HOA SEN UNIVERSITY

www.hoasen.edu.vn

Angular (cont)

Phạm Thị Kim Ngôn

Ngon.phamthikim@hoasen.edu.vn

Contents

- Introduction
- Reactive forms
- Template-driven forms

Forms

- **Reactive forms**

  Reactive forms are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

- **Template-driven forms**

  Template-driven forms are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, use template-driven forms.

Key differences

The table below summarizes the key differences between reactive and template-driven forms.

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

Form validation

Forms	Details
Reactive forms	Define custom validators as functions that receive a control to validate
Template-driven forms	Tied to template directives, and must provide custom validator directives that wrap validation functions

REACTIVE FORM

Reactive Forms

- Reactive forms provide **a model-driven approach** to handling form inputs whose values change over time
- Reactive forms use an explicit and immutable approach to managing the state of a form at a given point in time.
- **Each change to the form state returns a new state**, which maintains the integrity of the model between changes.
- Reactive forms are built around observable streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

HOA SEN UNIVERSITY

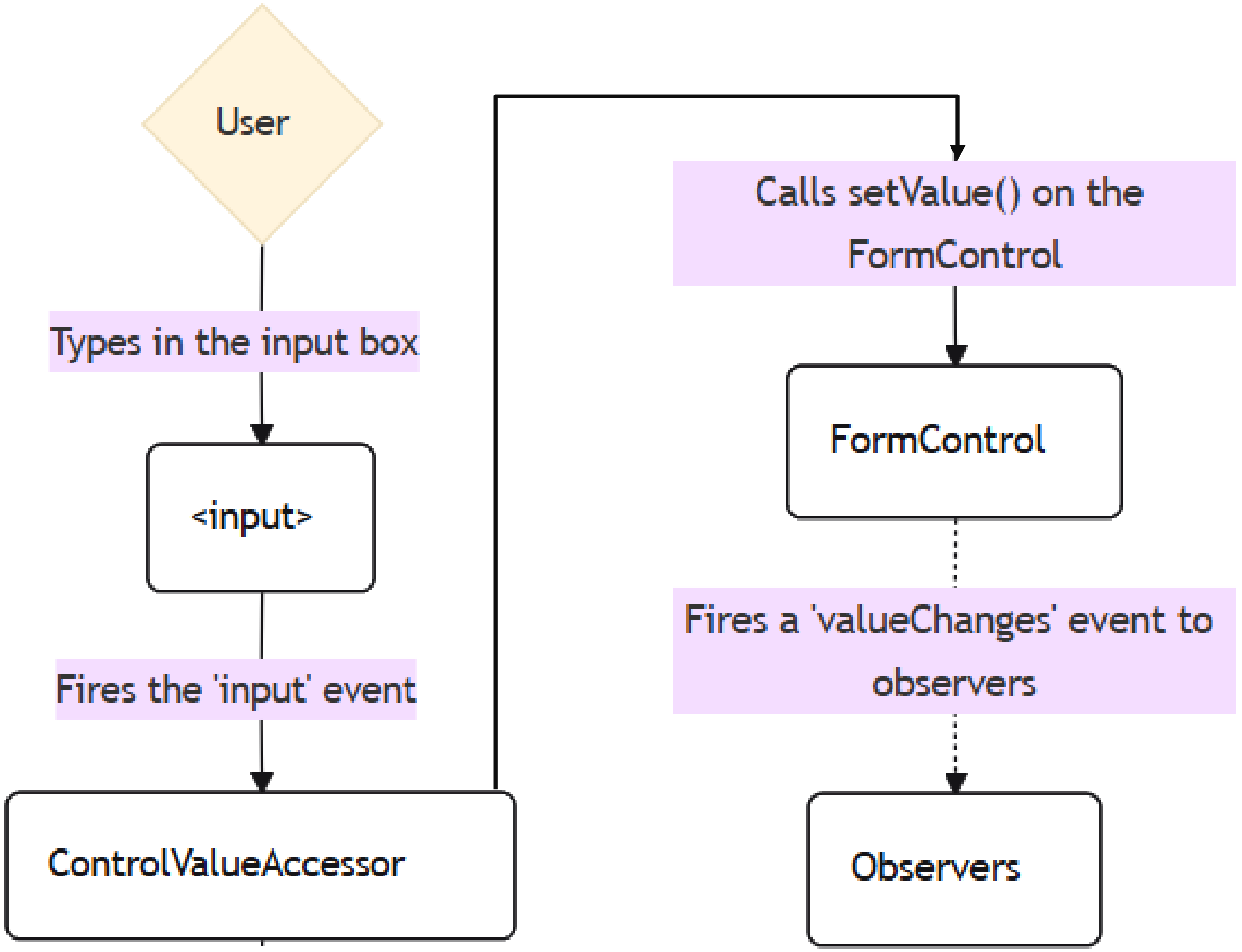
www.hoasen.edu.vn

Introduce Reactive Forms (1)

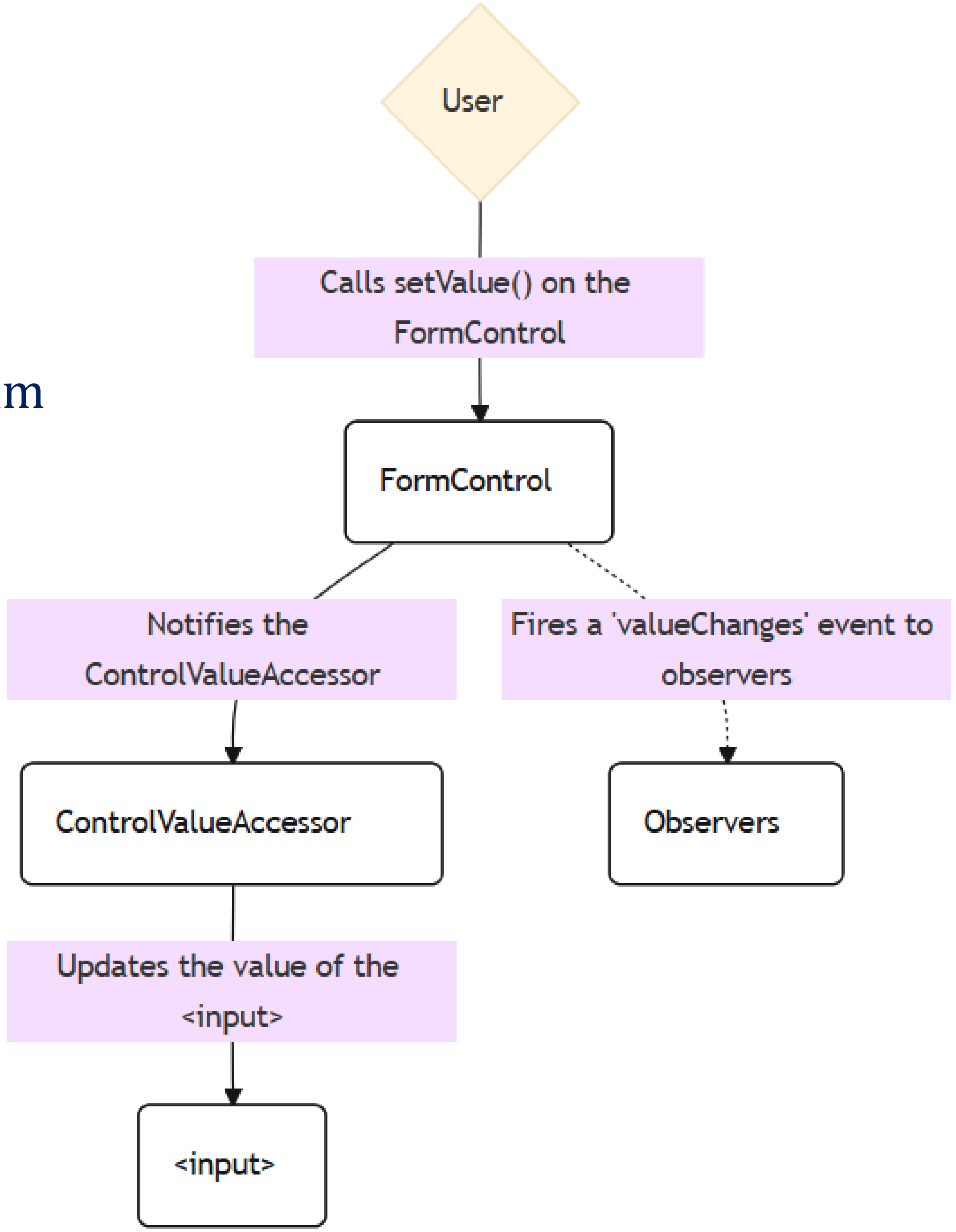
```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';
@Component({
    selector: 'app-reactive-favorite-color',
    imports: [ReactiveFormsModule],

    template: `
        Favorite Color: <input type="text"
        [formControl]="favoriteColorControl">
        `
})
export class FavoriteColorComponent {
        favoriteColorControl = new FormControl('');
}
```

