Table Of Contents

Note: see details at the end of this book for getting the Free Angular For Beginners Course, plus a bonus content Typescript Video List.

- But what happened to ngModel?
- Advantages and disadvantages of Model Driven Forms
- Functional Reactive Programming in Angular
- Advantages of building UIs using Functional Reactive Programming (FRP)
- Updating Form Values
- How To Reset a Form
- Model-Driven vs Template Driven: can they be mixed?

Section 3 - Section 3 - Which form type to choose, and why?

- Which form type to choose?

Section 4 - Final Thoughts & Bonus Content

- Final Thoughts
- Bonus Content - Typescript - A Video List
- Bonus Content - Angular For Beginners Course

Angular Forms Jumpstart

Introduction

Welcome to the Angular Forms Jumpstart Book, thank you for joining! Like the title says, the goal of the book is to get you proficient quickly in both Model-Driven and Template-Driven Angular Forms.

In this book, we will see how the Angular Forms API works and how it can be used to build complex forms. We will go through the following topics:

- What is Angular Forms all about
- Template Driven Forms, or the Angular 1 way
- Model Driven Forms, or the new Functional Reactive way
- Updating Form Values, How To Reset a Form
- Advantages and disadvantages of both form types
- Can and should the two be used together?
- Which one to choose by default?

So without further ado, let's get started!

I hope you will enjoy this book, please send me your feedback at admin@angular-university.io.

Introduction To Angular Forms

A large category of frontend applications are very form-intensive, especially in the case of enterprise development. Many of these applications are basically just huge forms, spanning multiple tabs and dialogs and with non-trivial validation business logic.

Every form-intensive application has to provide answers for the following problems:

- how to keep track of the global form state
- know which parts of the form are valid and which are still invalid
- properly displaying error messages to the user so that the user knows what to do to make the form valid

All of these are non-trivial tasks that are similar across applications, and as such could benefit from a framework.

The Angular framework provides us a couple of alternative strategies for handling forms: Let's start with the option that is the closest to Angular 1.

Section 1 - Angular Template-Driven Forms

Angular 1 tackled Forms via the famous `ng-model` directive. The instantaneous two-way data binding of `ng-model` in Angular 1 is really a life-saver as it allows to transparently keep in sync a form with a view model.

Forms built with this directive can only be tested in an end to end test because this requires the presence of a DOM, but still, this mechanism is very useful and simple to understand.

Angular now provides an identical mechanism named also `ngModel`, that allow us to build what is now called Template-Driven forms.

> *Note that `NgModel` includes all of the functionality of its Angular 1 counterpart.*

Enabling Template Driven Forms

Unlike the case of AngularJs, `ngModel` and other form-related directives are not available by default, we need to explicitly import them in our application module:

```
import {NgModule} from "@angular/core";
import {platformBrowserDynamic} from "@angular/platform-browser-dynamic";
import {BrowserModule} from "@angular/platform-browser";
import {FormsModule} from "@angular/forms";

@Component({...})
export class App { }

@NgModule({
    declarations: [App],
    imports: [BrowserModule, FormsModule],
    bootstrap: [App]
})
export class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);

```

We can see here that we have enabled Template Driven Forms by adding `FormsModule` to our application, and bootstrapped the application.

With this initial configuration in place, let's now build our first Angular Form.

Our First Template Driven Form

Let's take a look at a form built according to the template driven way:

```
<section class="sample-app-content">
    <h1>Template-driven Form Example:</h1>
    <form #f="ngForm" (ngSubmit)="onSubmitTemplateBased()">
        <p>
            <label>First Name:</label>
            <input type="text"
                [(ngModel)]="user.firstName" required>
        </p>
        <p>
            <label>Password:</label>
            <input type="password"
                [(ngModel)]="user.password" required>
        </p>
        <p>
            <button type="submit" [disabled]="!f.valid">Submit</button>
        </p>
    </form>
</section>

```

There is actually quite a lot going on in this simple example. What we have done here is to declare a simple form with two controls: first name and password, both of which are mandatory fields (marked with the `required` attribute).

The form will trigger the controller method `onSubmitTemplateBased` on submission, but the submit button is only enabled if both required fields are filled in.

But that is only a small part of what is going on here.

NgModel Validation Functionality

Notice the use of `[(ngModel)]`, this notation emphasizes that the two form controls are bi-directionally bound with a view model variable, named as simply `user`.

