

FORMS

LECTURE 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

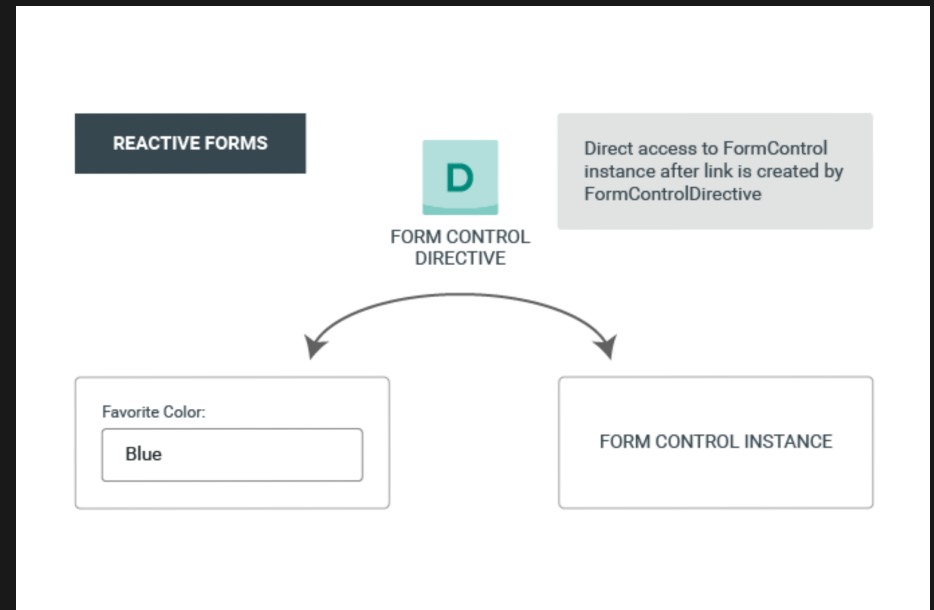
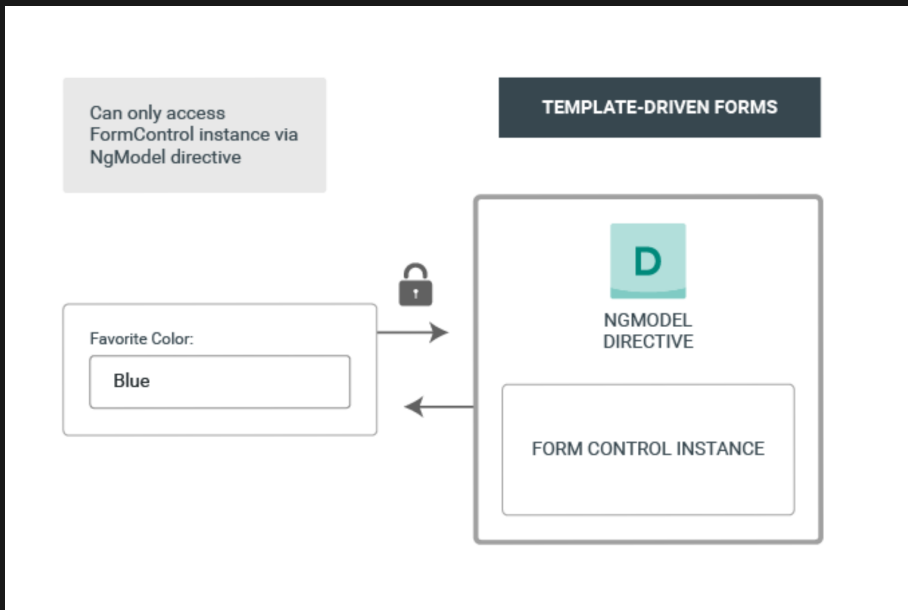
CHOOSING AN APPROACH