Data flow in reactive forms(2)

- The view-to-model diagram



The model-to-view diagram


User
Calls setValue() on the FormControl
FormControl
Notifies the ControlValueAccessor
Fires a 'valueChanges' event to observers
ControlValueAccessor
Observers
Updates the value of the <input>
<input>

CREATE REACTIVE FORM

Generating and importing a new form control

ng generate component NameEditor

- The FormControl class is the basic building block when using reactive forms.

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

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

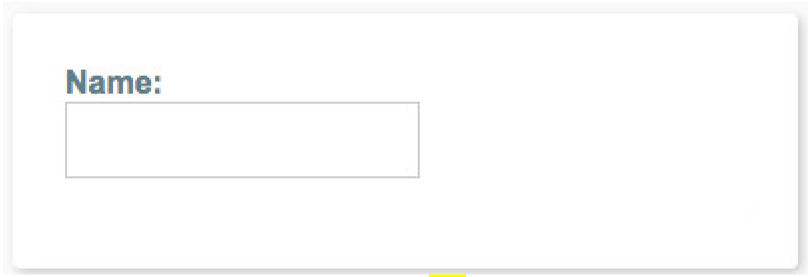
Registering the control in the template

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

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

```
src/app/app.component.html
```

```html
<app-name-editor></app-name-editor>
```



Managing control values

- Add t code block to “name-editor.component.html”

```
<p>
Value: {{ name.value }}
</p>
```

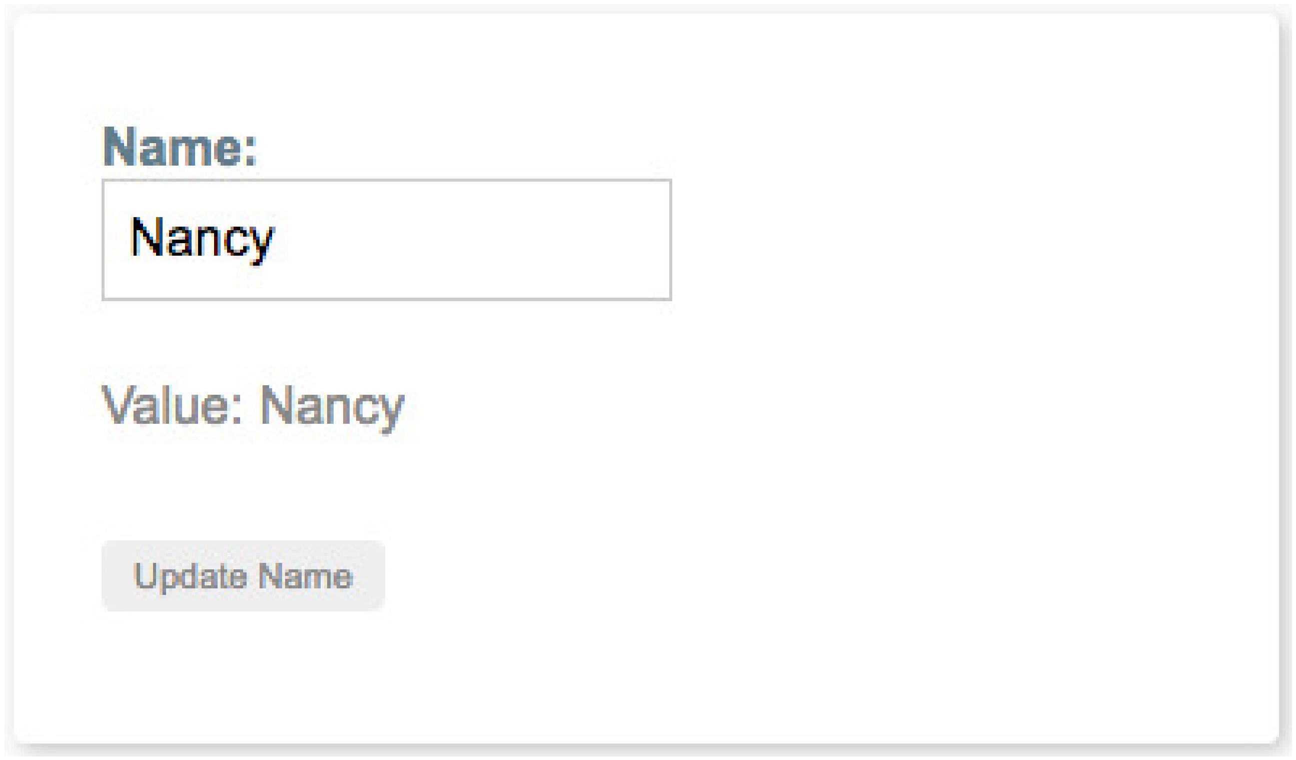
- Adds a method to the component class name-editor.component.ts”

```
updateName() {
        this.name.setValue('Nancy');
}
```

