FORMS

LESSON 04

SWAFE-01

FORMS IN ANGULAR

OVERVIEW

- Handling user input with forms is the cornerstone of many web applications
- Angular provides two different approaches:
    - Template-driven forms
    - Reactive forms
- Reactive and template-driven forms process and manage data differently

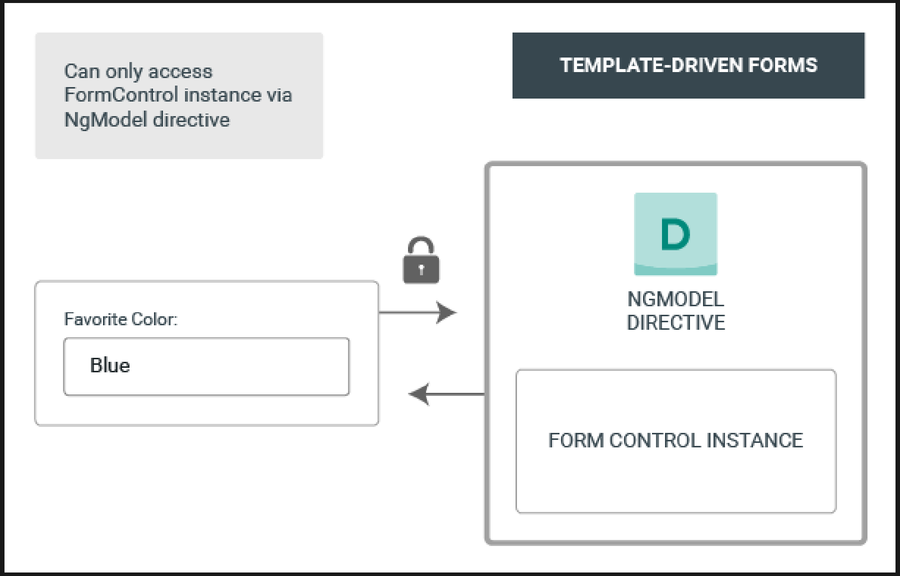
CHOOSING AN APPROACH

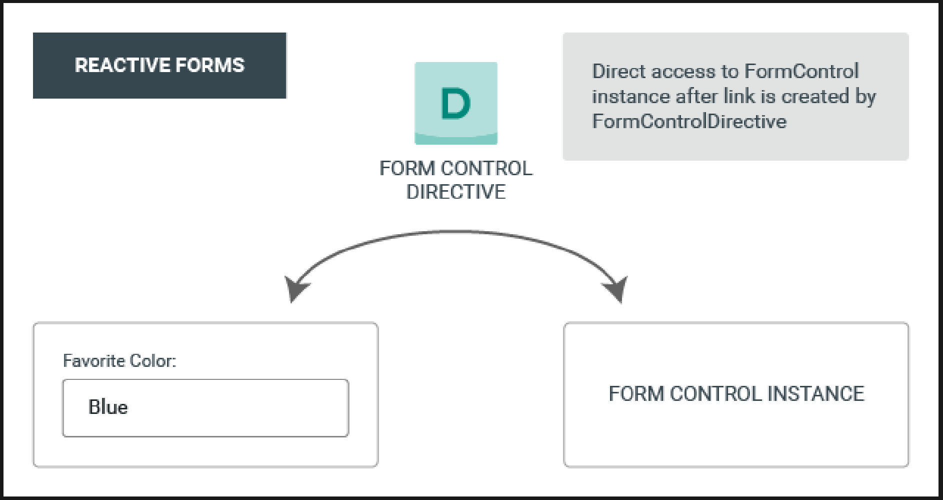
- Reactive forms
  - They provide a direct explicit access to the underlying forms object model
  - More scalable, reuseable and testable
  - Choose reactive forms if forms are a key part of the application
- Template-driven forms
  - Easier to implement
  - Choose if the requirements and logic can be managed solely in the template