> *This `[(ngModel)]` syntax is known as the 'Box of Bananas' syntax :-)*

More than that, when the user clicks a required field, the field is shown in red until the user types in something. Angular is actually tracking three form field states for us and applying the following CSS classes to both the form and its controls:

- touched or untouched
- valid or invalid
- pristine or dirty

These CSS state classes are very useful for styling form error states.

Angular is actually tracking the validity state of the whole form as well, using it to enable/disable the submit button. This functionality is actually common to both template-driven and model-driven forms.

The logic for all this must be in the controller, right?

Let's take a look at the controller associated with this view to see how all this form logic is implemented:

```
@Component({
    selector: "template-driven-form",
    templateUrl: 'template-driven-form.html'
})
export class TemplateDrivenForm {

    user: Object = {};

    onSubmitTemplateBased() {
        console.log(this.vm);
    }

}

```

Not much to see here! We only have a declaration for a view model object `user`, and an event handler used by `ngSubmit`.

All the very useful functionality of tracking form errors and registering validators is taken care for us without any special configuration!

How does Angular pull this off then?

The way that this works, is that there is a set of implicitly defined form directives that are being applied to the view. Angular will automatically apply a form-level directive to the form in a transparent way, creating a `FormGroup` and linking it to the form.

If by some reason you don't want this you can always disable this functionality by adding `ngNoForm` as a form attribute.

Furthermore, each input will also get applied a directive that will register itself with the control group, and validators are registered if elements like `required` or `maxlength` are applied to the input.

The presence of `[(ngModel)]` will also create a bidirectional binding between the form and the user model, so in the end there is not much more to do at the level of the controller.

This is why this is called template-driven forms, because both validation and binding are all setup in a declarative way at the level of the template.

What if we don't need bi-directional binding, but only field initialization?

Sometimes we just want to create a form and initialize it, but not necessarily do bi-directional binding. We could want to let the user fill in the form and press submit, and only then get the latest value. We can do this by using the plain `[ngModel]` syntax:

```
<p>
    <label>First Name:</label>
    <input type="text"  [ngModel]="user.firstName" required>
</p>
<p>
    <label>Password:</label>
    <input type="password"  [ngModel]="user.password" required>
</p>

```

This will allow us to initialize the form by filling in the fields of the user object:

```
user = {
    firstName: 'John',
    password:  'test'
};

```

What if we don't need field initialization, can we still get validation?

For example, creation forms don't need initial values, they only need validation. If we want to get only the validation functionality of `ngModel` without neither the initialization of values or the bi-directional binding, we can do so with the following syntax:

```
<p>
      <label>First Name:</label>
      <input type="text" name="firstName" required ngModel>
</p>
<p>
      <label>Password:</label>
      <input type="password" name="password" required ngModel>
</p>
```

Advantages and Disadvantages of Template Driven Forms

In this simple example we cannot really see it, but keeping the template as the source of all form validation truth is something that can become pretty hard to read rather quickly.

As we add more and more validator tags to a field or when we start adding complex cross-field validations the readability of the form decreases, to the point where it will be harder to hand it off to a web designer.

The upside of this way of handling forms is its simplicity, and it's probably more than enough to build a large range of forms.

On the downside, the form validation logic cannot be unit tested. The only way to test this logic is to run an end to end test with a browser, for

example using a headless browser like `PhantomJs`.

Template Driven Forms from a functional programming point of view

There is nothing wrong with template driven forms, but from a programming technique point of view bi-directional binding is a solution that promotes mutability.

Each form has a state that can be updated by many different interactions and it's up to the application developer to manage that state and prevent it from getting corrupted. This can get hard to do for very large forms and can introduce a category of potential bugs.

Again it's important to realize that this only happens in very large/complex forms. Angular does provide a different alternative for managing forms, so let's go through it.

Section 2 - Model Driven Or Reactive Forms

A model driven form looks on the surface pretty much like a template driven form. But in order to be able to create this type of forms, we need first to import a different module into our application:

```
1  import {NgModule} from "@angular/core";
2  import {ReactiveFormsModule} from "@angular/forms";
3
4  @Component({...})
5  export class App { }
6
```

```
@NgModule({
    declarations: [App],
    imports: [BrowserModule, ReactiveFormsModule],
    bootstrap: [App]
})
export class AppModule {}

```

Note that here we imported `ReactiveFormsModule` instead of `FormsModule`. This will load the reactive forms directives instead of the template driven directives.

If we find ourselves in a situation where we would happen to need both, then we should import both modules, more on this later.

