Reactive Forms in Angular

What are Reactive Forms?

- Reactive forms objects are objects that provide us with synchronous access to form value data. They're built from observables, so input values and the data value that they bind to are synchronous. Each change in a form state returns a new state.
- This is different from template-driven forms since changes are asynchronous in template-driven forms.
- The synchronous nature of reactive forms makes testing reactive forms easier than template-driven forms.

When Do We Use Reactive Forms?

- Reactive forms should be used in most Angular apps.
- The only benefit that template-driven forms have over reactive forms is that the syntax of template-driven forms is closer to Angular.js forms. Therefore, using template-driven forms would make migrating from Angular.js apps to Angular easier.
- Other than that, there isn't much benefit to using template-driven forms in our Angular apps. So unless we are migrating Angular.js apps to Angular, we should stick with reactive forms.
- The immutability and the predictability because of the synchronous updates of reactive forms makes development much easier. In addition, reactive forms let us define the form's structure explicitly so it's easy to understand and better for scalability

Introduction

Angular provides two ways to work with forms: *template-driven forms* and *reactive forms* (also known as *model-driven forms*). Template-driven forms are the default way to work with forms in Angular. With template-driven forms, template directives are used to build an internal representation of the form. With reactive forms, you build your own representation of a form in the component class.

Here are some of the advantages of reactive forms:

- Using custom validators
- Changing validation dynamically
- Dynamically adding form fields

Key concepts in Reactive Forms:

1. **FormGroup**: Represents a group of form controls. It aggregates the values of each control into a single object.

2. **FormControl**: Represents a single input field — a form control instance manages the value, validation status, and user interactions of an input field.
3. **FormBuilder**: A service that provides convenient methods for creating instances of FormGroup and FormControl.
4. **Validators**: Functions used for synchronous and asynchronous validation of form controls.

To work with reactive forms, you will be using the ReactiveFormsModule instead of the FormsModule.

Step 1

Open app.module.ts in your code editor amd add ReactiveFormsModule:

Step 2 — Adding a Form to the Component Template

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit(myForm)">
 <div>
  <label>
   Name:
   <input formControlName="name" placeholder="Your name">
  </label>
 </div>
 <div>
  <label>
   Email:
   <input formControlName="email" placeholder="Your email">
  </label>
 </div>
 <div>
  <label>
   Message:
   <input formControlName="message" placeholder="Your message">
  </label>
 </div>
 <button type="submit">Send</button>
</form>
```

This code will create a form with three fields: name, email, and message. There will also be a "submit" button with the label "Send". When submitting the form, the method onSubmit(myForm) will be called.

- **formGroup: The form will be treated as a FormGroup in the component class, so the formGroup directive allows to give a name to the form group.**
- **ngSubmit: This is the event that will be triggered upon submission of the form.**
- **formControlName: Each form field should have a formControlName directive with a value that will be the name used in the component class.**

Step 3 — Building the Component Class

Next, in the component class, you will define the `FormGroup` and individual `FormControl`s within the `FormGroup`.
If a value is provided when *newing* a `FormControl`, it will be used as the initial value for the field.
Notice how the `FormGroup` and `FormControl` names are the same that were used in the template. Also notice how you initialize the `FormGroup` in the `ngOnInit` lifecycle hook:

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  myForm: FormGroup;

  ngOnInit() {
    this.myForm = new FormGroup({
      name: new FormControl('Sammy'),
      email: new FormControl(''),
      message: new FormControl('')
```

```
    });
  }

  onSubmit(form: FormGroup) {
    console.log('Valid?', form.valid); // true or false
    console.log('Name', form.value.name);
    console.log('Email', form.value.email);
    console.log('Message', form.value.message);
  }
}
```

Step 4 — Updating the Component Class to Use FormBuilder

The ngOnInit form construction can be rewritten with the FormBuilder helper. This allows you to forgo of all the *newing* of form group and form controls.

Revisit app.component.ts in your code editor and remove FormControl and replace FormGroup with FormBuilder:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  myForm: FormGroup;

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.myForm = this.fb.group({
      name: 'Sammy',
```

```
    email: '',
    message: ''
  });
 }

 onSubmit(form: FormGroup) {
  console.log('Valid?', form.valid); // true or false
  console.log('Name', form.value.name);
  console.log('Email', form.value.email);
  console.log('Message', form.value.message);
 }
}
```

Step 5 — Updating the Component Class to Use Validators

Add the Validators class to your imports and declare your form controls with arrays instead of simple string values.

The first value in the array is the initial form value and the second value is for the validator(s) to use. Notice how multiple validators can be used on the same form control by wrapping them into an array.

RevisitRevisit app.component.ts in your code editor and add Validators:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
 myForm: FormGroup;

 constructor(private fb: FormBuilder) {}
```

```
  ngOnInit() {
    this.myForm = this.fb.group({
      name: ['Sammy', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      message: ['', [Validators.required, Validators.minLength(15)]],
    });
  }

  onSubmit(form: FormGroup) {
    console.log('Valid?', form.valid); // true or false
    console.log('Name', form.value.name);
    console.log('Email', form.value.email);
    console.log('Message', form.value.message);
  }
}
```