COMMON FORM FOUNDATION CLASSES

- Both reactive and template-driven forms are built on the following base classes
  - `FormControl` tracks the value and validation status of an individual form control
  - `FormGroup` tracks the same values and status for a collection of form controls
  - `FormArray` tracks the same values and status for an array of form controls
  - `ControlValueAccessor` creates a bridge between Angular FormControl instances and native DOM elements

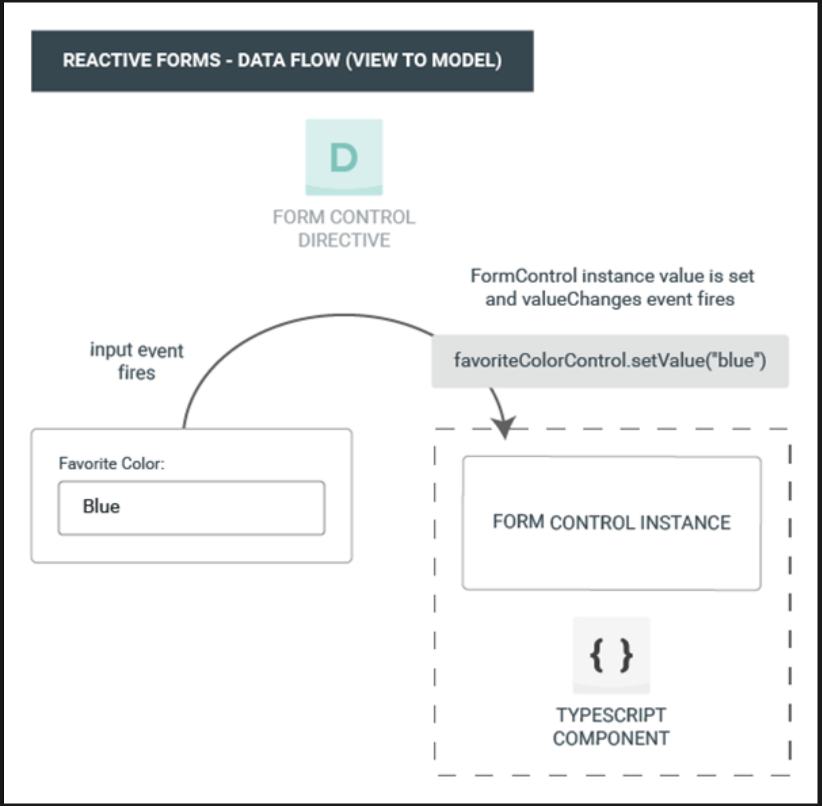
DATA ACCESS


Can only access FormControl instance via NgModel directive
TEMPLATE-DRIVEN FORMS
Favorite Color:
Blue
D
NGMODEL DIRECTIVE
FORM CONTROL INSTANCE


REACTIVE FORMS
D
FORM CONTROL DIRECTIVE
Direct access to FormControl instance after link is created by FormControlDirective
Favorite Color:
Blue
FORM CONTROL INSTANCE