Our First Reactive Form

Let's take our previous example and re-write it but this time in reactive style:

```
<section class="sample-app-content">
    <h1>Model-based Form Example:</h1>
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
        <p>
            <label>First Name:</label>
            <input type="text" formControlName="firstName">
        </p>
        <p>
            <label>Password:</label>
            <input type="password" formControlName="password">
        </p>
        <p>
            <button type="submit" [disabled]="!form.valid">Submit</button>
        </p>
    </form>
</section>

```

There are a couple of differences here. First, there is a `formGroup` directive applied to the whole form, binding it to a controller variable named `form`.

Notice also that the `required` validator attribute is not applied to the form controls. This means the validation logic must be somewhere in the controller, where it can be unit tested.

What does the controller look like?

There is a bit more going on in the controller of a Model Driven Form, let's take a look at the controller for the form above:

```
import { FormGroup, FormControl, Validators, FormBuilder }
    from '@angular/forms';

@Component({
    selector: "model-driven-form",
    templateUrl: 'model-driven-form.html'
})
export class ModelDrivenForm {
    form: FormGroup;

    firstName = new FormControl("", Validators.required);

    constructor(fb: FormBuilder) {
        this.form = fb.group({
            "firstName": this.firstName,
            "password":["", Validators.required]
        });
    }
    onSubmitModelBased() {
        console.log("model-based form submitted");
        console.log(this.form);
    }
}

```

We can see that the form is really just a `FormControl`, which keeps track of the global validity state. The controls themselves can either be instantiated individually or defined using a simplified array notation using the form builder.

In the array notation, the first element of the array is the initial value of the control, and the remaining elements are the control's validators. In this case both controls are made mandatory via the `Validators.required` built-in validator.

But what happened to `ngModel`?

Note that `ngModel` can still be used with model driven forms. It's just that the form value would be available in two different places: the view model and the `FormGroup`, which could potentially lead to some confusion.

Due to this reason mixing `ngModel` with reactive forms is best avoided.

Advantages and disadvantages of Model Driven Forms

You are probably wondering what we gained here. On the surface there is already a big gain: We can now unit test the form *validation* logic.

We can do that just by instantiating the class, setting some values in the form controls and perform assertions against the form global valid state and the validity state of each control.

But this is just one possibility. The `FormGroup` and `FormControl` classes provide an API that allows us to build UIs using a

completely different programming style known as Functional Reactive Programming.

Functional Reactive Programming in Angular

The form controls and the form itself now provide an Observable-based API. You can think of observables simply as a collection of values over time.

This means that both the controls and the whole form itself can be viewed as a continuous stream of values, that can be subscribed to and processed using commonly used functional primitives.

For example, it's possible to subscribe to the Form stream of values using the Observable API like this:

```
this.form.valueChanges
        .map((value) => {
            value.firstName = value.firstName.toUpperCase();
            return value;
        })
        .filter((value) => this.form.valid)
        .subscribe((value) => {
           console.log("Model Driven Form valid value: vm = ",
                       JSON.stringify(value));
        });


```

What we are doing here is taking the stream of form values (that changes each time the user types in an input field), and then apply to it some commonly used functional programming operators: `map` and `filter`.

In fact, the form stream provides the whole range of functional operators available in `Array` and many more.

In this case, we are converting the first name to uppercase using `map` and taking only the valid form values using `filter`. This creates a new stream of **valid-only** values to which we subscribe, by providing a callback that defines how the UI should react to a new valid value.

Advantages of building UIs using Functional Reactive Programming (FRP)

We are not obliged to use FRP techniques with Angular Model Driven Forms. Simply using them to make the templates cleaner and allow for component unit testing is already a big plus.

But like its usual in the case of Observable-based APIs, FRP techniques can help easily implement many use cases that would otherwise be rather hard to implement such as:

- pre-save the form in the background at each valid state, or even invalid (for example storing the invalid value in a cookie for later use)
- typical desktop features like undo/redo

Have a look at this video Introduction to Functional Reactive Programming - Using the Async Pipe - Pitfalls to Avoid, part of the Angular Services and HTTP course for more on Functional Reactive Programming in Angular.