- 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

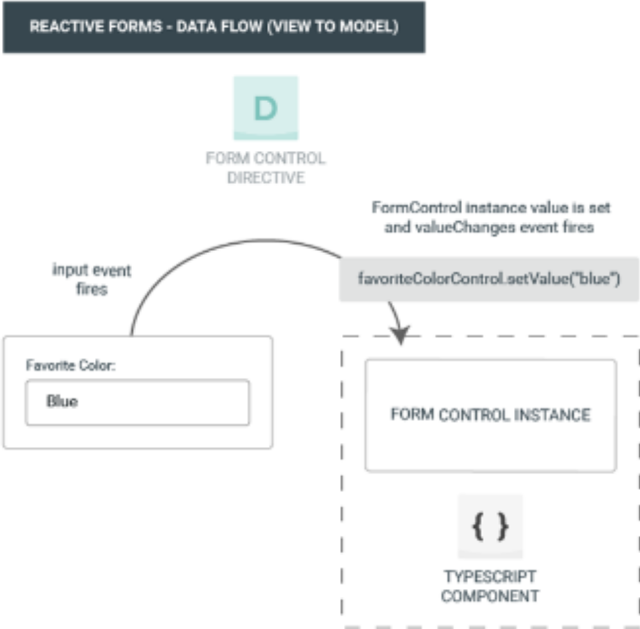
COMMON FORM FOUNDATION CLASSES

- 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



REACTIVE (VIEW TO MODEL)



- The user types a value into the input element, in this case the favorite color 'Blue'.
- The form input element emits an "input" event with the latest value.
- The control value accessor listening for events on the form input element immediately relays the new value to the `FormControl` instance.
- The `FormControl` instance emits the new value through the `valueChanges` observable.
- Any subscribers to the `valueChanges` observable receive the new value.

DATA FLOW

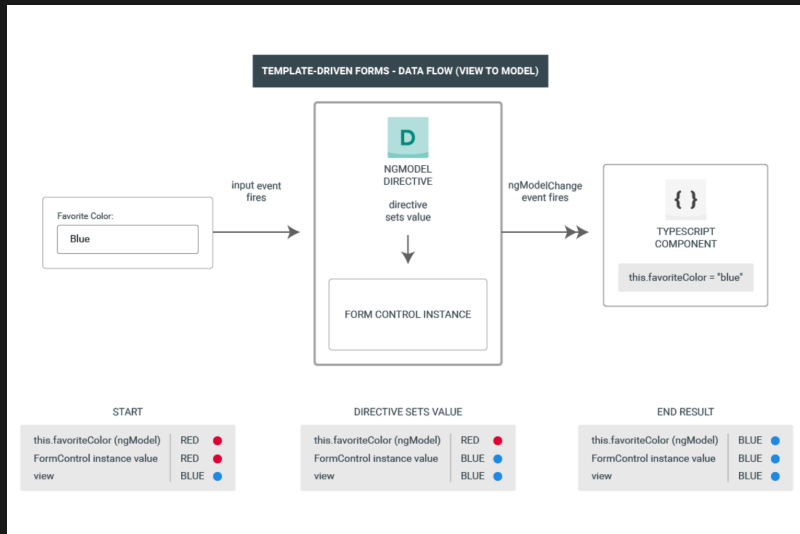
REACTIVE (MODEL TO VIEW)



- The user calls the `favoriteColorControl.setValue()` method, which updates the `FormControl` value.
- The `FormControl` instance emits the new value through the `valueChanges` observable.
- Any subscribers to the `valueChanges` observable receive the new value.
- The control value accessor on the form input element updates the element with the new value.