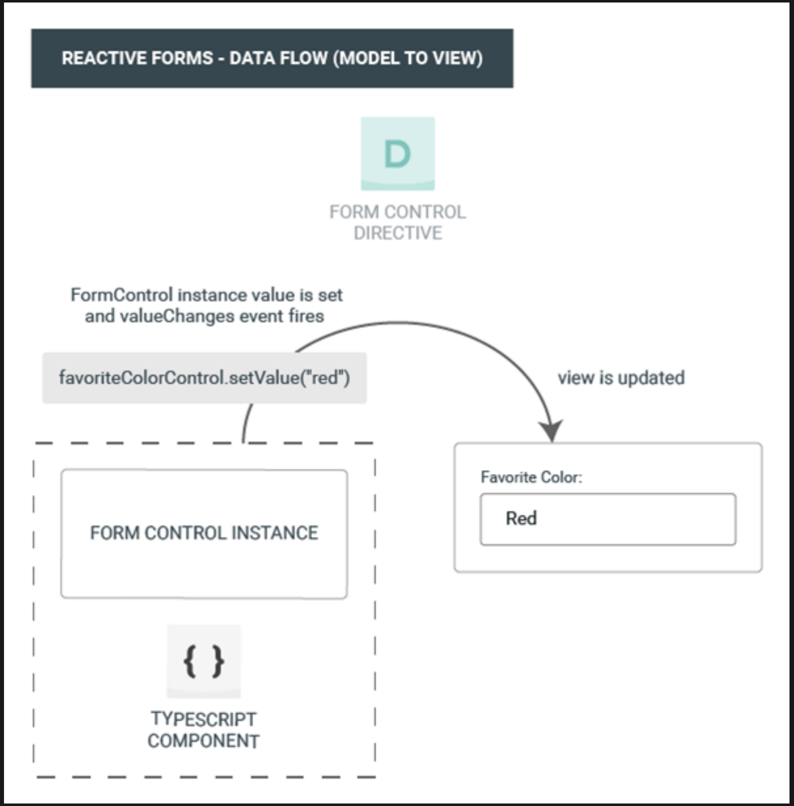
DATA FLOW

REACTIVE (VIEW TO MODEL)


REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)
D
FORM CONTROL DIRECTIVE
FormControl instance value is set and valueChanges event fires
input event fires
favoriteColorControl.setValue("blue")
Favorite Color:
Blue
FORM CONTROL INSTANCE
{ }
TYPESCRIPT COMPONENT