Updating Form Values

We now have APIs available for either updating the whole form, or just a couple of fields. For example, lets create a couple of new buttons on the model driven form above:

```
<p>
    <button type="submit" [disabled]="!form.valid">Submit</button>
    <button (click)="partialUpdate()">Partial Update</button>
    <button (click)="fullUpdate()">Full Update</button>
    <button (click)="reset()">Cancel</button>
</p>

```

We can see here that there are two buttons for updating the form value, one for the partial updates and the other for full updates. This is how the corresponding component methods look like:

```
fullUpdate() {
    this.form.patchValue({firstName: 'Partial', password: 'monkey'});
}

partialUpdate() {
    this.form.patchValue({firstName: 'Partial'});
}

```

We can see that `FormGroup` provides two API methods for updating form values:

- we have `patchValue()` which partially updates the form. This method does not need to receive values for all fields of the form, which allows for partial updates
- there is also `setValue()`, to which we are passing all the values of the form. In the case of this method, values for all

> form fields need to be provided, otherwise, we will get an error message saying that certain field values are missing

We might think that we could use these same APIs to reset the form by passing blank values to all fields.

That would not work as intended because the pristine and untouched statuses of the form and its fields would not get reset accordingly. But Angular Final provides an API for this use case.

How To Reset a Form

Notice on the button list above that there is a cancel button that calls a `reset()` method, which does reset everything back to pristine and untouched:

```
reset() {
    this.form.reset();
}
```

Let's now see if its possible to mix both form types and if that is advisable.

Model-Driven vs Template Driven: can they be mixed?

Model-Driven and Template-Driven under the hood are implemented in the same way: there is a `FormGroup` for the whole form, and one `FormControl` instance per each individual control.

If by some reason we would need to, we could mix and match the two ways of building forms, for example:

- we can use `ngModel` to read the data and use `FormBuilder` for the validations. we don't have to subscribe to the form or use RxJs if we don't wish to.
- We can declare a control in the controller, and then reference it in the template to obtain its validity state

But in general it's better to choose one of the two ways of doing forms, and using it consistently throughout the application.

Section 3 - Which form type to choose, and why?

Template Driven Forms are maybe for simple forms slightly less verbose, but the difference is not significant. Reactive Forms are actually much more powerful and have a nearly equivalent readability.

Most likely in a large-scale application, we will end up needing the functionality of reactive driven forms for implementing more advanced use cases like for example auto-save.

Everything can be done in both form types, but some things are simpler using reactive forms

It's not this that there is functionality that cannot be implemented with template driven forms. But there is a lot of functionality especially more modern form features like for example auto-save that can be implemented in just a few lines of RxJs.

Which form type to choose?

Are you migrating an Angular 1 application into Angular? That is the ideal scenario for using Template Driven Forms.

Or are you building a new application from scratch? Reactive forms are a good default choice because more complex validation logic is actually simpler to implement using them.

For example, imagine a validation that requires to inspect two fields and compare them: for example a password field and a password confirmation field need to be identical.

With model-driven forms, we just need to write a function and plug it into the `FormControl`. With template driven forms, there is more to it: we need to define a directive and somehow pass it the value of the two fields.

Model-driven forms seem great for example for enterprise applications with lots of complex inter-field business validation logic.

As mentioned before, we want to avoid situations where we are using both form types together, as it can get rather confusing. But it's still possible to use both forms together if by some reason we really need to.

Section 4 - Final Thoughts & Bonus Content

Final Thoughts

As we have seen, Angular provides us two ways to build forms: Template Driven and Model Driven.

The Template Driven approach is very familiar to Angular 1 developers and is ideal for easy migration of Angular 1 applications into Angular.

The Model Driven approach removes validation logic from the template, keeping the templates clean of validation logic. But also allows for a whole different way of building UIs that we can optionally use.

This is not an exclusive choice but for a matter of consistency, it's better to choose one of the two approaches and use it everywhere in our application.

I hope this helps get the most out of Angular Forms! If you have any questions about the book, any issues or comments I would love to hear from you at admin@angunar-university.io

I invite you to have a look at the bonus material below, I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 - Top 4 Advantages of Typescript - Why Typescript?
- Video 2 - ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 - Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related
- Video 4 - Debugging Typescript in the Browser and a Node Server - Step By Step Instructions
- Video 5 - Build Type Safe Programs Without Classes Using Typescript
- Video 6 - The Typescript Any Type - How Does It Really Work?
- Video 7 - Typescript @types - Installing Type Definitions For 3rd Party Libraries
- Video 8 - Typescript Non-Nullable Types - Avoiding null and undefined Bugs

- Video 9 - Typescript Union and Intersection Types- Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing

Angular For Beginners Course

If you are looking to learn Angular, have a look at this free 2h introductory course.

The Angular Tutorial for Beginners is an over 2 hours course that is aimed at getting a complete beginner in Angular comfortable with the main notions of the core parts of the framework:

Click Here To View The Angular For The Beginners Course

The other Angular courses on the same website have about 25% of its content free as well, have a look and enjoy the videos.