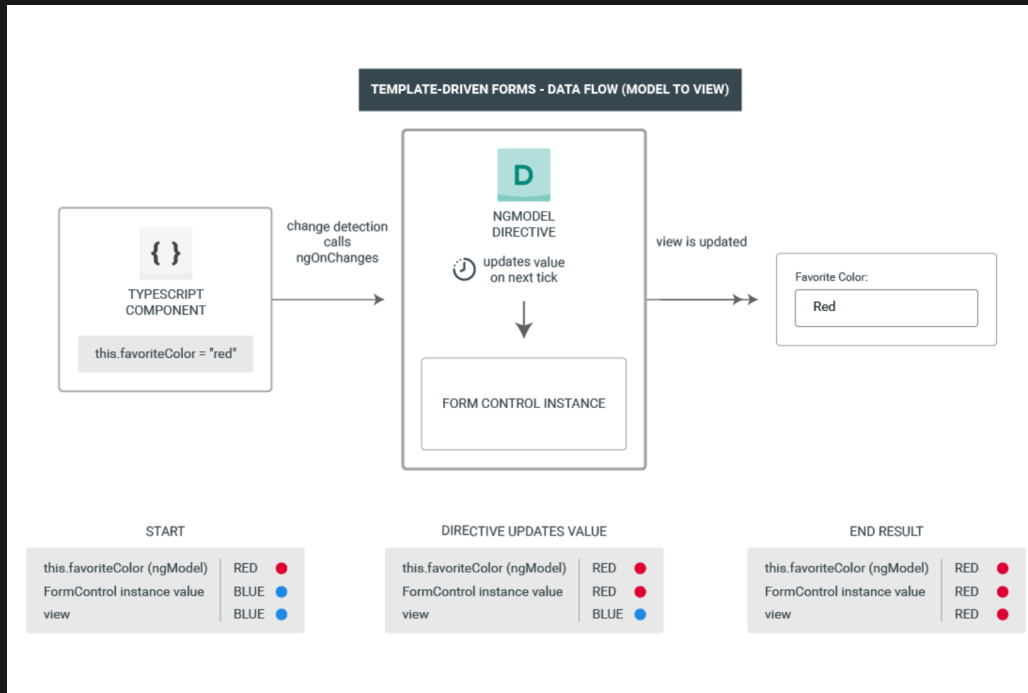
DATA FLOW

TEMPLATE-DRIVEN (VIEW TO MODEL)



DATA FLOW

TEMPLATE-DRIVEN (MODEL TO VIEW)



- The `favoriteColor` value is updated in the component
- Change detection begins
 - During change detection, the `ngOnChanges` lifecycle hook is called on the `NgModel` directive instance because the value of one of its inputs has changed
 - The `ngOnChanges()` method queues an async task to set the value for the internal `FormControl` instance
- Change detection completes
- On the next tick, the task to set the `FormControl` instance value is executed
 - The `FormControl` instance emits the latest value through the `valueChanges` observable
 - Any subscribers to the `valueChanges` observable receive the new value
- The control value accessor updates the form input element in the view with the latest `favoriteColor` value

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

```
1 import { Component } from '@angular/core';
2 import { NgForm } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { Class, WarcraftService } from 'warcraft';
5
6 @Component({
7   selector: 'app-root',
8   templateUrl: './app.component.html',
9   styleUrls: ['./app.component.scss']
10 })
11 export class AppComponent {
12   classes$: Observable<Class[]>;
13
14   constructor(warcraftService: WarcraftService) {
15     this.classes$ = warcraftService.getClasses()
16   }
17
18   onSubmit(form: NgForm) {
19     console.log(form.value)
```

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

TEMPLATE-DRIVEN FORM – TEMPLATE

```
1 <h1>Template-driven forms</h1>
2 <form #f="ngForm" (ngSubmit)="onSubmit(f)">
3   <div class="form-wrapper">
4     <label for="first_name">First name</label>
5     <input type="text" name="first_name" ngModel>
6     <label for="last_name">Last name</label>
7     <input type="text" name="last_name" ngModel>
8     <hr />
9     <label for="phone">Phone</label>
10    <input type="text" name="phone" ngModel>
11    <label for="email">E-mail</label>
12    <input type="text" name="email" ngModel>
13    <hr />
14    <label for="class">Class</label>
15    <select name="class" ngModel>
16      <option *ngFor="let class of classes$ | async" [ngValue]="class">{{ class
17    </select>
18    <hr />
19    <button type="submit">Submit</button>
20  </div>
21 </form>
```

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

REACTIVE FORMS

OVERVIEW

- The reactive directives come with `ReactiveFormsModule`
- Reactive 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

```
1 import { Component } from '@angular/core';
2 import { FormBuilder } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { Class, WarcraftService } from 'warcraft';
5
6 @Component({
7   selector: 'app-root',
8   templateUrl: './app.component.html',
9   styleUrls: ['./app.component.scss']
10 })
11 export class AppComponent {
12
13   profileForm = this.formBuilder.group({
14     first_name: [''],
15     last_name: [''],
16     phone: [''],
17     email: [''],
18     class:[''],
19   })
20 }
```

