be confusing.



Two-way Data-binding with Angular Context #1 Result

We can do better. Instead of consumer directive, `ContextConsumerComponent` can be used to **retrieve the context and use it on the template**.

```
@Component({
  selector: 'my-app',
  template: `
    <context-provider provide="rate onRating">
      <two-way></two-way>
    </context-provider>

    {{ rate }}
  `,
})
export class AppComponent {
  rate = 1;

  onRating = (value: number) => {
    this.rate = value;
  }
}
```

app.component.ts hosted with ❤️ by GitHub view raw

```
@Component({
  selector: 'two-way',
  template: `
```

Two-way Data-binding with Angular Context #2

You must have noticed: We could not do a two way binding directly with the `ngModel` directive. Instead, we established a one-way binding with it and are invoking the provided arrow function whenever model value changes. This solution, despite giving the same result and being less complicated than the last one, is still not what we wish to obtain. Next, we shall remove redundant properties from `TwoWayComponent` .

```
@Component({
  selector: 'two-way',
  template: `
    <ng-template
      contextDisposer="rate onRating"
      let-rate="rate"
      let-onRating="onRating"
    >
      <rating
        [ngModel]="rate"
        (ngModelChange)="onRating($event)"
```

Two-way Data-binding with Angular Context #3

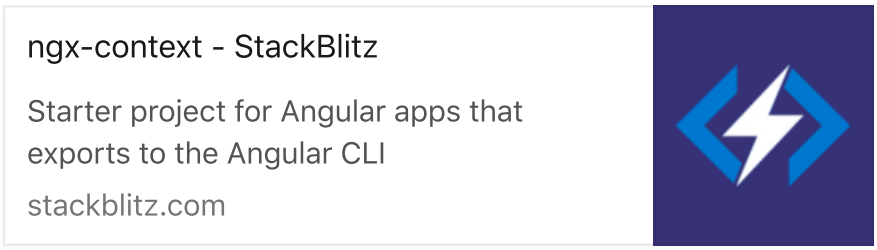
Once again, the intermediary component is free from properties and methods only good to be bound to the child component. We are getting there, yet there is one more thing to do: **Reactive form implementation**.

```
@Component({
  selector: 'my-app',
  template: `
    <form [formGroup]="form">
      <context-provider provide="rate">
        <two-way></two-way>
      </context-provider>
    </form>

    {{ rate }}
  `,
})
export class AppComponent {
  form: FormGroup;

  get rate(): AbstractControl {
    return this.form.controls.rating;
  }

  constructor(private fb: FormBuilder) {
    this.fb.group({
      rating: [1],
    });
  }
```

Two-way Data-binding with Angular Context #4

No extra properties, no event callbacks, no `ControlValueAccessor` ... This is as clean as it gets. The `NgIf` directive is necessary though, because the `control` value is imperative for `FormControlDirective` to function properly. Please keep in mind, currently the library **cannot guarantee the order** of properties to sync or be disposed and the initial value can be `undefined`.

You can find the demo application below.



. . .

Conclusion

Angular Context is not a silver bullet and I am not suggesting an extensive adoption. However, it brings an alternative approach to a common problem and can prove handy on occasion. It is quite new and requires some fine tuning. Yet, it also is well tested and production ready. Give it a try and let us know how we can improve it. Any feedback will be most welcome.

Thanks 👋

. . .

> *Come, say hello to us at Twitter. We are a group of Angular lovers from Turkey, organizing meetups, publishing articles, and contributing to open-source projects around Angular.—NG Turkey*

Follow NGTurkey on Twitter: @ngturkiye