

Angular training

Angular form and Reactive Form

Introduction to Angular Form

Web applications use forms when it comes to collecting data from the user. Use cases vary from allowing users to log in, filling in payment information, booking a flight, or even performing a search.

Form data can later be persisted on local storage or be sent to a server using a backend API.

A form usually has the following characteristics that enhance the user experience of a web application:

- Can define different kinds of input fields
- Can set up different kinds of validations and display validation errors to the user
- Can support different strategies for handling data if the form is in an error state

Introduction to Angular Form

The Angular framework provides two approaches to handling forms: **template-driven** and **reactive**. The main difference between the two approaches is how they manage data:

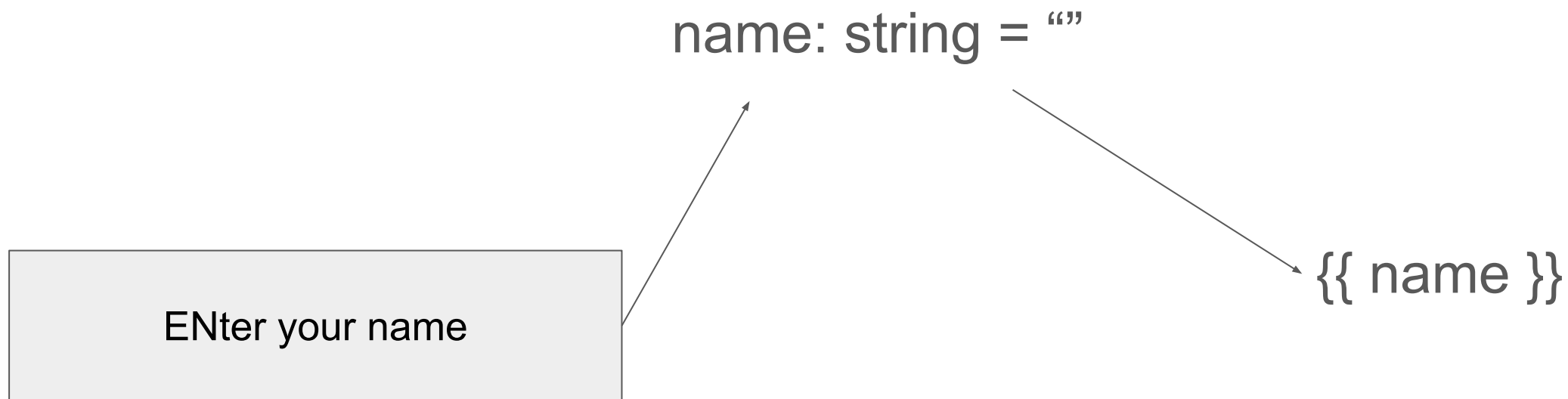
- Template-driven forms are easy to set up and add to an Angular application, but they do not scale well. They operate solely on the component template to create elements and configure validation rules; thus, they are not easy to test. They also depend on the change detection mechanism of the framework.
- Reactive forms are more robust when it comes to scaling and testing and when they are not interacting with the change detection cycle. They operate in the component class to manage input controls and set up validation rules. They also manipulate data using an intermediate form model, maintaining their immutable nature. This technique is for you if you use reactive programming techniques extensively or if your Angular application comprises many forms.

Data binding with template-driven forms

Template driven forms use a two-way binding that can read and write data simultaneously.

Template-driven forms provide the `ngModel` directive, which we can use in our components to get this behavior.

Two way data binding explanation



In two way data binding, when user enter information in the input, the variable name will be updated automatically.

Since there is a change on the variable, the value in {{ }} will be updated in real time as well

Data binding with template-driven forms (2)

product-detail.component.ts

```
import { FormsModule } from '@angular/forms';
```

```
@Component({  
  selector: 'app-product-detail',  
  standalone: true,  
  imports: [CommonModule, FormsModule],  
  templateUrl: './product-detail.component.html',  
  styleUrls: ['./product-detail.component.css']  
})
```

Data binding with template-driven forms (3)

product-detail.component.ts

```
<form [(ngSubmit)]=\"changePrice(product, product.price!)\">
|   <input placeholder=\"New price\" type=\"number\" [(ngModel)]=\"
product.price\" name=\"price\" />
|   <button type=\"submit\">Change</button>
</form>
```

Verify the two way data binding behaviour.

While we type inside the input box, the ngModel directive updates the value of the product price. The new price is directly reflected in the template because we use Angular interpolation syntax to display its value

Data binding with template-driven forms (3)

product-detail.component.ts

```
price: number | undefined;
```


Data binding with template-driven forms (4)

product-detail.component.html

```
<form (ngSubmit)="changePrice(product, price!)">
|   <input placeholder="New price" type="number" [(ngModel)]="
price" name="price" />
|   <button type="submit">Change</button>
</form>
```

Reactive Forms

Reactive forms, as the name implies, provide access to web forms in a reactive manner. They are built with reactivity in mind, where input controls and their values can be manipulated using observable streams.

