**📒 Notes on Reactive Forms in Angular**

**✅ 1. What is a Reactive Form?**

A **Reactive Form** is a structured, model-driven approach to handling form inputs, validations, and submission in Angular. It uses FormControl, FormGroup, and FormBuilder classes.

* Form state and validation logic are defined **in the component class**.
* Reactive forms are synchronous and explicit, making them ideal for complex forms.

**✅ 2. Key Building Blocks**

| **Concept** | **Description** |
| --- | --- |
| **FormControl** | Tracks the value and validation status of a single form input. |
| **FormGroup** | Groups multiple FormControls together. |
| **FormBuilder** | Service that simplifies creating FormGroups and FormControls. |
| **Validators** | Functions that validate a control (e.g. required, minLength, email). |
| **Form-level Validators** | Validators applied to the entire FormGroup, useful for cross-field validation (like password match). |

**✅ 3. Creating a Reactive Form (in ngOnInit)**

this.registerForm = this.formBuilder.group({

firstName: ['', Validators.required],

email: ['', [Validators.required, Validators.email]],

...

}, {

validators: [Validation.match('password', 'confirmPassword')]

});

* Use formBuilder.group() to initialize form controls.
* Nested validations are passed as arrays.
* Custom validation logic (like password confirmation) is added to the group as a **form-level validator**.

**✅ 4. Accessing Controls**

get firstName() {

return this.registerForm.get('firstName');

}

* Using getters simplifies code in the HTML template.
* You can access status like .valid, .touched, .errors, etc.

**✅ 5. Validating User Input**

* Use Angular’s built-in validators:

Validators.required,

Validators.email,

Validators.minLength(n),

Validators.requiredTrue

* Displaying errors:

<div \*ngIf="firstName?.hasError('required')">First Name is required</div>

* Use touched, dirty, or submitted flags to control when error messages appear.

**✅ 6. Submitting a Reactive Form**

onSubmit() {

this.submitted = true;

if (this.registerForm.invalid) {

return;

}

// Map form values to model

let contact = new Contact();

contact.firstname = this.firstName?.value;

...

}

* Always validate the form before proceeding.
* registerForm.value contains the form data as a plain object.
* Use .reset() if needed after submission.

**✅ 7. Custom Validators (e.g. Password Match)**

export default class Validation {

static match(controlName: string, checkControlName: string): ValidatorFn {

return (controls: AbstractControl) => {

const control = controls.get(controlName);

const checkControl = controls.get(checkControlName);

if (control?.value !== checkControl?.value) {

checkControl?.setErrors({ matching: true });

return { matching: true };

} else {

return null;

}

};

}

}

* Used for cross-field logic.
* Should check values and set custom error if mismatched ({ matching: true }).

**✅ 8. Commonly Used Validator Error Keys**

| **Validator** | **Error Key** |
| --- | --- |
| Validators.required | 'required' |
| Validators.email | 'email' |
| Validators.minLength(n) | 'minlength' |
| Validators.requiredTrue | 'requiredTrue' |
| Custom validator (e.g. password match) | 'matching' (or any name you return) |

**✅ 9. UI Feedback with Bootstrap / Custom Classes**

Use conditional classes for validation:

[ngClass]="{ 'is-invalid': control?.invalid && (control?.touched || control?.dirty) }"

And show error messages using:

<div \*ngIf="control?.invalid && (control?.touched || control?.dirty)" class="invalid-feedback">

<div \*ngIf="control?.hasError('required')">Field is required</div>

</div>

**✅ 10. Advantages of Reactive Forms**

* Better suited for complex and dynamic forms.
* Easier to test due to component-driven logic.
* Great control over validation and form state.
* Clean separation of concerns between logic and template.