- Update the template with a button to simulate a name update.

```
<p>
<button (click)="updateName()">Update Name</button>
</p>
```

Managing control values


Name:
Nancy

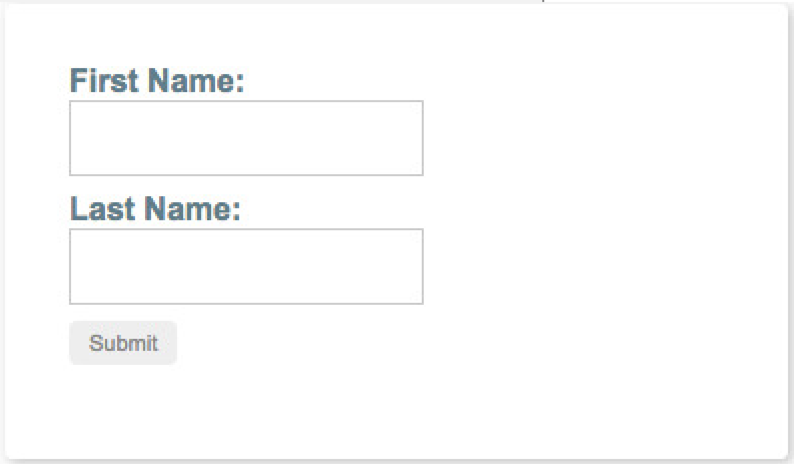
Value: Nancy

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Grouping form controls

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



Associating the FormGroup model and view

```
<form [formGroup]="profileForm">
<label>
First Name:
<input type="text" formControlName="firstName">
</label>

<label>
Last Name:
<input type="text" formControlName="lastName">
</label>
</form>
```