They also maintain an immutable state of form data, making them easier to test because we can be sure that the state of the form can be modified explicitly and consistently

Reactive forms have a programmatic approach to creating form elements and setting up validation rules. We set everything up in the component class and merely point out our created artifacts in the template.

Reactive Forms (2)

The Angular key classes involved in this approach are the following:

- FormControl: Represents an individual form control, such as an `<input>` element.
- FormGroup: Represents a collection of form controls. The `<form>` element is the top most FormGroup in the hierarchy of a reactive form.
- FormArray: Represents a collection of form controls, just like FormGroup, but can be modified at runtime. For example, we can add or remove FormControl objects dynamically as needed.

Example of Dyanmic Form

Product A	1
Product B	2
Product C	1

Reactive Forms (3)

product-create.component.ts

```
import {ReactiveFormsModule} from '@angular/forms';
```

```
@Component({  
  selector: 'app-product-create',  
  standalone: true,  
  imports: [ReactiveFormsModule],  
  templateUrl: './product-create.component.html',  
  styleUrls: ['./product-create.component.css']  
})
```

Reactive Forms (3)

product-create.component.ts

```
productForm = new FormGroup({  
  name: new FormControl('', { nonNullable: true }),  
  price: new FormControl<number | undefined>(undefined, {  
    nonNullable: true })  
});
```


Reactive Forms (4)

product-create.component.html

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
<div>
  <label for="name">Name</label>
  <input id="name" formControlName="name" />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number"/>
</div>
<div>
  <button type="submit">Create</button>
</div>
</form>
```

Reactive Forms (5)

product-create.component.ts

```
get name() { return this.productForm.controls.name }  
get price() { return this.productForm.controls.price }
```


Reactive Forms (6)

product-create.component.ts

```
createProduct() {  
  this.productsService.addProduct(this.name.value, Number(this.  
    price.value)).subscribe(product =>  
  {  
    this.productForm.reset();  
    this.added.emit(product);  
  });  
}
```

Providing form status feedback


The Angular framework sets the following CSS classes automatically in a form control according to the current status of the control:


- `ng-untouched`: Indicates that we have not interacted with the control yet
- `ng-touched`: Indicates that we have interacted with the control
- `ng-dirty`: Indicates that we have set a value to the control
- `ng-pristine`: Indicates that the control does not have a value yet
- `ng-valid`: Indicates that the value of the control is valid
- `ng-invalid`: Indicates that the value of the control is not valid


Each class name has a similar property in the form model. The property name is the same as the class name without the `ng-` prefix. We could try to leverage both and provide a unique experience with our forms.

Providing form status feedback (2)

styles.css

```
input.ng-touched {  
  border: 3px solid  lightblue;  
}
```

```
input.ng-dirty.ng-valid {  
  border: 2px solid  green;  
}
```

```
input.ng-dirty.ng-invalid {  
  border: 2px solid  red;  
}
```

Providing form status feedback (3)

Add validation rules (required) to verify the style.

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
<div>
  <label for="name">Name</label>
  <input id="name" formControlName="name" required/>
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required/>
</div>
<div>
  <button type="submit">Create</button>
</div>
</form>
```

Validating controls in a reactive way

We will look how we are able to define validation rules when building the FormGroup instance and perform validation at the form control level.

create-product.component.ts

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

```
productForm = new FormGroup({  
  name: new FormControl('', { nullable: true, validators: Validators.required }),  
  price: new FormControl<number | undefined>(undefined, {  
    nullable: true, validators: [Validators.required, Validators.min(1)]})  
});
```

<https://angular.io/guide/form-validation>
<https://angular.io/api/forms/Validators>

Validating controls in a reactive way (2)

Using valid property, you can check if the form is valid or not.

You may also verify the state of the form control, eg: input, and comes out with message, for eg

create-product.component.ts

```
<div>  
  <button type="submit" [disabled]="!productForm.valid">Create</button>  
</div>
```

```
<div>  
  <label for="name">Name</label>  
  <input id="name" formControlName="name" required/>  
  <span *ngIf="name.touched && name.invalid">  
    The name is not valid  
  </span>
```

Validating controls in a reactive way (3)

create-product.component.ts

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required/>
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
  </span>
  <span *ngIf="price.touched && price.hasError('min')">
    The price should be greater than 1
  </span>
</div>
```

Validating controls in a reactive way (4)

The Angular framework provides a set of built-in validators that we can use in our forms.

A validator is a function that returns either a `ValidationErrors` object or `null` when the control does not have any errors.

According to the scenario, a validator can also return a value synchronously or asynchronously

Building a custom validator

With that principle we can write a custom validator and use it in an Angular reactive form.

In this example, we will build a validator to check whether the price of a product is in a predefined amount range.

