Angular Forms and Services

Importance of forms and services in Angular applications

Forms are UI elements that capture user input, allowing users to interact and provide data to the application.

forms act as the user interface for data entry, while services act as the behind-the-scenes engine that powers data processing and logic execution…

Services are singleton objects that implement business logic, manage data, and provide methods to other components in the application

Angular Forms

Angular forms are a key feature that allows users to interact with web applications by inputting data.

They handle user input, provide validation to ensure data accuracy, and offer error messages for incorrect entries.

Forms maintain their state, making it easy to track user actions, and can be created using either template-driven or reactive approaches for flexibility and ease of use.

Template-driven vs. Reactive forms

Template-driven Forms	Reactive Forms
Forms are structured and managed primarily within the template using directives like ngForm and ngModel.	Forms are built and managed in the component class using reactive programming principles and form model classes (FormGroup, FormControl, etc.).
Form model is implicit in the template	Form model is explicitly defined in the component class
Validation rules are defined in the template.	Validation rules are defined in the component class.
Relies on two-way data binding with ngModel.	Uses observable streams for form value and state changes.
Access form model through ngForm directive in the template	Offers a synchronous model, allowing direct access to form controls

Advantages and Use Cases of Template-driven Forms

Advantages

- **Simplicity and Quick Setup:** Easy to learn and use
- **Two-Way Data Binding:**Automatic synchronization between model and view using ngModel.
- **Declarative Validation:** Define validation rules directly in the template using directives
- **Natural for Simple Forms:** Well-suited for forms with basic validation and straightforward structure.

Use Cases

- **Login forms:** Simple forms with username and password fields.
- **Contact forms:** Basic forms for collecting user information (name, email, phone, message).
- **Subscription forms:** Forms for capturing email addresses or basic user details.
- **Search forms:** Forms with input fields for filtering data.
- **Profile editing forms:** Forms for updating user information with limited fields.

Advantages and Use Cases of Reactive Forms

Advantages

- Increased Control and Flexibility.
- Synchronous Access and Predictability.
- Improved Unit Testing.
- Scalability for Large Forms
- Reusable Validation Logic.
- Integration with Reactive Patterns.

Use Cases

- Complex forms with multiple sections and nested groups
- Dynamic forms that change structure based on user input
- Forms requiring extensive validation with custom rules
- Real-time error feedback and dynamic UI updates
- Integration with asynchronous operations
- Large-scale applications with numerous interactive forms

How to create template-driven forms

1.Import FormsModule: In your app module or the module where you want to use the form, import FormsModule from @angular/forms

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

@NgModule({
  imports: [
    // ... other imports
    FormsModule
  ],
  // ...
})
export class AppModule { }
```

How to create template-driven forms

2. Create the Form Template:

Use the <form> tag

with [(ngModel)] on input elements

to bind to properties in your

component.

3. Add validation (optional):

Optionally, you can implement

validation rules for your form

controls using built-in validators or

custom validation directives.

HTML

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" [(ngModel)]="user.name" required>
  <button type="submit" [disabled]="!myForm.valid">Submit</button>
</form>
```

How to create template-driven forms

4 .Handle Form Submission:

In your component class, create a method to handle form submission.

```typescript
export class MyComponent {
  user = { name: '' };

  onSubmit() {
    console.log('Form data:', this.user);
    // Send data to server, etc.
  }
}
```