Saving form data

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
```

```
<button type="submit"
[disabled]="!profileForm.valid">Submit</button>
```

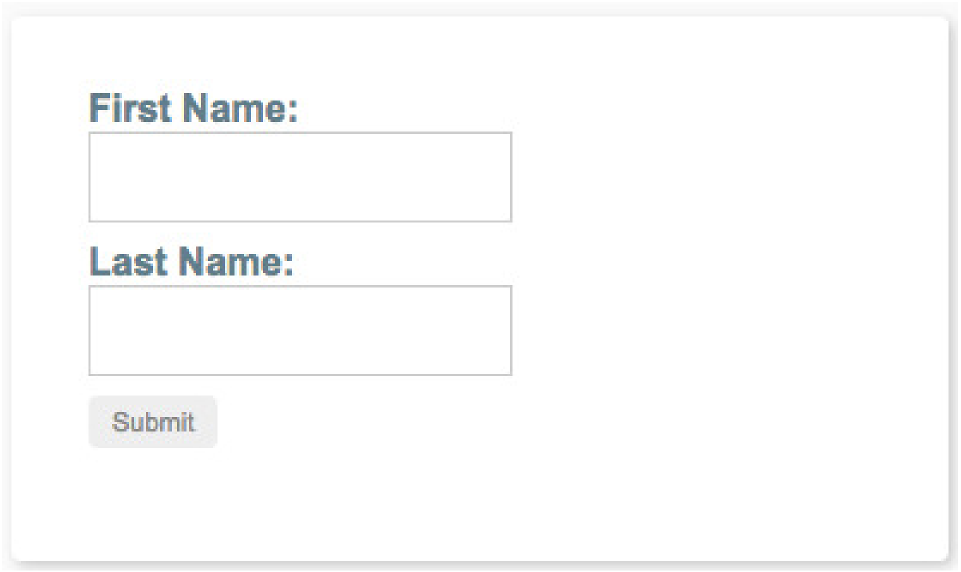
src/app/profile-editor/profile-editor.component.ts

```
onSubmit() {
    // TODO: Use EventEmitter with form value
    console.warn(this.profileForm.value);
}
```

Displaying the component

src/app/app.component.html (profile editor)

```
<app-profile-editor></app-profile-editor>
```



Partial model updates

- There are two ways to update the model value:
  - Use the **setValue()** method to **set a new value** for an **individual control**. The setValue() method strictly adheres to the structure of the form group and replaces the entire value for the control.
  - Use the **patchValue()** method to **replace any properties** defined in the object that have changed in the form model.

```
updateProfile() {
    this.profileForm.patchValue({
        firstName: 'Nancy',
        address: {street: '123 Drew Street'}
        });
}
```

```
<p><button (click)="updateProfile()">Update Profile</button></p>
```

Generating form controls with FormBuilder

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';
@Component({
    selector: 'app-profile-editor',
    imports: [ReactiveFormsModule],
    templateUrl: './profile-editor.component.html',
    styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
     private fb = inject(FormBuilder);
    profileForm = this.fb.group({
    firstName: [''], 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.formBuilder.control(''));
}
```

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

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

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Simple form validation

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

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

```
<input type="text" formControlName="firstName" required>
```

```
<p>
Form Status: {{ profileForm.status }}
</p>
```

Form Status (valid or invalid)