Step 6 — Accessing Form Value and Validity in the Template

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit(myForm)">
  <div>
    <label>
      Name:
      <input formControlName="name" placeholder="Your name">
    </label>
    <div *ngIf="myForm.get('name').invalid && (myForm.get('name').dirty ||
myForm.get('name').touched)">
      Please provide a name.
    </div>
  </div>
  <div>
    <label>
      Email:
```

```
    <input formControlName="email" placeholder="Your email">
  </label>
  <div *ngIf="myForm.get('email').invalid && (myForm.get('email').dirty ||
myForm.get('email').touched)">
    Please provide a valid email address.
  </div>
 </div>
 <div>
  <label>
    Message:
    <input formControlName="message" placeholder="Your message">
  </label>
  <div *ngIf="myForm.get('message').invalid && (myForm.get('message').dirty ||
myForm.get('message').touched)">
    Messages must be at least 15 characters long.
  </div>
 </div>
 <button type="submit" [disabled]="myForm.invalid">Send</button>
</form>
```

Template Driven Forms Vs. Reactive Forms:

Template-driven forms and reactive forms are two ways to create forms in Angular, each with its own strengths and weaknesses.

Template Driven Forms:

- Easier to use and understand, especially for simple forms.
- Form logic is mostly in the template, making it less maintainable for complex forms.
- Two-way data binding is used [(ngModel)] for form controls.
- Less control over form validation and error handling compared to reactive forms.

Reactive Forms:

- More flexible and maintainable, especially for complex forms.
- Form logic is in the component class, making it easier to test and manage.
- Reactive forms provide better support for dynamic forms and custom validation.

- More boilerplate code compared to template-driven forms.

Reactive Form Control and Sync View:

In reactive forms, FormControl is used to manage the value and validation state of an individual form control. It provides methods to set and get the value, update the validation status, and listen to value changes.

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-form-control',
  template: `
    <input type="text" [formControl]="nameControl">
    <p>Value: {{ nameControl.value }}</p>
    <p>Validation status: {{ nameControl.status }}</p>
  `
})
export class ReactiveFormControlComponent {
  nameControl = new FormControl('');

  constructor() {
    this.nameControl.valueChanges.subscribe(value => {
      console.log('Value changed:', value);
    });
  }
}
```

In this example, we create a FormControl instance for an input field. The form control is bound to the input field using the formControl directive. We also subscribe to the valueChanges observable to listen to changes in the form control's value and log the new value to the console.

Dynamic Validations in Reactive Forms:

In reactive forms, you can dynamically set and update validations for form controls based on user interactions or other factors. This can be done using the setValidators method of a FormControl instance.

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

@Component({
  selector: 'app-dynamic-validations',
```

```
  template: `
    <form [formGroup]="myForm">
      <input type="text" formControlName="name">
      <div *ngIf="myForm.get('name').errors?.required">Name is
required</div>
      <button (click)="toggleValidation()">Toggle Validation</button>
    </form>
  `
})
export class DynamicValidationsComponent {
  myForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.myForm = this.fb.group({
      name: ['', Validators.required]
    });
  }

  toggleValidation() {
    const control = this.myForm.get('name');
    if (control.validator === Validators.required) {
      control.clearValidators();
    } else {
      control.setValidators(Validators.required);
    }
    control.updateValueAndValidity();
  }
}
```

In this example, we create a form with a single form control ‘name’ that is initially required. The toggleValidation method is used to dynamically add or remove the required validation based on the current validation state of the control.

Getting Reactive Form Values:

You can get the current value of a FormGroup or FormControl by accessing the value property. This property returns an object containing the current value of each form control in the group.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-get-values',
  template: `
    <form [formGroup]="myForm" (ngSubmit)="onSubmit()">
```

```
      <input type="text" formControlName="name">
      <input type="email" formControlName="email">
      <button type="submit">Submit</button>
    </form>
  `
})
export class GetValuesComponent {
  myForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.myForm = this.fb.group({
      name: [''],
      email: ['']
    });
  }

  onSubmit() {
    console.log('Form values:', this.myForm.value);
  }
}
```

In this example, we create a form with two form controls ‘name’ and ‘email’. The onSubmit method logs the current values of the form controls when the form is submitted.

Dirty, Touched, Pristine, Untouched:

These are properties of form controls that provide information about their state.

- Dirty: A control is considered dirty if the user has changed the value. It becomes true once the user interacts with the control.
- Touched: A control is touched if the user has focused on the control and then moved away. It becomes true once the control loses focus.
- Pristine: A control is considered pristine if the user has not changed the value. It is the opposite of dirty. A control starts as pristine and becomes dirty once the user interacts with it.
- Untouched: A control is untouched if the user has not focused on the control. It is the opposite of touched.

