**What are Reactive Forms?**

**Reactive forms**are a type of form that uses explicit and immutable approaches to managing the state of a form at a given point in time. This means that Reactive forms provide a way to ensure the integrity of your data between changes.

Reactive forms are a form of dynamic form validation, where inputs and values are provided as streams of input values that can be accessed synchronously.

Reactive forms make it easy to test because you are assured that your data is consistent and predictable. Any consumers of the streams have access to manipulate that data safely.

Reactive forms are a distinct type of form from template-driven forms because they provide:

* synchronous access to the data model
* immutability with observable operators
* change tracking through observable streams

In contrast to reactive forms, template-driven forms allow direct access to modify data in your template. They rely on directives embedded in the template as well as mutable data to track changes asynchronously.

**Creating forms in Angular**

When it comes to fomrs, Angular is often overlooked. Yet, there are two different approaches in Angular to build a form.

The first category is **Template-Driven Forms**, where you can use HTML syntax. The second one is called **Reactive Forms** and it’s based on **Observable** and **RxJS** library.

Its structure is defined inside the Component class instead of the Template file definition:

*RxJS is a library with a clear API to work with both asynchronous and synchronous code thanks to pipeline operators and the usage of the Observable concept.*

Template-Driven Forms represent the first approach we learn as developers, and often it is the only one we use inside our applications. After all, Angular Reactive Forms are a bit harder to understand, and Template Forms can handle almost everything. So, why bother?

The syntax is very much like that of the HTML Forms, and this gives us a good reason to prefer it. Template-driven forms are driven by code in the template. This means that you will see directives like ngModel and ngForm that you can use to basically have the form built for us; they do the dirty work behind its creation.

Reactive forms are also known as *model-driven forms*, where it is explicitly defined in the component class and the HTML content changes depending on the code in the component.

Look at the following template-driven approach:

<form #form="ngForm" (ngSubmit)="addUser(form.value)">

<img class="spinner" \*ngIf="usernameRef.pending">

<input type="text"

name="username"

[ngModel]

        #usernameRef="ngModel"

required

        UsernameAsyncValidator

    >

<button [disabled]="form.invalid || form.pending">Add User</button>

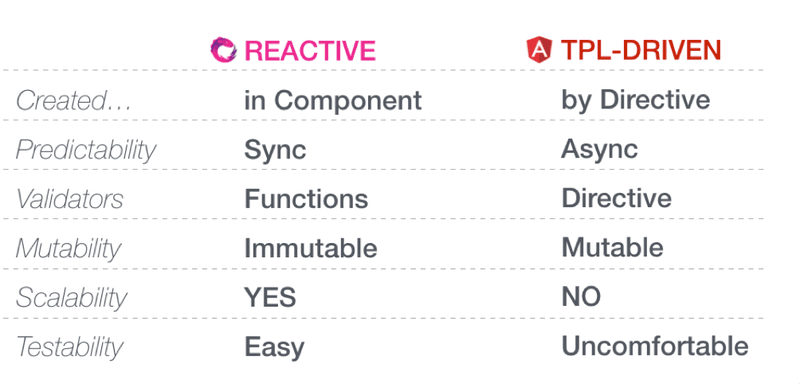
</form>

You can write and handle an Angular form without writing any single line of JavaScript code. All you need is some directives and built-in validators such as “required” or “max-length” or “min-length” for the form validation.

Or else you can write your own **custom validator** like “UsernameAsyncValidator” – like in the example above.

**Angular Reactive Form Vs. TPL-Driven Form**

Let’s do a comparison between Reactive Forms and TPL-Driven Forms with a simple table:

A comparison between reactive and template driven forms

As you probably know, Angular is organised in Modules. So, in order to use Template-Driven Form or Reactive Form, you need to import the FormsModule for the first one and the ReactiveFormsModule for the other one – both from the ‘@angular/forms’ package.

There are many differences between Reactive and Template-Driven Forms, the main one being that Reactive is **synchronous** while TPL form is **asynchronous**.