Unified control state change events

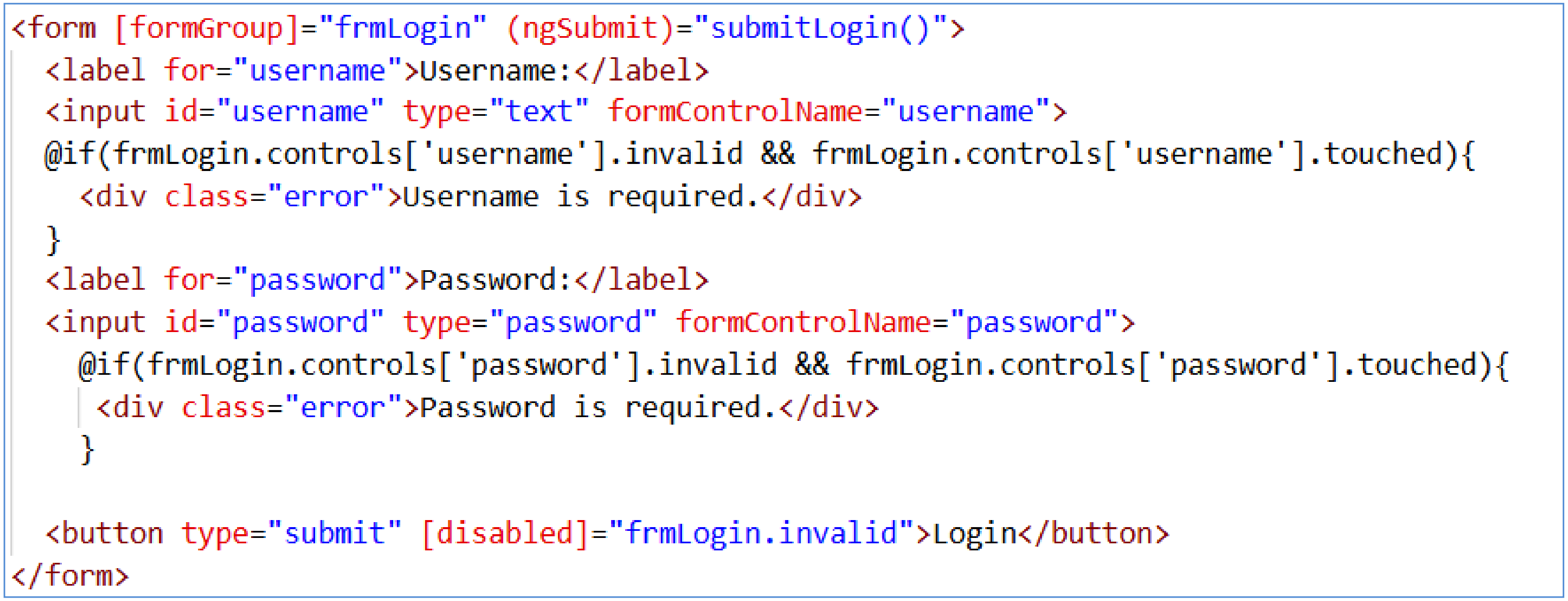
- All form controls expose a single unified stream of control state change events through the events observable on AbstractControl (FormControl, FormGroup, FormArray, and FormRecord).
- This unified stream lets you react to value, status, pristine, touched and reset state changes and also for form-level actions such as submit , allowing you to handle all updates with a one subscription instead of wiring multiple observables.

Event types

- Each item emitted by events is an instance of a specific event class:
  - ValueChangeEvent — when the control's value changes.
  - StatusChangeEvent — when the control's validation status updates to one of the FormControlStatus values (VALID, INVALID, PENDING, or DISABLED).
  - PristineChangeEvent — when the control's pristine/dirty state changes.
  - TouchedChangeEvent — when the control's touched/untouched state changes.
  - FormResetEvent — when a control or form is reset, either via the reset() API or a native action.
  - FormSubmittedEvent — when the form is submitted.

```
import { Component } from '@angular/core';
import { FormControl, ValueChangeEvent, StatusChangeEvent,
  PristineChangeEvent,TouchedChangeEvent, FormResetEvent,
  FormSubmittedEvent, ReactiveFormsModule, FormGroup} from
'@angular/forms';
@Component({
  selector: 'app-login',
  imports: [ReactiveFormsModule],
  templateUrl: './login.html',
  styleUrl: './login.css'
})
```

```
export class Login {
  form = new FormGroup({
    username: new FormControl(''),
    password: new FormControl('')});
  constructor() {
    this.form.events.subscribe((e) => {
      if (e instanceof ValueChangeEvent) {
        console.log('Value changed to: ', e.value);
      }
      if (e instanceof StatusChangeEvent) {
        console.log('Status changed to: ', e.status);
      }
      if (e instanceof PristineChangeEvent) {
        console.log('Pristine status changed to: ', e.pristine);
      }
      if (e instanceof TouchedChangeEvent) {
        console.log('Touched status changed to: ', e.touched);
      }
      if (e instanceof FormResetEvent) {
        console.log('Form was reset');
      }
      if (e instanceof FormSubmittedEvent) {
        console.log('Form was submitted');
      }
    });
  }
```



```
<form [formGroup]="frmLogin" (ngSubmit)="submitLogin()">
  <label for="username">Username:</label>
  <input id="username" type="text" formControlName="username">
  @if(frmLogin.controls['username'].invalid && frmLogin.controls['username'].touched){
    <div class="error">Username is required.</div>
  }
  <label for="password">Password:</label>
  <input id="password" type="password" formControlName="password">
    @if(frmLogin.controls['password'].invalid && frmLogin.controls['password'].touched){
     <div class="error">Password is required.</div>
    }

  <button type="submit" [disabled]="frmLogin.invalid">Login</button>
</form>
```

