Never again be confused when implementing ControlValueAccessor in Angular forms

Max Koretskyi aka Wizard Follow
Sep 14, 2017 · 9 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.

If you're working on a complex project inevitably you will face the situation when you have to create a custom form control. The essential component of this task will be implementing

`ControlValueAccessor` . There are some articles on the web that explain how to implement it but none provides an insight into what role this component plays in the Angular forms architecture. If you want to know not only *how* to implement it but also *why* this article is for you.

Here I'll first explain why we need `ControlValueAccessor` and how it's used inside Angular. Then I'll demonstrate how to wrap a 3rd party widget into an Angular component and setup communication with a parent component using the standard input/output mechanism. And finally I'll show how to implement `ControlValueAccessor` that introduces a new communication mechanism specifically for Angular forms.

> 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!**

FormControl and ControlValueAccessor

If you've worked with forms in Angular before you are probably familiar with FormControl. The Angular docs describe it as an entity that tracks the value and validation status of an individual form control. It's important to understand that when you work with forms, a `FormControl` is always created regardless of whether you use template driven or reactive forms. With the reactive approach, you create a control yourself explicitly and use the `formControl` or the `formControlName` directive to bind it to a native control. If you use template driven approach, the `FormControl` is created implicitly by the NgModel directive:

```
@Directive({
  selector: '[ngModel]...',
  ...
})
export class NgModel ... {
  _control = new FormControl();   <---------------- here
```

A `formControl` created implicitly or explicitly has to interact with a native form control like input or textarea. And instead of a native form control it's also possible to have a custom form control created as an Angular component. A custom form control usually wraps a control that is written using pure JavaScript like jQueryUI's slider. Throughout this article I'll be using "native form control" phrase to distinguish between the Angular specific `formControl` and a form control you use in HTML. But you should understand that instead of a native form control like `input` , any custom form control can interact with a `formControl` .

The number of native form controls is limited, but the variety of custom form controls can be potentially infinite. So, Angular needs a generic mechanism to stand between Angular's `formControl` and a native/custom form control. This is where the ControlValueAccessor object comes into play. This is the object that stands between the Angular `formControl` and a native form control and synchronizes values between the two. Here is what the docs say about it:

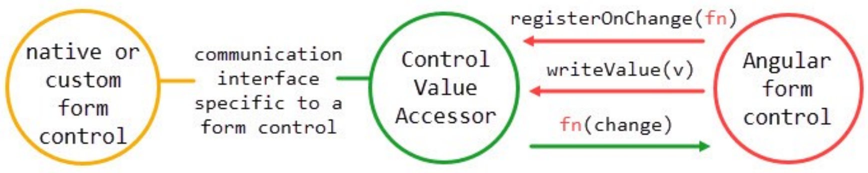
> *A* `ControlValueAccessor` *acts as a bridge between the Angular forms API and a native element in the DOM.*

Any component or directive can be turned into `ControlValueAccessor` by implementing the `ControlValueAccessor` interface and registering itself as an `NG_VALUE_ACCESSOR` provider. We will see in a minute how this can be done. Among others the interface defines two important methods— `writeValue` and `registerOnChange` :

```
interface ControlValueAccessor {
  writeValue(obj: any): void
  registerOnChange(fn: any): void
  registerOnTouched(fn: any): void
  ...
}
```

The `writeValue` method is used by `formControl` to set the value to the native form control. The `registerOnChange` method is used by `formControl` to register a **callback** that is expected to be triggered every time the native form control is updated. It is your responsibility to **pass the updated value to this callback** so that the value of respective Angular form control is updated. The `registerOnTouched` method is used to indicate that a user interacted with a control.

Here is the diagram that demonstrates an interaction:



Again, it’s important to understand that `controlValueAccessor` **always interacts with a form control** created explicitly (reactive forms) or implicitly (template driven).

