



# ANGULAR 6

Lesson 06



The slide features a dark blue background with large, flowing red shapes on the left and bottom. On the right side, there is a red L-shaped graphic element. The title is centered in white, bold, uppercase letters.

# TEMPLATE-DRIVEN AND REACTIVE FORMS

Objectives

# Template-Driven and Reactive Forms

- Template-Driven vs Reactive Approach
- Understanding Form State
- Built-in Validators & Using HTML5 Validation
- Grouping Form Controls
- FormGroup, FormControl, FormBuilder, FormArray
- Forms with Reactive Approach
- Predefined Validators & Custom Validators
- Async Validators
- Showing validation errors



## Template-Driven vs Reactive Approach



An abstract graphic featuring a dark blue background with several red shapes. On the left, there is a large, flowing red shape that curves upwards and then downwards. In the top-left corner, there is a smaller red shape. On the right side, there is a red L-shaped element consisting of a vertical bar and a horizontal bar meeting at a right angle.

# TEMPLATE-DRIVEN FORMS

# Template Driven Forms



- Forms build by writing templates in the Angular template syntax with the form-specific directives and techniques are called as Template Driven Forms.
- The app component doesn't need to do much since the form fields and validators are defined in the template when using Angular template-driven forms
- The component defines a model object which is bound to the form fields in the template in order to give you access to the data entered into the form from the app component
- 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.
  - *Angular2 "infers" the `FormGroup` from HTML Code*
  - *Form data is passed via `ngSubmit()`*



The user should be able to submit this form after filling it in. The *Submit* button at the bottom of the form does nothing on its own, but it will trigger a form submit because of its type (`type="submit"`).

A "form submit" is useless at the moment. To make it useful, bind the form's `ngSubmit` event property to the hero form component's `onSubmit()` method: `src/app/hero-form/hero-form.component.html (ngSubmit)content_copy<form (ngSubmit)="onSubmit()" #heroForm="ngForm">`

The `<input>` element carries the HTML validation attributes: `required` and [minlength](#). It also carries a custom validator directive, `forbiddenName`. For more information, see [Custom validators](#) section.

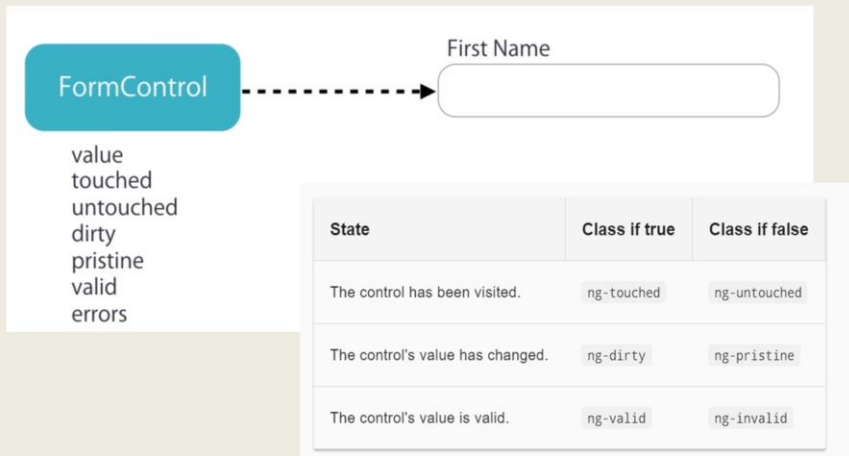
`#name="ngModel"` exports [NgModel](#) into a local variable called `name`. [NgModel](#) mirrors many of the properties of its underlying [FormControl](#) instance, so you can use this in the template to check for control states such as `valid` and `dirty`. For a full list of control properties, see the [AbstractControl](#) API reference.

The `*ngIf` on the `<div>` element reveals a set of nested message divs but only if the `name` is invalid and the control is either `dirty` or `touched`.

Each nested `<div>` can present a custom message for one of the possible validation

errors. There are messages for required, [minlength](#), and forbiddenName.

## Understanding Form State



The ng-valid/ng-invalid pair is the most interesting, because you want to send a strong visual signal when the values are invalid.

hide the message when the control is valid or pristine; "pristine" means the user hasn't changed the value since it was displayed in this form.

This user experience is the developer's choice. Some developers want the message to display at all times. If you ignore the pristine state, you would hide the message only when the value is valid. Some developers want the message to display only when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. You'll see the significance of this choice when you add a new hero to the form.

