JAVA FULL STACK DEVELOPMENT PROGRAM

Session 21: Angular Form & Agile & Git

OUTLINE

- Angular
- Working with forms
  - Template-driven forms
  - Reactive Forms

FORMS

- Handling user input with forms is the cornerstone of many common application

- With form, we enables users to log in, to update profile, to enter sensitive information, and to perform many other data-entry task

- Angular provides two different approaches to handling user input through forms: reactive and template-driven.

- Both way capture user input events from the **view**, validate user input, create form model and data model to update, and provide a way to track use changes

REACTIVE VS TEMPLATE-DRIVEN

- Reactive forms and template-driven forms process and manage form data differently. Each approach offers different advantages.

- Reactive forms provide direct, explicit access to the underlying forms object model. It is more robust, more scalable, reusable and testable. If forms are important part of your application, chose reactive forms.

- Template-driven forms rely on directives in the template to create and manipulate the underlying object model. When we only need to add a simple form to our app, such as submit feedback (from quiz project), we can use template-driven form

- Template driven form is easy to add, but they don't scale as well as reactive forms.

REACTIVE VS TEMPLATE-DRIVEN

	Reactive	Template driven
Setup of form model	Explicit, created in component class	Implicit, created with directive
Data model	Structured, and immutable	Unstructured and mutable
predictability	Synchronous	Asynchronous
Form validation	Functions	Directive

TEMPLATE DRIVEN

- In Template driven approach is the easiest way to build the Angular 2 forms. The logic of the form is placed in the template.
- It allows us to create sophisticated looking forms easily without writing any JS or TS code
- The FormsModule contains all the form directives and constructs for working with forms.
- To use template-driven form we first need to import FormsModule

IMPORT FORMSMODULE

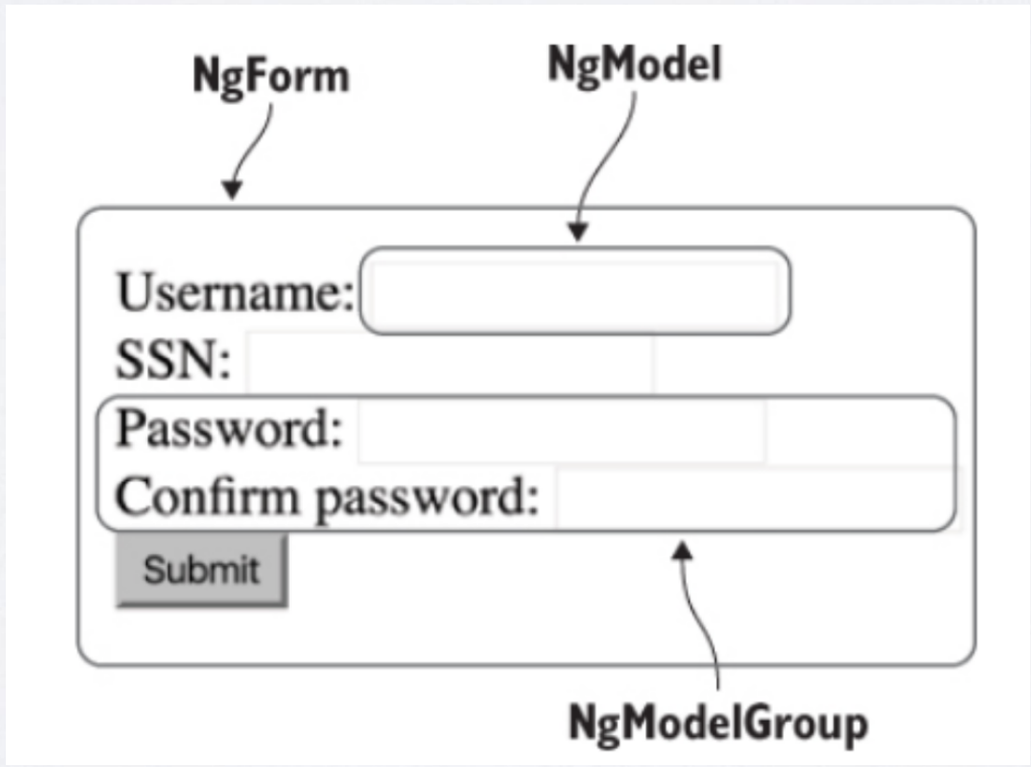
src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { HeroFormComponent } from './hero-form/hero-form.component';