Angular implements default value accessors for all standard native form elements:

```
+------------------------------------+----------------------
+
|               Accessor             |     Form Element
|
+------------------------------------+----------------------
+
| DefaultValueAccessor               | input, textarea
|
| CheckboxControlValueAccessor       | input[type=checkbox]
|
| NumberValueAccessor                | input[type=number]
|
| RadioControlValueAccessor          | input[type=radio]
|
| RangeValueAccessor                 | input[type=range]
|
| SelectControlValueAccessor         | select
|
| SelectMultipleControlValueAccessor | select[multiple]
|
```

As you can see the `DefaultValueAccessor` is used when Angular encounters `input` or `textarea` in a component template:

```
@Component({
  selector: 'my-app',
  template: `
      <input [formControl]="ctrl">
  `
})
export class AppComponent {
  ctrl = new FormControl(3);
}
```

All form directives, including the `formControl` directive used above, call the setUpControl function to setup interaction between a `formControl` and a `ControlValueAccessor`. Here is the code snippet demonstrating that for the `formControl` directive:

```
export class FormControlDirective ... {
  ...
  ngOnChanges(changes: SimpleChanges): void {
    if (this._isControlChanged(changes)) {
      setUpControl(this.form, this);
```

And here is the gist of the `setUpControl` function that shows how the native and Angular's form controls are synchronized:

```
export function setUpControl(control: FormControl, dir:
NgControl) {

  // initialize a form control
  dir.valueAccessor.writeValue(control.value);

  // setup a listener for changes on the native control
  // and set this value to form control
  dir.valueAccessor.registerOnChange((newValue: any) => {
    control.setValue(newValue, {emitModelToViewChange:
false});
  });

  // setup a listener for changes on the Angular formControl
  // and set this value to the native control
  control.registerOnChange((newValue: any, ...) => {
    dir.valueAccessor.writeValue(newValue);
  });
```

Once we understand the mechanics, we can continue implementing our own accessor for a custom form control.

. . .

Implementing widget wrapper

Since Angular provides control value accessors for all default native controls a new value accessor is most often implemented to wrap 3rd party plugins/widgets. I mentioned a slider widget from jQueryUI library earlier and this is the plugin we will use for our custom form control.

Simple wrapper

Let’s start with the most basic implementation which just wraps the widget and shows it on the screen. To do that we implement a new `NgxJquerySliderComponent` and use a DOM element from its template to render the slider:

```
@Component({
  selector: 'ngx-jquery-slider',
  template: `
      <div #location></div>
  `,
  styles: ['div {width: 100px}']
})
export class NgxJquerySliderComponent {
  @ViewChild('location') location;
  widget;

  ngOnInit() {
    this.widget = $(this.location.nativeElement).slider()
  }
}
```

Here we create a slider widget on the native DOM element using standard jQuery approach. Then we save the reference to the widget into the `widget` property.

Once we have our wrapper component ready we can use it in the parent `App` component like this:

```
@Component({
  selector: 'my-app',
  template: `
      <h1>Hello {{name}}</h1>
      <ngx-jquery-slider></ngx-jquery-slider>
  `
})
export class AppComponent { ... }
```

To run the application we need to include the jQuery related dependencies. For simplicity we will add them as global dependencies into `index.html` :

```
<script
    src="https://code.jquery.com/jquery-3.2.1.js">
</script>
```

```
<script
    src="https://code.jquery.com/ui/1.12.1/jquery-ui.js">
</script>
<link
    rel="stylesheet"
href="//code.jquery.com/ui/1.12.1/themes/smoothness/jquery-
ui.css">
```

Here is the application that demonstrates the setup.

Interactive form control

With the above implementation our custom slider control doesn't have any way to interact with the parent component. So let's use standard input/output mechanisms as communication channels:

```
export class NgxJquerySliderComponent {
  @ViewChild('location') location;
  @Input() value;
  @Output() private valueChange = new EventEmitter();
  widget;

  ngOnInit() {
    this.widget = $(this.location.nativeElement).slider();
    this.widget.slider('value', this.value);
    this.widget.on('slidestop', (event, ui) => {
      this.valueChange.emit(ui.value);
    });
  }

  ngOnChanges() {
    if (this.widget && this.widget.slider('value') !==
this.value) {
      this.widget.slider('value', this.value);
    }
  }
}
```

Once the slider widget is created we subscribe to its value changes using `slidestop` event. Once the event is triggered we notify the parent using `valueChanges` output event emitter. And we also track changes to the input `value` binding using `ngOnChanges` lifecycle hook and once the value is updated we set it into the slider widget.

And here is how we use the component now in parent `App` component:

```
<ngx-jquery-slider
    [value]="sliderValue"
    (valueChange)="onSliderValueChange($event)">
</ngx-jquery-slider>
```

Here is the application that demonstrates the setup.

However, if we want to use our slider as part of a form and communicate with it using template driven or reactive form directives we need to implement a value accessor. And we don't need standard input/output communication mechanism so we will remove it when implementing the value accessor.