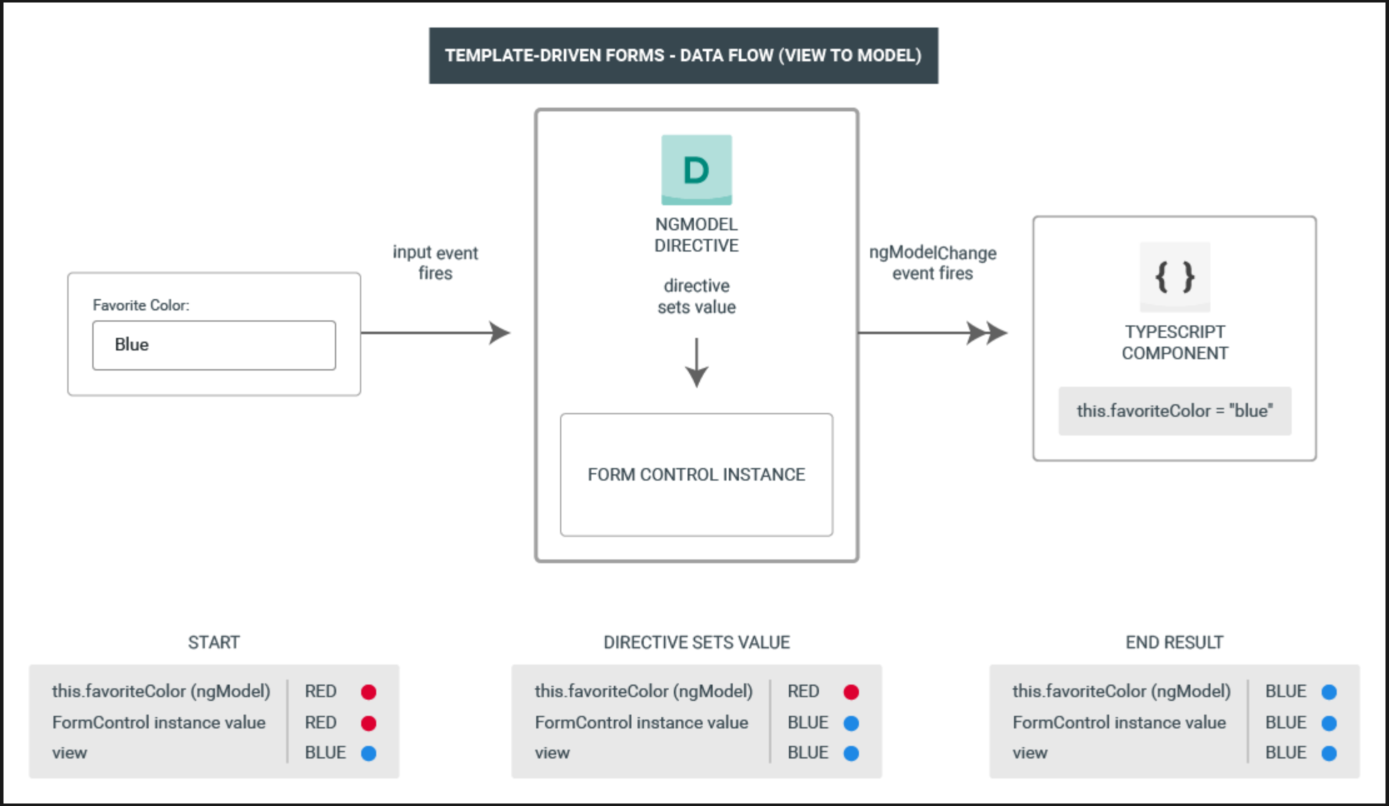
DATA FLOW

REACTIVE (MODEL TO VIEW)


REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)
D
FORM CONTROL DIRECTIVE
FormControl instance value is set and valueChanges event fires
favoriteColorControl.setValue("red")
view is updated
Favorite Color:
Red
FORM CONTROL INSTANCE
{ }
TYPESCRIPT COMPONENT

DATA FLOW

TEMPLATE-DRIVEN (VIEW TO MODEL)



START

this.favoriteColor (ngModel)	RED	🔴
FormControl instance value	RED	🔴
view	BLUE	🔵

DIRECTIVE SETS VALUE

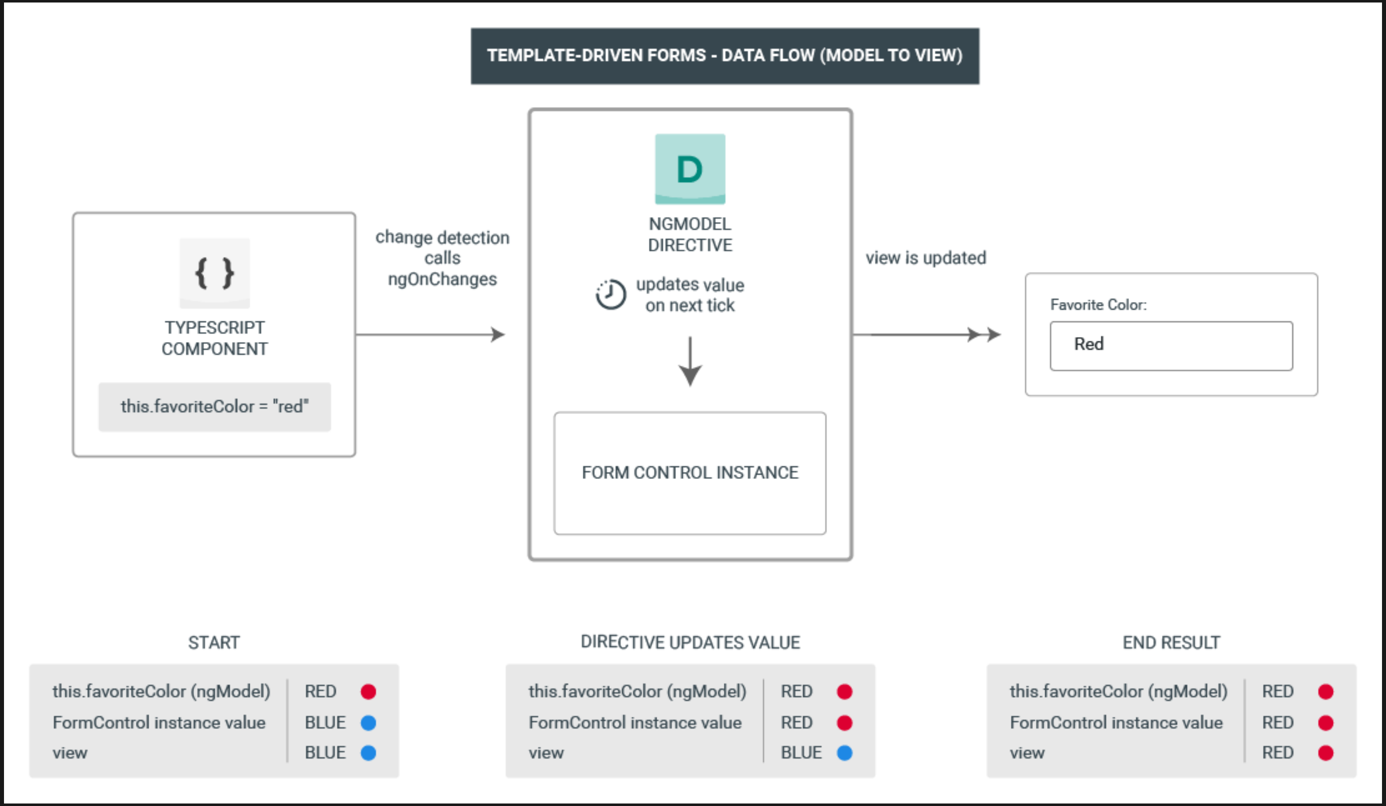
this.favoriteColor (ngModel)	RED	🔴
FormControl instance value	BLUE	🔵
view	BLUE	🔵