@NgModule({
  imports: [
    BrowserModule,
    CommonModule,
    FormsModule
  ],
```

TEMPLATE DRIVEN

- Here we'll briefly describe the three main directives from FormsModule: NgModel, NgModelGroup, and NgForm.

- We'll show how they can be used in the template and highlight their most important features.



NG-FORM

- NgForm
  - NgForm is the directive that represents the entire form.
  - It's automatically attached to every <form> element.
  - NgForm intercepts the standard HTML form's submit event and prevents automatic form submission.
  - ngSubmit event we use to submit the form data.

src/app/hero-form/hero-form.component.html (ngSubmit)

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

NG-MODEL

- NgModel
  - In the context of the Forms API, NgModel represents a single field on the form.
- When you use [(ngModel)] on an element, you must define a name attribute for that element. Angular uses the assigned name to register the element with the NgForm directive attached to the parent <form> element.

src/app/hero-form/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
```

NG-MODEL-GROUP

- NgModelGroup
  - NgModelGroup represents a part of the form and allows you to group form fields together.
  - All the child fields of NgModelGroup become properties on the nested object.

```
<div ngModelGroup="name" #nameCtrl="ngModelGroup">
  <input name="first" [ngModel]="name.first" minlength="2">
  <input name="middle" [ngModel]="name.middle" maxlength="2">
  <input name="last" [ngModel]="name.last" required>
</div>
```

INITIALIZE MAPPING

- The simplest way is to initialize value of the form element in Component class and Set up one way or two way binding in the Template.
- The correct way is to create an Angular Form is to build a model class. Let us build a model for our Form.

INITIALIZE MAPPING

src/app/hero.ts

```
export class Hero {

  constructor(
    public id: number,
    public name: string,
    public power: string,
    public alterEgo?: string
  ) {  }

}
```

TEMPLATE DRIVEN VALIDATION

- We can also validate the input in template-drive style. We can use built-in and customized validator

- Built-in Validators

- The Angular Built-in validators use the HTML5 validation attributes like required, minlength, maxlength & pattern.

  - required — There must be a value

  - minlength — The number of characters must be more than the value of the attribute.

  - maxlength — The number of characters must not exceed the value of the attribute.

  - pattern — The value must match the pattern.

TEMPLATE DRIVEN VALIDATION

- Adding Required Validator
  - This can be done by adding the required attribute as shown in below and marks the input as a required field.

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
```

- Disable Submit button
  - We need to disable the submit button if our form is not valid.

src/app/hero-form/hero-form.component.html (submit-button)