5. Access Form Model:

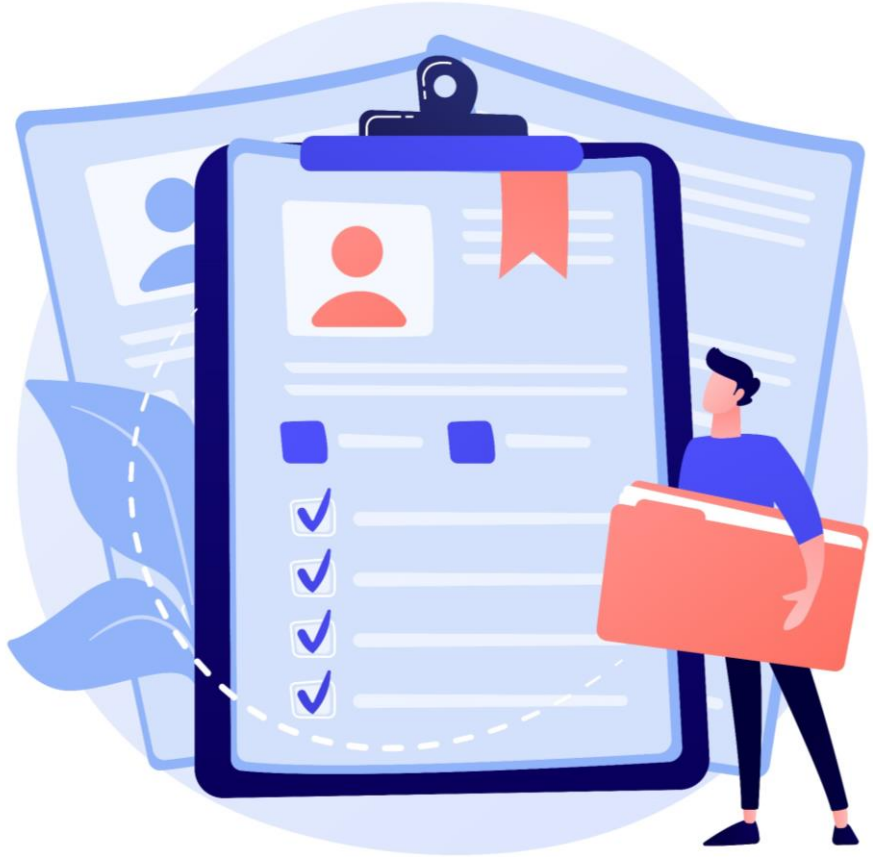
Use the template reference variable to access form properties in the template

```html
{{ myForm.value | json }} ```
```

Introduction To Reactive forms

- A model-driven approach where forms are represented as code in component classes, providing more control and flexibility over validation.
- Defined in the component class using classes like FormGroup and FormControl. These represent the form structure and individual controls.
- Defined in the component class with built-in validators or custom functions.
- Each FormControl offers fine-grained control over validation, error messages, and state management.

FormControl, FormGroup, and FormArray classes



FormControl

- Represents a single input field or form control.
- Tracks value, validation status, and state changes
- Created using new FormControl(initialValue, validators).
- Example: name = new FormControl(' ', Validators.required);

FormGroup

- Groups multiple FormControls together into a logical structure.
- Represents a collection of related controls, often representing a complete form or a section of a form.
- Manages validation and state for the entire group.
- Created using new FormGroup({ controlName1: new FormControl(), controlName2: new FormControl() });
- Example: addressForm = new FormGroup({ street: new FormControl(), city: new FormControl(), state: new FormControl() });

FormArray

- Manages an array of FormControls or FormGroups.
- Useful for handling dynamic forms with varying numbers of controls.
- Allows adding, removing, and reordering controls at runtime.
- Created using new FormArray([initialControls]);
- Example: hobbies = new FormArray([new FormControl('reading'), new FormControl('coding')]);

Validators in Reactive forms

- In Angular forms, validators are used to validate the user input. Validators can be applied to form controls to ensure that the entered data meets certain criteria. Angular provides built-in validators as well as the ability to create custom validators.

- Validators are functions that process form controls or groups and return an error map (if validation fails) or null (if it passes).

Validators in Reactive forms

1.Built-in Validators:

- **required:** Ensures a control has a non-empty value.
- **minLength:** Enforces a minimum length.
- **maxLength:** Enforces a maximum length.
- **pattern(pattern):** Matches against a regular expression.
- **email:** Validates email format.
- **min:** Ensures a value is greater than or equal to a minimum.
- **max:** Ensures a value is less than or equal to a maximum.

2.Custom Validators:

- Define your own validation logic for specific needs.
- Create a function that accepts an AbstractControl and returns an error object or null.

How to create reactive forms

Import ReactiveFormsModule:

- Make sure to import the ReactiveFormsModule in your module to enable reactive forms and access form-related directives.