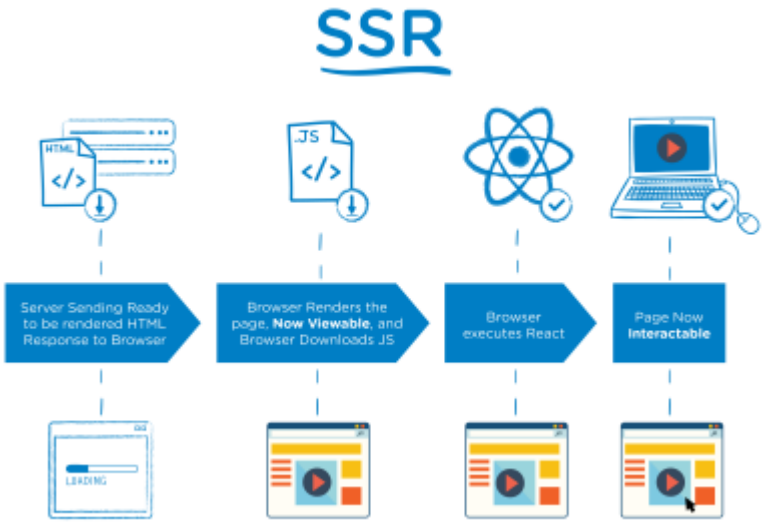
- **valid**: This property returns **true** if **the element's contents are valid and false otherwise.**
- **invalid**: This property returns **true** if **the element's contents are invalid and false otherwise.**
- **pristine**: This property returns **true** if **the element's contents have not been changed.**
- **dirty**: This property returns **true** if **the element's contents have been changed**.
- **untouched**: This property returns **true if the user has not visited the element.**
- **touched**: This property returns **true** if **the user has visited the element.**

- **Single Page Application**: The entire application is loaded on the client, and a single page is loaded. The appearance of navigating throughout the application is controlled dynamically through JavaScript on the client

- **Server Side Rendering**:

    - Rather than delivering the app entirely to the client, the app will run on an external server. Requests for pages can be made to that server, the server will render that page, and then deliver it to the client

    - Server-side rendering (SSR) is the process of rendering web pages on a server and passing them to the browser (client-side), instead of rendering them in the browser. SSR sends a fully rendered page to the client; the client's JavaScript bundle takes over and enables the SPA framework to operate.



    - This means that if your application is server-side rendered, the content is fetched from the server and passed to the browser to be displayed to the user. Client-side rendering is different: The user would have to navigate to the page before the browser fetches data from the server, meaning that the user would have to wait for some seconds to pass before the browser is served with the content for that page. Applications that have SSR enabled are called server-side-rendered applications.

    - This approach is good if you're building a complex application that requires user interaction, that relies on a database, or whose content changes often. If the content changes often, then users would need to see the updates right away. The approach is also good for applications that tailor content according to who is viewing it and that store user data such as email addresses and user preferences, while also attending to SEO. An example would be a large e-commerce or social media platform. Let's look at some of the advantages of SSR for your applications.

- **Pros**

    1. The content is up to date because it is fetched on the go.
    2. The website loads quickly because the browser fetches content from the server before rendering it for the user.
    3. Because the JavaScript is rendered server-side, the user's device has little bearing on the loading time of the page, making for better performance.

- **Cons #**

    1. More API calls to the server are made, because they're made per request.
    2. The website cannot be deployed to a static content delivery network (CDN).

- **Static Site Generation:**
- The application is built once to create many different static pages, and then these pages are loaded directly from a server (rather than being generated by the server on the fly)

  A static-site generator (SSG) is a software application that creates HTML pages from templates or components and a given content source. Give it some text files and content, and the generator will give you back a complete website; this completed website is referred to as a static-generated site. This means that the website's pages are generated at build time, and their contents do not change unless you add new content or components and then **rebuild** — you have to rebuild the website if you want it to be updated with the new content.

  This approach is good for building applications whose content does not change often. So, you wouldn't necessarily use it for a website that has to be modified according to the user or one that has a lot of user-generated content. However, a blog or personal website would be an ideal use. Let's look at some advantages of static-generated sites.

  Pros #

- **Speed**
  Because all of your website's pages and content will be generated at build time, you do not have to worry about API calls to a server for content, which will make your website very fast.
- **Deployment**
  Once your static site has been generated, you will be left with static files. Hence, it can be easily deployed to a platform such as Netlify.

- **Security**
  A static-generated site solely comprises static files, with no database for an attacker to exploit by injecting malicious code. So, vulnerability to a cyber attack is minimal.
- **Verson control**
  You can use version control software (such as Git) to manage and track changes to your content. This comes in handy when you want to roll back changes made to the content.

Cons #

- If the content changes too quickly, it can be hard to keep up.
- To update content, you have to rebuild the website.
- The build time increases according to the size of the application.