```
<button type="submit" class="btn btn-success"
[disabled]="!heroForm.form.valid">Submit</button>
```

TEMPLATE DRIVEN VALIDATION

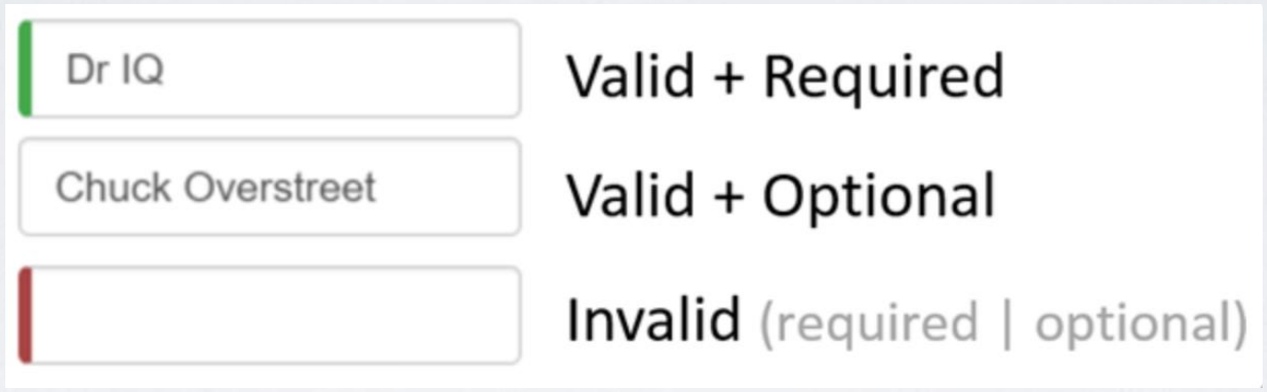
- Track control state and validity with ngModel
  - Using ngModel in a form gives you more than just two-way data binding.
  - It also tells you if the user touched the control, if the value changed, or if the value became invalid.
  - The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state.
  - You can leverage those class names to change the appearance of the control.

FORM CONTROL STATE

State	Class if true	Class if false
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

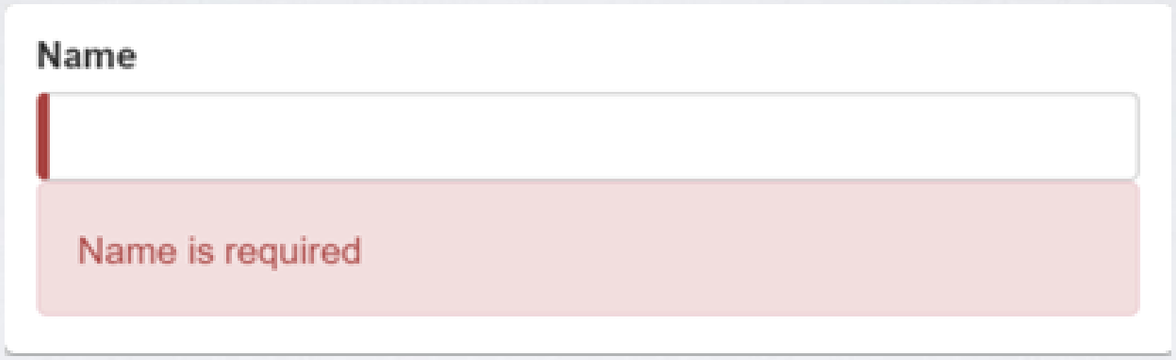
VALIDATION

- Add custom CSS for visual feedback




```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

SHOW AND HIDE VALIDATION ERROR MESSAGES



src/app/hero-form/hero-form.component.html (excerpt)

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
  Name is required
</div>
```

REACTIVE FORM

- In Reactive Forms (also known as Model Driven Forms) the form is built in the component class, unlike Template Driven Model where the model is built-in templates.
- Advantages of Reactive Forms
  - The Angular 2 Model Driven Forms enable us to test our forms without being required to rely on end-to-end tests.
  - The model driven Forms allows you to have an ability to setup validation rules in code rather than as directive in Template.
  - You can subscribe to field value changes in your code. Since everything happens in Component class, you can easily write unit Tests.

IMPORT REACTIVE-FORMS-MODULE

- To use reactive form controls, import ReactiveFormsModule from the @angular/forms package and add it to your NgModule's imports array.

src/app/app.module.ts (excerpt)

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


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

FORM CONTROL

- A FormControl represents a single input field in an Angular form.

- The FormControl is an object that encapsulates all information related to the single input element

- It tracks the value and validation status of its input

src/app/name-editor/name-editor.component.ts

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

REGISTER THE CONTROL IN THE TEMPLATE

- After you create the control in the component class, you must associate it with a form control element in the template.

- Update the template with the form control using the formControl binding provided by FormControlDirective, which is also included in the ReactiveFormsModule

src/app/name-editor/name-editor.component.html

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">
```