Other reasons you may want to consider to use Reactive Forms are:

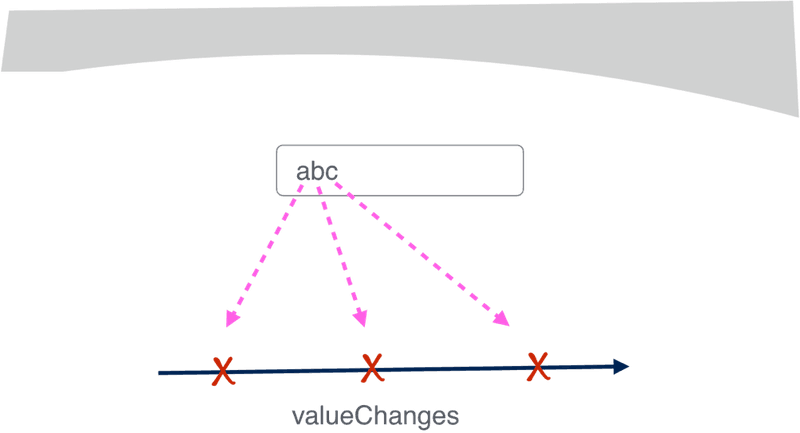
1. Template-Driven Forms are Asynchronous because of ngForm and ngModel: these directives take more than one cycle to build the entire control tree, meaning that you must wait for a tick before manipulating any of the controls from within the component class
2. Custom validators in TDFs need more work
3. As a template grows, it becomes harder to understand its structure and readability, so Reactive forms are more scalable than TDFs
4. Although Template-Driven forms are easier to create, they become a challenge when you want to do unit testing, because testing requires the presence of a *DOM*;
5. Reactive Forms are built with immutable data while Template-Driven forms promote the use of bidirectional binding

**Reactive Forms**

To appreciate Reactive Forms you must understand how they handle complex ones with a lot of validation.

Every control in the form (select, input, checkbox, etc.) has a valueChanges property, an Observable that we can subscribe to to observe changes over time and transform values with RxJS operators in a functional way.

Therefore, a form is treated like a*stream of data* where every field acts like an Observable.

A Control Form with the property valueChanges

Let’s look at an example of a Reactive Form:

import { Component, OnInit } from '@angular/core';

import { FormControl } from '@angular/forms';

@Component({

    selector: 'app-login-form',

    template: `

    <input type="text" [formControl]="name">

  `

)

export class LoginComponent implements OnInit {

  name = new FormControl('name');

  ngOnInit() {

this.name.valueChanges.subscribe(data => console.log(data));

}

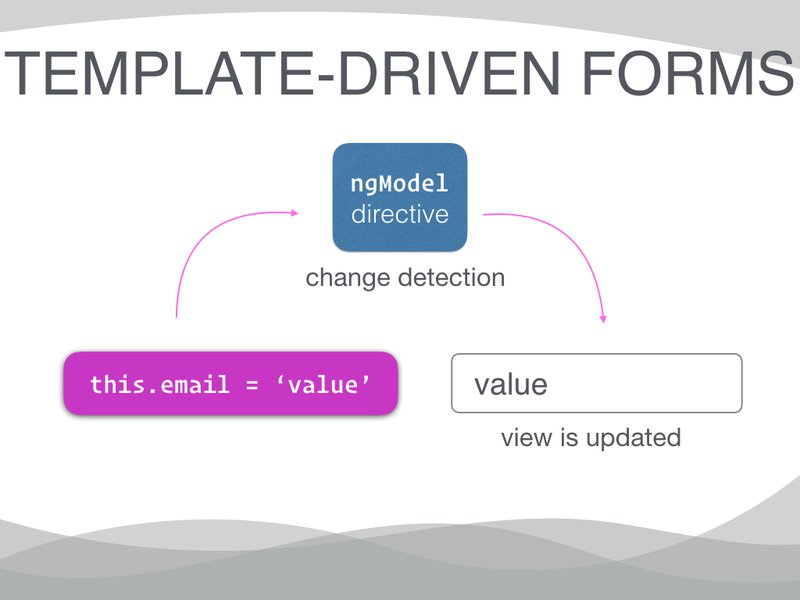
}

As you can see, there is a property being declared as **FormControl**. You can initialise it with an arbitrary value and link it to the input form from the template. This way, they are always in sync.

You can observe that the value changes made in the template with an explicit subscription to the valueChange property, or use the setValue method to change the value in the component and reflect it into the template. It acts like a bidirectional data binding.

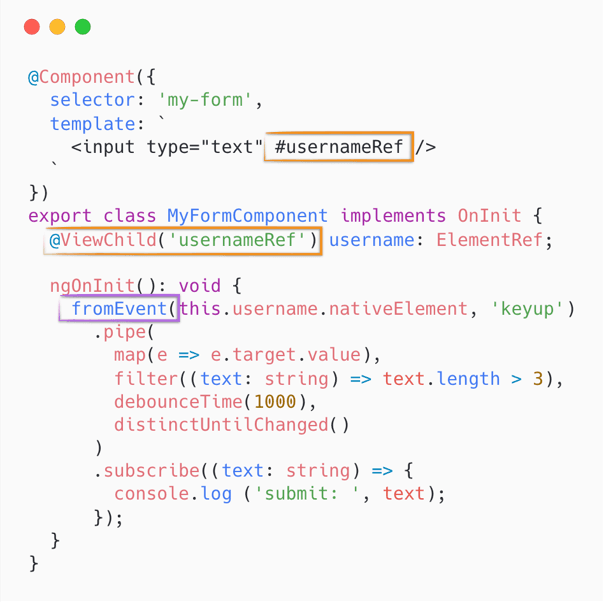
In a Template-Driven Form, every form element is connected to the *Model property* (inside the Component logic) with a ngModel directive. This is the reason why TPL Forms are asynchronous.