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

REACTIVE – TEMPLATE

```
1 <h1>Reactive forms</h1>
2 <form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
3   <div class="form-wrapper">
4     <label for="first_name">First name</label>
5     <input type="text" formControlName="first_name">
6     <label for="last_name">Last name</label>
7     <input type="text" formControlName="last_name">
8     <hr />
9     <label for="phone">Phone</label>
10    <input type="text" formControlName="phone">
11    <label for="email">E-mail</label>
12    <input type="text" formControlName="email">
13    <hr />
14    <label for="class">Class</label>
15    <select formControlName="class" [compareWith]="compareClasses">
16      <option *ngFor="let class of classes$ | async" [ngValue]="class">{{ class
17    </select>
18    <hr />
19    <button type="button" (click)="onAutofill()">Autofill</button>
```

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

REACTIVEFORM – CLASS

```
1 import { Component } from '@angular/core';
2 import { FormBuilder } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { Class, WarcraftService } from 'warcraft';
5
6 @Component({
7   selector: 'app-root',
8   templateUrl: './app.component.html',
9   styleUrls: ['./app.component.scss']
10 })
11 export class AppComponent {
12
13   profileForm = this.formBuilder.group({
14     first_name: [''],
15     last_name: [''],
16     phone: [''],
17     email: [''],
18     class:[''],
19   })
20 }
```

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

```
1 import { Component } from '@angular/core';
2 import { FormArray, FormBuilder } from '@angular/forms';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.scss']
8 })
9 export class AppComponent {
10
11   profileForm = this.formBuilder.group({
12     first_name: [''],
13     last_name: [''],
14     loot: this.formBuilder.array([''])
15   })
16
17   constructor(private formBuilder: FormBuilder) { }
18
19   onSubmit() {
```

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

DYNAMIC FORM – TEMPLATE

```
1 <h1>Dynamic forms</h1>
2 <form [formGroup]='profileForm' (ngSubmit)="onSubmit()">
3   <div class="form-wrapper">
4     <label for="first_name">First name</label>
5     <input type="text" formControlName="first_name">
6     <label for="last_name">Last name</label>
7     <input type="text" formControlName="last_name">
8     <hr />
9     <div formArrayName="loot">
10      <button type="button" (click)="addLoot()">Add more loot</button>
11      <div *ngFor="let loot of loot.controls; let i=index">
12        <input id="loot-{{i}}" type="text" [formControlName]="i" />
13        <button (click)="removeLoot(i)">x</button>
14      </div>
15    </div>
16    <hr />
17    <button type="submit">Submit</button>
18  </div>
19 </form>
```

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

UPDATING FORM CONTROL VALUES

- Reactive forms methods to change a form control's value programmatically
- There are two methods
 - `setValue()` —updates the values in the form data model. Must match the complete form data model
 - `patchValue()` —updates selected properties in the form data model. Used when partially updating form data model
- Provide developers with the 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

TEMPLATE-DRIVEN FORM

```
1 <h1>Template-driven forms</h1>
2 <form #f="ngForm" (ngSubmit)="onSubmit(f)">
3   <div class="form-wrapper" ngModelGroup="name" appFullName #name="ngModelGrou
4     <label for="first_name">First name</label>
5     <input type="text" name="first_name" ngModel >
6     <label for="last_name">Last name</label>
7     <input type="text" name="last_name" ngModel>
8     <div *ngIf="name.invalid && (name.dirty || name.touched)">
9       <div *ngIf="name.errors?.must_be_set">
10         {{ name.errors?.must_be_set }}
11       </div>
12     </div>
13   </div>
14   <hr />
15   <label for="phone">Phone</label>
16   <input type="text" name="phone" ngModel>
17   <label for="email">E-mail</label>
18   <input type="text" name="email" ngModel #email="ngModel" minlength="5" requi
19   <div *ngIf="email.invalid && (email.dirty || email.touched)">
```

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

REACTIVE FORM

```
1 import { Component } from '@angular/core';
2 import { FormBuilder, FormControl, FormGroup, ValidationErrors, Validators } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { Class, WarcraftService } from 'warcraft';
5
6 @Component({
7   selector: 'app-root',
8   templateUrl: './app.component.html',
9   styleUrls: ['./app.component.scss']
10 })
11 export class AppComponent {
12
13   profileForm = this.formBuilder.group({
14     name: this.formBuilder.group({
15       first_name: ['', Validators.required],
16       last_name: ['', Validators.required],
17     }, { validators: this.fullNameRequired, updateOn: 'change' }),
18     phone: ['', Validators.nullValidator],
19     email: ['', [Validators.required, Validators.email, Validators.minLength(5)]]
20   });
```

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 (it 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 setting
 - `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 by 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