Template Forms and Reactive Forms in Angular:

Template Forms vs Reactive Forms

Aspect	Template-Driven Forms	Reactive Forms
Form Creation	Defined directly in HTML using directives like ngModel	Defined in the Component class using FormControl, FormGroup, etc.
Form Control	Angular automatically tracks controls	Developer explicitly manages controls
Complex Forms	Less suitable for complex forms	Better for dynamic and complex forms
Validation	Uses directives in the template	Uses validator functions inside the component

---

Key Classes for Reactive Forms

1. FormBuilder

- A **helper service** to build forms easily.
- Reduces boilerplate code when creating forms.

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

this.form = this.fb.group({
  name: ['', Validators.required],
  email: ['', [Validators.required, Validators.email]],
  password: ['', [Validators.required, Validators.minLength(6)]]
})
```

2. FormGroup

- A **container** for multiple FormControls.
- It manages state & validation for a group of inputs.

```
this.profileForm = new FormGroup({
  firstName: new FormControl(''),
  lastName: new FormControl(''),
});
```

3. FormControl

- Represents a **single input** (e.g., input, select).
- Manages value, validation, and status.

```
this.email = new FormControl('', [Validators.required, Validators.email]);
```

Validating Forms

Example with Reactive Form

```
this.userForm = this.fb.group({
  email: ['', [Validators.required, Validators.email]],
  password: ['', [Validators.required, Validators.minLength(8)]]
});
```

Display Error in Template

```
<input type="email" formControlName="email">
<div *ngIf="userForm.get('email')?.invalid && userForm.get('email')?.touched">
  Invalid email address.
</div>
```

Creating Custom Validators

Example: Password Validator (at least 1 uppercase and 1 number)

```
function customPasswordValidator(control: FormControl) {
  const value = control.value;
  if (!/[A-Z]/.test(value) || !/[0-9]/.test(value)) {
    return { passwordStrength: true };
  }
  return null; // valid
}
this.signupForm = this.fb.group({
  password: ['', [Validators.required, customPasswordValidator]]
});
```

Asynchronous Validations

Example: Checking if email already exists (simulate API call)

```
emailExistsValidator(control: FormControl): Observable<ValidationErrors |
null> {
  return of(control.value).pipe(
    delay(1000), // simulate server latency
    map(value => value === 'already@used.com' ? { emailTaken: true } : null)
  );
}
this.form = this.fb.group({
  email: ['', [Validators.required, Validators.email],
[this.emailExistsValidator]]
});
<div *ngIf="form.get('email')?.errors?.['emailTaken']">
  Email is already taken.
</div>
```

Submitting Form Data to Server

```
submitForm() {
  if (this.form.valid) {
    this.http.post('https://api.example.com/submit', this.form.value)
```

```
      .subscribe(response => {
        console.log('Form submitted successfully', response);
      });
  }
}
```

```
<form [formGroup]="form" (ngSubmit)="submitForm()">
  <input formControlName="email">
  <button type="submit">Submit</button>
</form>
```

FormControl States

State	Meaning	When does it change?
touched	The field has been visited (focused and then blurred).	When the user **focuses into** the field and **leaves** it.
untouched	The field has never been visited (never gained focus).	Initially, all fields start as `untouched`. Once touched, they never go back to `untouched`.
dirty	The value of the field has changed (user typed something).	As soon as the user **types or modifies** the value.
pristine	The field's value is still the **initial value**.	Initially all fields start as `pristine`. Once changed, they become `dirty`.

Example Timeline (for a text input)

1 Initial state:

- `touched = false` **(user hasn't clicked on the input yet)**
- `untouched = true`
- `dirty = false` **(user hasn't typed anything yet)**
- `pristine = true`

---

2 User clicks into the input field (focus), then clicks outside (blur):

- `touched = true`
- `untouched = false`

---

3 User types something into the field (modifies value):

- `dirty = true`
- `pristine = false`

Why these states matter in forms?

State	Usage Example
touched	Show validation error **after user visits the field** (don’t scare user with errors immediately).
dirty	Track if user changed something before submission (useful for showing "Unsaved changes" warning).
pristine	Detect if form is in its **original state** (for reset button logic).

In Example

For **Username field**, you used:

```
*ngIf="(profileData.controls['uname'].touched ||
profileData.controls['uname'].dirty)
&& profileData.controls['uname'].hasError('required')"
```

This means:

Should we show the error?
Yes, if:

- The user **visited the field** (touched) OR
- The user **changed the value** (dirty)

Quick Summary Table

Property	True When
touched	Field was focused and then blurred.
untouched	Field was never focused.
dirty	Field value was changed.
pristine	Field value was never changed.