```
ng g directive price-range
```

Building a custom validator (2)

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

export function priceRangeValidator(): ValidatorFn {
  return (control: AbstractControl<number>): ValidationErrors | null => {
    const inRange = control.value > 1 && control.value < 10000;
    return inRange ? null : { outOfRange: true };
  };
}
```

<https://angular.io/api/forms/ValidatorFn>
https://angular.io/api/forms/ValidatorFn


Building a custom validator (3)

product-create.component.ts

```
import { priceRangeValidator } from '../price-range.directive';
```

```
productForm = new FormGroup({  
  name: new FormControl('', { nullable: true, validators: Validators.required }),  
  price: new FormControl<number | undefined>(undefined, {  
    nullable: true, validators: [Validators.required, priceRangeValidator()]})  
});
```

Building a custom validator (4)

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required/>
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
  </span>
   <span *ngIf="price.touched && price.hasError('outOfRange')">
    The price is out of range
  </span>
</div>
```

Watching state change on a form

A reactive form can listen to changes in the form controls when they happen and react accordingly. A suitable reaction could be to disable/enable a control, provide a visual hint, or do something else according to your needs.

A `FormControl` instance contains two observable properties: `statusChanges` and `valueChanges`.

`statusChanges`: notifies us when the status of the control changes, such as going from invalid to valid.

`valueChanges`: notifies us when the value of the control changes.

Watching state change on a form (2)

Create-product.component.html

```
<span *ngIf="showPriceRangeHint">  
|   Price should be between 1 and 10000  
</span>
```

create-product.component.ts

```
showPriceRangeHint = false;
```


Watching state change on a form (3)

create-product.component.ts

Subscribe to valueChanges on the ngOnInit

```
ngOnInit(): void {  
  this.price.valueChanges.subscribe(price => {  
    if (price) {  
      this.showPriceRangeHint = price > 1 && price < 10000;  
    }  
  });  
}
```

Modifying a dynamic form

Angular's `FormArray` is a powerful tool for managing dynamic forms in your Angular applications. It allows you to handle scenarios where the number of form controls is not fixed and can change dynamically during runtime.

In this example we will create a cart page that will show an item dynamically based on the number of items user added in cart

Modifying a dynamic form (2)

Create a cart service

```
ng generate service cart
```

Modifying a dynamic form (3)

cart.service.ts

```
import { Product } from './product';
```

```
cart: Product[] = [];
```

```
addProduct(product: Product) {  
  this.cart.push(product);  
}
```

Modifying a dynamic form (4)

product-detail.component.ts

```
import { CartService } from '../../../services/cart.service';
```

```
constructor(private productService:ProductService,  
  public authService:AuthService,  
  public route:ActivatedRoute,  
  public cartService:CartService  
) {
```

```
  buyProduct(product:Product){  
    this.cartService.addProduct(product)  
  }
```

Modifying a dynamic form (5)

product-detail.component.html

```
<button (click)="buy(product)" *ngIf="authService.isLoggedIn">Buy Now</button>
```

Modifying a dynamic form (6)

cart.component.ts

```
cartForm = new FormGroup({  
  products: new FormArray<FormControl<number>>([])  
});  
  
cart: Product[] = [];
```

Modifying a dynamic form (7)

cart.component.ts

```
import { FormArray, FormControl, FormGroup, FormsModule } from '@angular/forms';  
import { Product } from '../product';  
import { CartService } from '../cart.service';
```

Modifying a dynamic form (8)

cart.component.ts

```
ngOnInit(): void {  
  this.cart = this.cartService.cart;  
  this.cart.forEach(() => {  
    this.cartForm.controls.products.push(  
      new FormControl(1, { nullable: true })  
    );  
  });  
}
```


Modifying a dynamic form (9)

cart.html.ts

```
<h2>My Cart</h2> <div [formGroup]="cartForm">
  <div
    formArrayName="products"
    *ngFor="let product of cartForm.controls.products.controls; let
i=index">
    <label>{{cart[i].name}}</label>
    <input type="number" [formControlName]="i" />
  </div>
</div>
```


Modifying a dynamic form (10)

cart.component.ts

```
@Component({  
  selector: 'app-cart',  
  standalone: true,  
  imports: [CommonModule,ReactiveFormsModule],  
  templateUrl: './cart.component.html',  
  styleUrls: ['./cart.component.css']  
})
```

Example retrieve value

```
itemSubmit() {  
    const data: any = this.cartForm.controls.products.controls.map((val, index)  
=> ({ 'item': this.cart[index], 'quantity': val.value })))  
    console.log(data);  
  
}
```