Implementing custom value accessor

Implementing a custom value accessor is not difficult. It requires 2 simple steps:

1. registering a `NG_VALUE_ACCESSOR` provider
2. implementing `ControlValueAccessor` interface methods

`NG_VALUE_ACCESSOR` provider specifies a class that implements `ControlValueAccessor` interface and is used by Angular to setup synchronization with `formControl`. It's usually the class of the component or directive that registers the provider. All form directives inject value accessors using the token `NG_VALUE_ACCESSOR` and then select a suitable accessor. If there is an accessor which is not built-in or `DefaultValueAccessor` it is selected. Otherwise Angular picks the default accessor if it's provided. And there can be no more than one custom accessor defined for an element.

So let's first define the provider:

```
@Component({
  selector: 'ngx-jquery-slider',
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: NgxJquerySliderComponent,
    multi: true
  }]
  ...
})
class NgxJquerySliderComponent implements
ControlValueAccessor {...}
```

We specified the class directly in component decorator descriptor. However, all default accessors implemented by Angular define a provide outside the class metadata like this:

```
export const DEFAULT_VALUE_ACCESSOR: any = {
  provide: NG_VALUE_ACCESSOR,
  useExisting: forwardRef(() => DefaultValueAccessor),
  multi: true
};

@Directive({
  selector:'input',
  providers: [DEFAULT_VALUE_ACCESSOR]
  ...
})
export class DefaultValueAccessor implements
ControlValueAccessor {}
```

and so they need to use `forwardRef`. To learn more about `forwardRef` read What is `forwardRef` in Angular and why we need it. When implementing a custom `controlValueAccessor` I recommend specifying a class directly in the decorator descriptor.

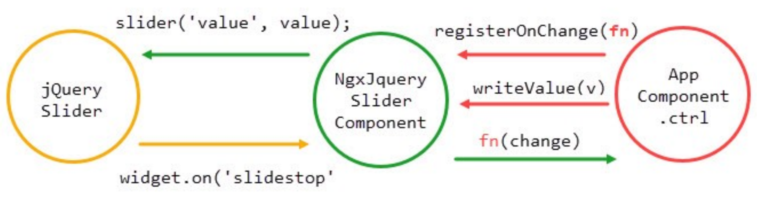
Once we defined a provider let's implement `ControlValueAccessor` interface:

```
export class NgxJquerySliderComponent implements
ControlValueAccessor {
  @ViewChild('location') location;
  widget;
  onChange;
  value;

ngOnInit() {
    this.widget =
$(this.location.nativeElement).slider(this.value);

this.widget.on('slidestop', (event, ui) => {
      this.onChange(ui.value);
    });
  }


writeValue(value) {
    this.value = value;
    if (this.widget && value) {
      this.widget.slider('value', value);
    }
  }

registerOnChange(fn) { this.onChange = fn;  }
```

```
registerOnTouched(fn) {  }
```

We are not interested in learning whether user interacted with a control or not so we leave `registerOnTouched` empty. Inside `registerOnChange` we simply save the reference to the callback `fn` function passed by `formControl`. We will trigger it every time the change in the slider value occurs. And inside the `writeValue` method we set the value to the slider widget.

So now if we depict the above functionality to the interaction picture it looks something like this:



If you compare two implementations as a simple wrapper and as a `controlValueAccessor` you should see the interaction with a parent component is different, while the interaction with the underlining slider widget is the same. You may also notice that `formControl` actually simplifies the interaction with the parent component. We use `writeValue` where we used `ngOnChanges` in the simple wrapper and call `this.onChange` where we emitted value before with `this.valueChange.emit(ui.value)`.

The custom slider control implemented as `ControlValueAccessor` can now be used like this:

```
@Component({
  selector: 'my-app',
  template: `
      <h1>Hello {{name}}</h1>
      <span>Current slider value: {{ctrl.value}}</span>
      <ngx-jquery-slider [formControl]="ctrl"></ngx-jquery-
slider>
      <input [value]="ctrl.value"
(change)="updateSlider($event)">
  `
})
export class AppComponent {
  ctrl = new FormControl(11);

  updateSlider($event) {
    this.ctrl.setValue($event.currentTarget.value,
{emitModelToViewChange: true});
  }
}
```

You can find the final implementation here.

Github

Final project is also available on github here.

. . .

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.