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

@NgModule({
  imports: [ReactiveFormsModule],
  // other module configurations
})
export class YourModule { }
```

How to create reactive forms

Create a form group:

- Use the FormGroup class to create a form group, which represents the entire form or a logical section of it. You can also use the FormBuilder service to simplify the process.

typescript

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

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

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.myForm = this.fb.group({
      // form controls go here
    });
  }
}
```

How to create reactive forms

Add form controls:

- Inside the form group, add form controls using the **FormControl** class. Specify initial values and any validators you want to apply.

```typescript
this.myForm = this.fb.group({
  username: ['JohnDoe'],
  email: ['john.doe@example.com'],
  password: ['password'],
});
```

- If you want to add validators, you can pass an array of validators as the second argument

```typescript
(property) username: any[]
{
  username: ['JohnDoe', Validators.required],
  email: ['john.doe@example.com', [Validators.required, Validators.email]],
  password: ['password', Validators.minLength(6)],
});
```

How to create reactive forms

Bind the form in the template:

- In your template, bind the form using the formGroup directive. Also, bind each form control using the formControlName directive.

```html
<form [formGroup]="myForm">
  <label for="username">Username:</label>
  <input type="text" id="username" formControlName="username">

  <label for="email">Email:</label>
  <input type="email" id="email" formControlName="email">

  <label for="password">Password:</label>
  <input type="password" id="password" formControlName="password">

  <button type="submit" (click)="submitForm()">Submit</button>
</form>
```

How to create reactive forms

Handle form submission:

- Implement the logic to handle form submission in your component.

```typescript
submitForm() {
  if (this.myForm.valid) {
    // Perform actions with form data
    console.log(this.myForm.value);
  }
}
```

Creating dynamic forms using FormArray

- **Import FormArray:** In your component, import the FormArray class from the '@angular/forms' module.
- **Initialize FormArray:** Create a new instance of FormArray in your component and set it as a property.
- **Add Form Controls Dynamically:** Use the FormArray's 'push' method to add new FormControl instances to the FormArray when needed.

Creating dynamic forms using FormArray

- **Remove Form Controls Dynamically:** Use the FormArray's 'removeAt' method to remove form controls from the FormArray based on the user's actions or other conditions.

- **Link FormArray to the Template:** In the template, use *ngFor directive to loop through the FormArray and display the dynamically added form controls.

Services in Angular

- Services in Angular are singleton objects that provide shared functionality across your application.
- Services are used to share data, business logic, and other common functionalities across components.
- Services are classes that contain methods and data accessible across multiple components in your Angular app.
- Use the ng generate service command to create a new service class.

What is Dependency Injection

- Dependency Injection (DI) is a design pattern widely used in software development and is a core concept in Angular.
- Dependency Injection is a way to provide and manage the dependencies (services or objects) that a class (component, service, or other Angular construct) needs..
- define the dependencies that a class needs in its constructor.
- The injector is responsible for creating and managing instances of objects (services, components, etc.) and their dependencies.
- To enable dependency injection, a class needs to be marked with the **@Injectable** decorator.

Dependency Injection

Dependency: A class or service that another class needs to function.

Injector: A mechanism that creates and provides dependencies to classes that need them.

Provider: A recipe that tells the injector how to create a dependency.

How does Services and Dependencies works in angular

- Services are used to encapsulate and share functionality across components, and dependency injection is the mechanism Angular uses to provide and manage these services.

Step 1: Create a Service :

1.Generate a Service: Use the Angular CLI to generate a service

2.Define the Service: Open the generated service file and define your service class. Add properties and methods that encapsulate the functionality you want to share.

```bash
ng generate service your-service-name
```

```typescript
// your-service-name.service.ts

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

@Injectable({
  providedIn: 'root',
})
export class YourServiceNameService {
  private data: string = 'Hello from the service!';

  getData(): string {
    return this.data;
  }

  setData(newData: string): void {
    this.data = newData;
  }
}
```

How does Services and Dependencies works in angular

Step 2: Use the Service in a Component

1.Inject the Service into a Component:

In the component where you want to use the service, inject it in the constructor.

typescript

```typescript
// your-component.component.ts

import { Component } from '@angular/core';
import { YourServiceNameService } from './your-service-name.service';

@Component({
  selector: 'app-your-component',
  template: `
    <div>
      {{ message }}
    </div>
  `,
})
export class YourComponent {
  message: string;

  constructor(private yourService: YourServiceNameService) {
    this.message = yourService.getData();
  }
}
```

How does Services and Dependencies works in angular

Step 3: Register the Service

1. Register the Service in a Module:

- Make sure to register your service in an Angular module to make it available for dependency injection.
- The providedIn: 'root' in the @Injectable decorator ensures that Angular creates a single instance of the service for the entire application.

typescript

```typescript
// app.module.ts

import { NgModule } from '@angular/core';
import { YourServiceNameService } from './your-service-name.service';

@NgModule({
  providers: [YourServiceNameService],
  // other module configurations
})
export class AppModule { }
```