FORM GROUP

- FormGroup is a group of FormControl Instances.
- Forms often have more than one field. A form group defines a form with a fixed set of controls that you can manage together

src/app/profile-editor/profile-editor.component.ts (form group)

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

FORM GROUP

- Like form control, we bind **formGroup** to a form html tag
- For each of the **formControl** inside **formGroup**, we use **formControlName** to map them to an input tag
- Since the **formGroup** is bounded to the form html tag, we can also bind the submit event to any method in our component
- To implement submit event, we use (ngSubmit).

src/app/profile-editor/profile-editor.component.html (template form group)

```
<form [formGroup]="profileForm">

  <label for="first-name">First Name: </label>
  <input id="first-name" type="text" formControlName="firstName">

  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text" formControlName="lastName">

</form>
```

NESTED FORM GROUP

- Form groups can accept both individual form control instances or other form group instances as input.
- This allows us to nested form groups and make it easier to implement logic and maintain the logical group.

src/app/profile-editor/profile-editor.component.ts (nested form group)

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

NESTED FORM GROUP

- The nested Form group address is mapped to the div html element using **formGroupName**

src/app/profile-editor/profile-editor.component.html (template nested form group)

```
<div formGroupName="address">
  <h2>Address</h2>

  <label for="street">Street: </label>
  <input id="street" type="text" formControlName="street">

  <label for="city">City: </label>
  <input id="city" type="text" formControlName="city">

  <label for="state">State: </label>
  <input id="state" type="text" formControlName="state">

  <label for="zip">Zip Code: </label>
  <input id="zip" type="text" formControlName="zip">
</div>
```

FORM ARRAY

- FormArray is an alternative to FormGroup for managing any number of unnamed controls. We can dynamically insert and remove controls from FormArray instances.

- The benefit is we don’t need to define a key for each control by name, so this is a great option if you don’t know the number of child values in advance

- Because FormArray does not define key, we are not able to distinguish each of the FormControl. So, this object is used when we have a group of similar information, ex. Phone number/alias.

FORM ARRAY

- To use FormArray:
  - Import FormArray
  - Define a FormArray Control
  - Access the FormArray control with getter method
  - Display the form array in a template

FORM ARRAY

src/app/profile-editor/profile-editor.component.ts (aliases form array)

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
  aliases: this.fb.array([
    this.fb.control('')
  ])
});
```

src/app/profile-editor/profile-editor.component.ts (add alias)

```
addAlias() {
  this.aliases.push(this.fb.control(''));
}
```

src/app/profile-editor/profile-editor.component.html (aliases form array template)

```
<div formArrayName="aliases">
  <h2>Aliases</h2>
  <button (click)="addAlias()" type="button">+ Add another alias</button>

  <div *ngFor="let alias of aliases.controls; let i=index">
    <!-- The repeated alias template -->
    <label for="alias-{{ i }}">Alias:</label>
    <input id="alias-{{ i }}" type="text" [formControlName]="i">
  </div>
</div>
```

FORM BUILDER

- When we are dealing with multiple forms, it can be convenient to use **FormBuilder** service to generate the controls

- To use FormBuilder:
  - Import FormBuilder class
  - Inject the service
  - Generate the form content

src/app/profile-editor/profile-editor.component.ts (form builder)

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

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

FORM BUILDER

- Form Group has three different methods, these are all factory methods for generating instances in our component classes including form controls, form groups and form arrays

- Control(): generate a FormControl
- Group(): generate a FormGroup
- Array(): generate a FormArray

FORM VALIDATION

- Form validation is used to ensure that user input is complete and correct.

- To add the form validation, we do the following steps:

  - Import a validator function in your form component

  - Add the validator to the field in the form

  - Add logic to handle the validation status

Making the firstName input a required field

src/app/profile-editor/profile-editor.component.ts (required validator)

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
});
```

ANY QUESTIONS?