The directive ngModel is connected to the “change detection application life-cycle”, so the value between the HTML element and the component is updated after a “tick” of the event loop.

Template Driven Form with ngModel Directive

Thanks to the RxJS library, you can use the *fromEvent method*to Observe and subscribe to an event.

Using the Angular *ViewChild*decorator and without using Reactive Forms, you can take the reference of an element of the DOM (*e.g.* a getElementById) and take advantage of RxJS operators to manipulate the data inside an input element:

Using Viewchild decorator to access an element

Of course, you need to know a bit of RxJS – at least the basic operators like *map, filter, take, debounceTime and distinctUntilChanged*.

Until now, we have not used the basic constructor of Reactive Forms from FormBuilder like FormControl, FormGroup and FormArray. We can use them with FormBuilder or alone, like in the previous example.

**Scale your forms**

The next example is a bit harder, because we introduce FormGroup and FormBuilder to construct our form and to have a better scalability on our forms:

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-login-form',

template: `

<form [formGroup]="loginForm" (ngSubmit)="submit()">

<input type="text" [formControl]="username">

<button [disabled]="loginForm.invalid">Login</button>

</form>

`

})

export class LoginComponent implements OnInit {

loginForm: FormGroup;

username = new FormControl('username');

constructor(private formBuilder: FormBuilder) {}

ngOnInit() {

this.loginForm = this.formBuilder.group({

username: this.username

});

this.loginForm.valueChanges.subscribe(data => console.log(data));

}

submit() {

console.log(this.loginForm.value);

}

}

Angular provides a service called **FormBuilder** that allows you to write less code. To use it, you need to inject it inside a Component via **Angular Dependency Injection**. Then you can use FormGroup to easily build your forms.

Inside the FormGroup there is an object of key-value pairs, where the key is the name of the control and the value is an array of options such as the initial value, the built-in validators and some custom validators.

**Custom validator**

A **custom validator** is a function that receives an input as to where the control is applied. Inside the body, you can handle some verification code like a match to a Regular Expression.

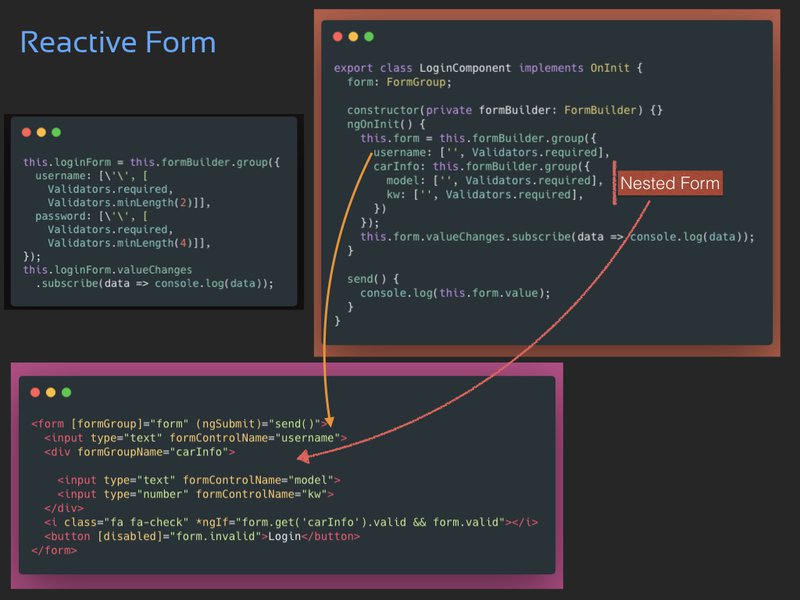
Inside the component logic, there arethree ways to get a**control reference**:

1. form.get(‘company’)
2. form.controls[‘company’]
3. form.controls.company

Here, “form” is the reference of my form. They are all equivalent. This is can be useful to control an HTML element inside the component logic.