END RESULT

this.favoriteColor (ngModel)	BLUE	🔵
FormControl instance value	BLUE	🔵
view	BLUE	🔵

DATA FLOW

TEMPLATE-DRIVEN (MODEL TO VIEW)



TEMPLATE-DRIVEN FORMS - DATA FLOW (MODEL TO VIEW)
{ }
TYPESCRIPT COMPONENT
this.favoriteColor = "red"
change detection calls ngOnChanges
D
NGMODEL DIRECTIVE
updates value on next tick
FORM CONTROL INSTANCE
view is updated
Favorite Color:
Red

START

this.favoriteColor (ngModel)	RED 🔴
FormControl instance value	BLUE 🔵
view	BLUE 🔵

DIRECTIVE UPDATES VALUE

this.favoriteColor (ngModel)	RED 🔴
FormControl instance value	RED 🔴
view	BLUE 🔵

END RESULT

this.favoriteColor (ngModel)	RED 🔴
FormControl instance value	RED 🔴
view	RED 🔴

TEMPLATE-DRIVEN FORMS

OVERVIEW

- Control elements are bound to data properties
- Implicitly creates data model
- Template directives
    - `NgForm`–Creates a top-level instance and binds it to a form to track aggregate form value and validation status
    - `NgModel`—used to mark HTML elements as part of the data model (different context that two-way data binding from Lesson 01)
    - `NgModelGroup`—represents a part of the form. Used to group elements together
- Template-driven forms rely on mutability of the data model

TEMPLATE-DRIVEN FORM – CLASS

```typescript
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';
import { Observable } from 'rxjs';
import { Class, WarcraftService } from 'warcraft';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  classes$: Observable<Class[]>;

  constructor(warcraftService: WarcraftService) {
    this.classes$ = warcraftService.getClasses()
  }

  onSubmit(form: NgForm) {
    console.log(form.value)
```

examples/lesson04-forms/projects/template-driven/src/app/app.component.ts

TEMPLATE-DRIVEN FORM – TEMPLATE

```
<h1>Template-driven forms</h1>
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <div class="form-wrapper">
    <label for="first_name">First name</label>
    <input type="text" name="first_name" ngModel>
    <label for="last_name">Last name</label>
    <input type="text" name="last_name" ngModel>
    <hr />
    <label for="phone">Phone</label>
    <input type="text" name="phone" ngModel>
    <label for="email">E-mail</label>
    <input type="text" name="email" ngModel>
    <hr />
    <label for="class">Class</label>
    <select name="class" ngModel>
      <option *ngFor="let class of classes$ | async" [ngValue]="class">{{ clas
    </select>
    <hr />
    <button type="submit">Submit</button>
```