**Required** - The form field is mandatory

**Maxlength** - Maximum number of characters in the field.

**Minlength** - Minimum number of characters in the field.

**Pattern** - Regular expression to match the input values and validate.

**Custom Validator** - Writing confirm password fields.

Dr IQ	TODO: remove this: form-control ng-untouched ng-pristine ng-valid	Untouched
Dr IQ	TODO: remove this: form-control ng-pristine ng-valid ng-touched	Touched
Dr IQ///	TODO: remove this: form-control ng-valid ng-touched ng-dirty	Changed
	TODO: remove this: form-control ng-touched ng-dirty ng-invalid	Invalid



# Validation

```
<form #empForm=ngForm (ngSubmit)="getData(empForm)">
  <table>
    <tr>
      <td>Product ID</td>
      <td><input required id="eid" name ="id" [(ngModel)]= "emp.eId"
        type="text" #idcontrol="ngModel"/>
        <span *ngIf="idcontrol.invalid && idcontrol.touched" > ID is
        required</span>
      </td>
    </tr>
    <tr>
      <td>Product Name</td>
      <td><input required id="ename" name ="empname"
        [(ngModel)]= "emp.eName" type="text"
        #namecontrol="ngModel"/></td>
      <span *ngIf="namecontrol.invalid && namecontrol.touched" >
        Name is required</span>
      </td>
    </tr>
  </table>
</form>
```



## Built-in Validators & Using HTML5 Validation



## Grouping Form Controls



# FormGroup, FormControl, FormBuilder, FormArray





Add instructor notes  
here.

Cap

# Demo

- Demo Template Driven Forms





Add the notes here.

An abstract graphic featuring a dark blue background with several red shapes. On the left, a red shape curves upwards from the bottom. In the bottom right, there is a red L-shaped element. The text "REACTIVE FORMS" is centered in white, uppercase letters.

# REACTIVE FORMS

## Forms with Reactive Approach

- The app component defines the form fields and validators for our registration form using an Angular FormBuilder to create an instance of a FormGroup that is stored in the registerForm property.
- The registerForm is then bound to the form in the template below using the [formGroup] directive.
- Also need to be added a getter 'f' as a convenience property to make it easier to access form controls from the template. So for example you can access the email field in the template using f.email instead of registerForm.controls.email.



# Forms with Reactive Approach



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

@Component({
  selector: 'app',
  templateUrl: 'app.component.html'
})
export class AppComponent implements OnInit {
  registerForm: FormGroup;
  submitted = false;
  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(6)]]
    });
  }
  // convenience getter for easy access to form fields
  get f() { return this.registerForm.controls; }
  onSubmit() {
    this.submitted = true;
    // stop here if form is invalid
    if (this.registerForm.invalid) {
      return;
    }
    alert("SUCCESS!! :-)")
  }
}
```





## Forms with Reactive Approach

```
<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label>First Name</label>
    <input type="text" formControlName="firstName" class="form-control" [ngClass]="{'is-invalid': submitted
    && f.firstName.errors}" />
    <div *ngIf="submitted && f.firstName.errors" class="invalid-feedback">
      <div *ngIf="f.firstName.errors.required">First Name is required</div> </div> </div>
    <div class="form-group">
      <label>Last Name</label>
      <input type="text" formControlName="lastName" class="form-control" [ngClass]="{'is-invalid':
      submitted && f.lastName.errors}" />
      <div *ngIf="submitted && f.lastName.errors" class="invalid-feedback">
        <div *ngIf="f.lastName.errors.required">Last Name is required</div> </div> </div>
    <div class="form-group">
      <label>Email</label>
      <input type="text" formControlName="email" class="form-control" [ngClass]="{'is-invalid': submitted &&
      f.email.errors}" />
      <div *ngIf="submitted && f.email.errors" class="invalid-feedback">
        <div *ngIf="f.email.errors.required">Email is required</div>
        <div *ngIf="f.email.errors.email">Email must be a valid email address</div> </div> </div>
    <div class="form-group">
      <label>Password</label>
      <input type="password" formControlName="password" class="form-control" [ngClass]="{'is-invalid':
      submitted && f.password.errors}" />
      <div *ngIf="submitted && f.password.errors" class="invalid-feedback">
        <div *ngIf="f.password.errors.required">Password is required</div>
        <div *ngIf="f.password.errors.minlength">Password must be at least 6 characters</div></div> </div>
    <div class="form-group"> <button [disabled]="loading" class="btn btn-primary">Register</button>
    </div> </form>
```



## Predefined Validators & Custom Validators



## Async Validators



## Showing validation errors