**Nested Form**

A **Nested Form Group** is a group of input elements that you need to validate together. It allows you to create an array of objects:

Reactive Forms

Some working examples:

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-login-form',

template: `

<form [formGroup]="loginForm" (ngSubmit)="submit()">

<input type="text" formControlName="username">

<input type="password" formControlName="password">

<button [disabled]="loginForm.invalid">Login</button>

</form>

`

})

export class LoginComponent implements OnInit {

loginForm: FormGroup;

constructor(private formBuilder: FormBuilder) {}

ngOnInit() {

this.loginForm = this.formBuilder.group({

username: ['', [Validators.required, Validators.minLength(2)]],

password: ['', [Validators.required, Validators.minLength(4)]],

});

this.loginForm.valueChanges.subscribe(data => console.log(data));

}

submit() {

console.log(this.loginForm.value);

}

}

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-login-form',

template: `

<form [formGroup]="form" (ngSubmit)="send()">

<input type="text" formControlName="username">

<div formGroupName="carInfo">

<input type="text" formControlName="model">

<input type="number" formControlName="kw">

</div>

<i class="fa fa-check" \*ngIf="form.get('carInfo').valid && form.valid"></i>

<button [disabled]="form.invalid">Login</button>

</form>

`

})

export class LoginComponent implements OnInit {

form: FormGroup;

constructor(private formBuilder: FormBuilder) {}

ngOnInit() {

this.form = this.formBuilder.group({

username: ['', Validators.required],

carInfo: this.formBuilder.group({

model: ['', Validators.required],

kw: ['', Validators.required],

})

});

this.form.valueChanges.subscribe(data => console.log(data));

}

send() {

console.log(this.form.value);

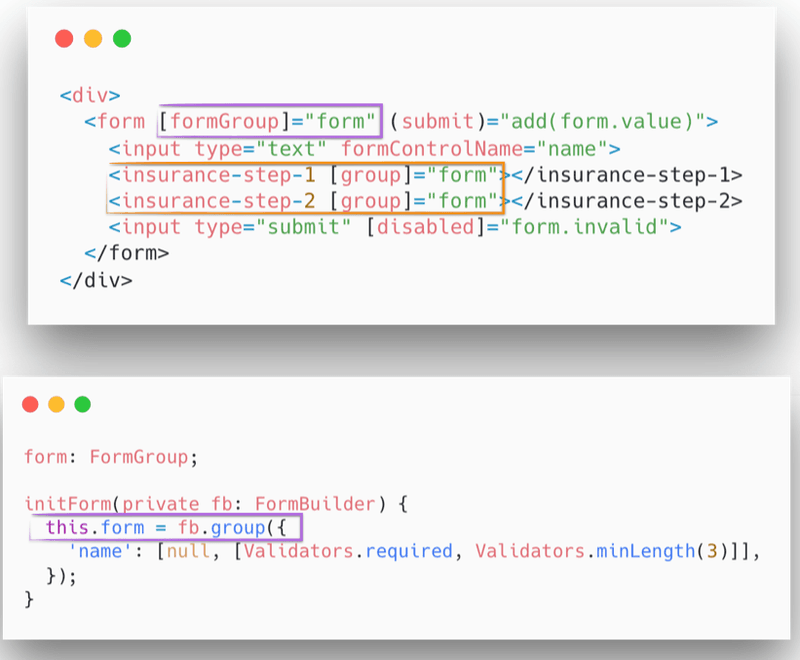
}

}

You can also validate a field based on the value of another input element such as the password matching form fields.

Every form group can have a *second parameter* where you can specify a **custom validator** for the form group itself. This validator is a function where you can pass the reference of the two password fields and do a matching check inside of it.

When your forms grow in complexity, you can split it in **FormGroup** or *split the UI in Components*. So, every FormGroup can be represented with a custom element. See below:

Using Viewchild decorator to access an element

**Dynamic Forms**

With **FormArray** we are able to display an element dynamically. It is a variant of **FormGroup** where data is serialised inside an array instead of an object, like in FormGroup. This is useful when you don’t know how many controls will be hosted inside your dynamic forms.

Finally, you can generate a form at runtime based on a **JSON structure** that may come from an *XHR REST call*. The structure of the JSON can contain properties such as label, type (of the input), validators, etc.

Inside the template, you will use \*ngFor directive and ngSwitch to select which input type you are going to render to the template.

The problem with this solution is that ngSwitch can grow. Another solution comes from using a **Hash Map** for your components to create a form template. For each control you can create a component.

So, in the JavaScript code you’ll have a cycle for reading the JSON configuration and build a form control at runtime, while for the template you can use a \*ngFor; for every iteration a directive called dynamicField will do the trick instantiating the right component for you based on the configuration of the control.

Reactive Forms offer a solution to an otherwise tedious task. All you have to do is write and handle an Angular Reactive form without writing a line of JS.Directives and built-in validators such as “required” or “max-length” or “min-length” are all you need for the form validation.