examples/lesson04-forms/projects/template-driven/src/app/app.component.html

REACTIVE FORMS

OVERVIEW

- The reactive directives come with `ReactiveFormsModule`
- Directives
  - `formGroup`—binds to an instance of `FormGroup` that represents the entire form model
  - `formGroupName`—used when binding to nested `FormGroup` objects
  - `formControl`—used for individual controls without the need to create a model, but want Forms API features
  - `formControlName`—used when binding to nested `FormControl` objects
- Define data model in component class

REACTIVEFORM — CLASS

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';
import { Observable } from 'rxjs';
import { Class, WarcraftService } from 'warcraft';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  profileForm = this.formBuilder.group({
    first_name: [''],
    last_name: [''],
    phone: [''],
    email: [''],
    class: [{
      name: '', roles: [{ name: ''}]
```

examples/lesson04-forms/projects/reactive/src/app/app.component.ts

REACTIVE – TEMPLATE

```html
<h1>Reactive forms</h1>
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
  <div class="form-wrapper">
    <label for="first_name">First name</label>
    <input type="text" formControlName="first_name">
    <label for="last_name">Last name</label>
    <input type="text"  formControlName="last_name">
    <hr />
    <label for="phone">Phone</label>
    <input type="text" formControlName="phone">
    <label for="email">E-mail</label>
    <input type="text"  formControlName="email">
    <hr />
    <label for="class">Class</label>
    <select formControlName="class" [compareWith]="compareClasses">
      <option *ngFor="let class of classes$ | async" [ngValue]="class">{{ clas
    </select>
    <hr />
    <button type="button" (click)="onAutofill()">Autofill</button>
```

examples/lesson04-forms/projects/reactive/src/app/app.component.html

REACTIVEFORM – CLASS

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';
import { Observable } from 'rxjs';
import { Class, WarcraftService } from 'warcraft';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  profileForm = this.formBuilder.group({
    first_name: [''],
    last_name: [''],
    phone: [''],
    email: [''],
    class:[''],
  })
```

examples/lesson04-forms/projects/reactive/src/app/app.component.ts

DYNAMIC FORMS

- Some use cases requires adding/removing controls based on input and/or state
- Use `FormArray` to manage any number of unnamed controls
- A great option when the number of controls is not known in advance
    - Use `push(control: AbstractControl)` to add controls
    - Use `removeAt(index: number)` to remove controls
- Bind in template with the `formArrayName` directive

DYNAMIC FORM – CLASS

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  profileForm = this.formBuilder.group({
    first_name: [''],
    last_name: [''],
    loot: this.formBuilder.array([''])
  })

  constructor(private formBuilder: FormBuilder) { }

  onSubmit() {
```

examples/lesson04-forms/projects/dynamic-forms/src/app/app.component.ts

DYNAMIC FORM – TEMPLATE

