**Understanding FormBuilder, FormGroup, and FormControl in Angular**

Angular's Reactive Forms module provides a powerful and flexible way to manage form data and validation. The core components of this module are FormBuilder, FormGroup, and FormControl.

**FormBuilder**

* **Purpose:** Simplifies the creation of FormGroups and FormControls.
* **Functionality:** Offers methods like group(), control(), and array() to create form elements efficiently.
* **Example:**

TypeScript

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

constructor(private fb: FormBuilder) {

this.myForm = this.fb.group({

name: ['', Validators.required],

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

[1. github.com](https://github.com/srmono/neo_ang" \t "_blank)

[github.com](https://github.com/srmono/neo_ang" \t "_blank)

});

}

Use code [with caution.](/faq#coding)

**FormGroup**

* **Purpose:** Represents a collection of FormControls.
* **Functionality:** Manages the overall state of the form, including validation, value changes, and user interactions.
* **Example:**

TypeScript

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

this.myForm = new FormGroup({

name: new FormControl('', Validators.required),

email: new FormControl('', [Validators.required, Validators.email])

});

Use code [with caution.](/faq#coding)

**FormControl**

* **Purpose:** Represents an individual form input element.
* **Functionality:** Tracks the value, validation status, and user interactions of a single form field.
* **Example:**

TypeScript

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

this.nameControl = new FormControl('', Validators.required);

Use code [with caution.](/faq#coding)

**Key Differences and Relationships:**

* **FormBuilder** is a utility class that helps create FormGroup and FormControl instances efficiently.
* **FormGroup** is a container for multiple FormControl instances.
* **FormControl** represents a single form input element.

**When to Use Which:**

* Use FormBuilder when you need to create complex forms with multiple controls and nested groups.
* Use FormGroup when you have a group of related form controls that need to be validated and managed together.
* Use FormControl when you have a standalone form control that doesn't belong to a group.

**Additional Notes:**

* You can nest FormGroups within FormGroups to create complex form structures.
* FormControl instances can be added or removed dynamically from a FormGroup.
* Reactive Forms offer advanced features like validation, dirty checks, and value changes, making them suitable for complex form scenarios.

By understanding these core components, you can effectively build and manage forms in your Angular applications.

**Would you like to see a practical example of a reactive form with these components?**

[1. github.com](https://github.com/LinasEiningis/ng-register-screen" \t "_blank)

[github.com](https://github.com/LinasEiningis/ng-register-screen" \t "_blank)

Sources and related content

**Template-Driven Forms in Angular**

Template-driven forms offer a declarative approach to building forms in Angular. The form logic is directly embedded within the HTML template, making it simpler for smaller forms.

**Key Directives:**

* **ngModel:** Binds a property in the component to the form control element.
* **ngForm:** Creates a form instance and provides access to form-related properties and methods.
* **required:** Built-in validator for required fields.
* **minlength, maxlength:** Built-in validators for input length.
* **pattern:** Built-in validator for matching a regular expression.

**Example:**

HTML

<form #userForm="ngForm">

<div>

<label for="name">Name:</label>

<input type="text" id="name" name="name" required [(ngModel)]="user.name">

<div \*ngIf="userForm.controls['name'].invalid && (userForm.controls['name'].dirty || userForm.controls['name'].touched)">

<div \*ngIf="userForm.controls['name'].errors.required">Name

[1. github.com](https://github.com/codingAyush/order-management-workspace" \t "_blank)

[github.com](https://github.com/codingAyush/order-management-workspace" \t "_blank)

is required</div>

</div>

</div>

<div>

<label for="email">Email:</label>

<input type="email" id="email" name="email"

[1. github.com](https://github.com/nawres3b4/AngularProjet" \t "_blank)

[github.com](https://github.com/nawres3b4/AngularProjet" \t "_blank)

required [(ngModel)]="user.email" pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}">

<div \*ngIf="userForm.controls['email'].invalid && (userForm.controls['email'].dirty || userForm.controls['email'].touched)">

<div \*ngIf="userForm.controls['email'].errors.required">Email

[1. github.com](https://github.com/juliangruendner/ketos_gui" \t "_blank)

[MIT](https://github.com/juliangruendner/ketos_gui" \t "_blank)

[github.com](https://github.com/juliangruendner/ketos_gui" \t "_blank)

is required</div>

<div \*ngIf="userForm.controls['email'].errors.pattern">Invalid email format</div>

</div>

</div>

<button type="submit"

[1. github.com](https://github.com/vega2k/userApp-boot-angular" \t "_blank)

[github.com](https://github.com/vega2k/userApp-boot-angular" \t "_blank)

[disabled]="userForm.invalid">Submit</button>

</form>

Use code [with caution.](/faq#coding)

TypeScript

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

@Component({

selector: 'app-template-driven-form',

templateUrl: './template-driven-form.component.html',

styleUrls: ['./template-driven-form.component.css']

[1. github.com](https://github.com/Heyff12/angular-session-demo" \t "_blank)

[github.com](https://github.com/Heyff12/angular-session-demo" \t "_blank)

})

export class TemplateDrivenFormComponent

[1. mukeshkumar.net](http://mukeshkumar.net/articles/angular/template-driven-forms-and-validation-in-angular-with-typescript" \t "_blank)

[mukeshkumar.net](http://mukeshkumar.net/articles/angular/template-driven-forms-and-validation-in-angular-with-typescript" \t "_blank)

{

user = {

name: '',

email: ''

};

onSubmit() {

console.log(this.user);

}

}

Use code [with caution.](/faq#coding)

**Explanation:**

* **ngForm:** Creates a form instance with the reference userForm.
* **ngModel:** Binds form controls to properties in the component's user object.
* **required, pattern:** Apply built-in validators to the form controls.
* **[disabled]="userForm.invalid":** Disables the submit button if the form is invalid.
* \***ngIf:** Displays error messages based on form control validity and user interaction.

**Advantages:**

* Simple to implement for small forms.
* Declarative syntax is often easier to read and understand.

**Disadvantages:**

* Becomes complex for larger forms with multiple interactions.
* Less testable compared to reactive forms.
* Limited control over form state and behavior.

**When to use Template-Driven Forms:**

* Small forms with basic validation requirements.
* Rapid prototyping or simple data collection.

**Remember:** While template-driven forms are suitable for simple scenarios, reactive forms offer more flexibility and control for complex forms.

**Would you like to see a comparison between template-driven and reactive forms?**

**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.