```html
<h1>Dynamic forms</h1>
<form [formGroup]='profileForm' (ngSubmit)="onSubmit()">
  <div class="form-wrapper">
    <label for="first_name">First name</label>
    <input type="text" formControlName="first_name">
    <label for="last_name">Last name</label>
    <input type="text"  formControlName="last_name">
    <hr />
    <div formArrayName="loot">
      <button type="button" (click)="addLoot()">Add more loot</button>
      <div *ngFor="let loot of loot.controls; let i=index">
        <input id="loot-{{i}}" type="text" [formControlName]="i" />
        <button (click)="removeLoot(i)">x</button>
      </div>
    </div>
    <hr />
    <button type="submit">Submit</button>
  </div>
</form>
```

examples/lesson04-forms/projects/dynamic-forms/src/app/app.component.html

UPDATING `FormControl` VALUES

- Reactive forms methods provides two methods to change values
  - `setValue()`—updates the values in the form data model. Must match the complete form date model
  - `patchValue()`—updates selected properties in the form data model. Used to do partial updates to the form data model
- Provide flexibility to change control values without user interaction

FORM VALIDATION

OVERVIEW

- Improve overall data quality
  - Accuracy—are users providing a usable value?
  - Completeness—are they providing all values needed?
- Display useful messages to users
  - Guide the user to input valid data
- Every time the value of a form control changes
  - Angular runs validation
  - Generates a list of validation errors
  - Results in `VALID` or `INVALID`
- The class `Validators` from Forms API provide built-in validators for the most common use cases

BUILT-IN VALIDATORS

- `min` / `max` —value must be greater/less than or equal to the provided number
- `required` —value must be non-empty
- `requiredTrue` —value must be true
- `email` —value must pass an email validation test
- `minLength` / `maxLength` —value must be greater/less than or equal to the provided number. Intented for types with numeric `length` value
- `pattern` —value must match a regex pattern
- `null validator` —Validator that performs no operation
- `compose` / `composeAsync` —compose multiple (async) validators into a single function that returns the union of the individual error maps

TEMPLATE-DRIVEN FORM

```
<h1>Template-driven forms</h1>
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <div class="form-wrapper" ngModelGroup="name" appFullName #name="ngModelGrou
    <label for="first_name">First name</label>
    <input type="text" name="first_name" ngModel >
    <label for="last_name">Last name</label>
    <input type="text" name="last_name" ngModel>
    <div *ngIf="name.invalid && (name.dirty || name.touched)">
      <div *ngIf="name.errors?.must_be_set">
        {{ name.errors?.must_be_set }}
      </div>
    </div>
  </div>
  <hr />
  <label for="phone">Phone</label>
  <input type="text" name="phone" ngModel>
  <label for="email">E-mail</label>
  <input type="text" name="email" ngModel #email="ngModel" minlength="5" requi
  <div *ngIf="email.invalid && (email.dirty || email.touched)">
```

examples/lesson04-forms/projects/template-driven-validation/src/app/app.component.html

REACTIVE FORM

```
import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, ValidationErrors, Validators } f
import { Observable } from 'rxjs';
import { Class, WarcraftService } from 'warcraft';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  profileForm = this.formBuilder.group({
    name: this.formBuilder.group({
      first_name: [''],
      last_name: [''],
    }, { validators: this.fullNameRequired, updateOn: 'change' }),
    phone: ['', Validators.nullValidator],
    email: ['', [Validators.required, Validators.email, Validators.minLength(5
```

examples/lesson04-forms/projects/reactive-validation/src/app/app.component.ts

CUSTOM VALIDATORS

- Apply application-specific validation
- Cross-field validation
  - Validate values in two different form controls in a form
  - Mutually incompatible—Select only one of two options
  - Dependencies—Select only an option, if another one is selected
- Asynchronous validators
  - Similar to their synchronous counterparts
  - They must return a `Promise` or an `Observable`
  - The observable must be finite (is has to complete at some point in time)
- Add directive for template-driven forms
  - Create a `Directive` and implement the `Validator` interface

TRIGGERING VALIDATION

- Angular will trigger validation whenever a form control changes per default
- This can be overridden with `updateOn` property
    - `change` —the value is checked as soon as it changes (*default*).
    - `blur` —the value is checked when the control loses focus
    - `submit` —the value is checked when the form is submitted
- Can be applied to individual form controls or complete forms

ANGULAR VS. HTML5 VALIDATION

- HTML5 offers native constraint validation
    - Disabled by Angular per default
- Add `ngNativeValidate` to the `<form>` element to use native validation in combination with the Angular-based validation

KEY DIFFERENCES

- Setup of form model
  - Template-driven Implicit, created by directives
  - Reactive Explicit, created in component class
- Data model
  - Template-driven Unstructured and mutable
  - Reactive Structured and immutable
- Data flow
  - Template-driven Asynchronous
  - Reactive Synchronous
- Form validation
  - Template-driven Directives
  - Reactive Functions