TEMPLATE-DRIVEN FORM

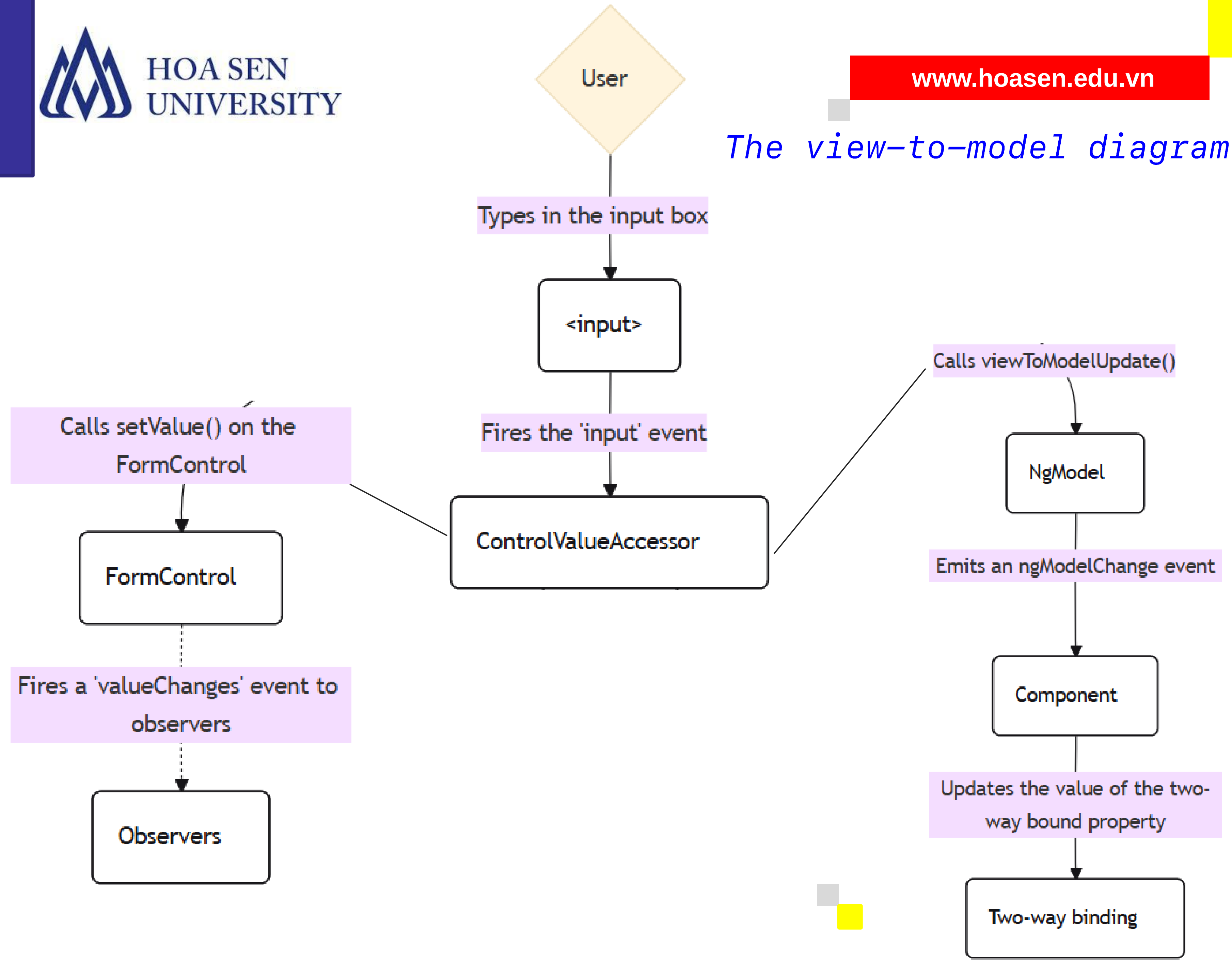
Introduce Template-driven Forms (1)

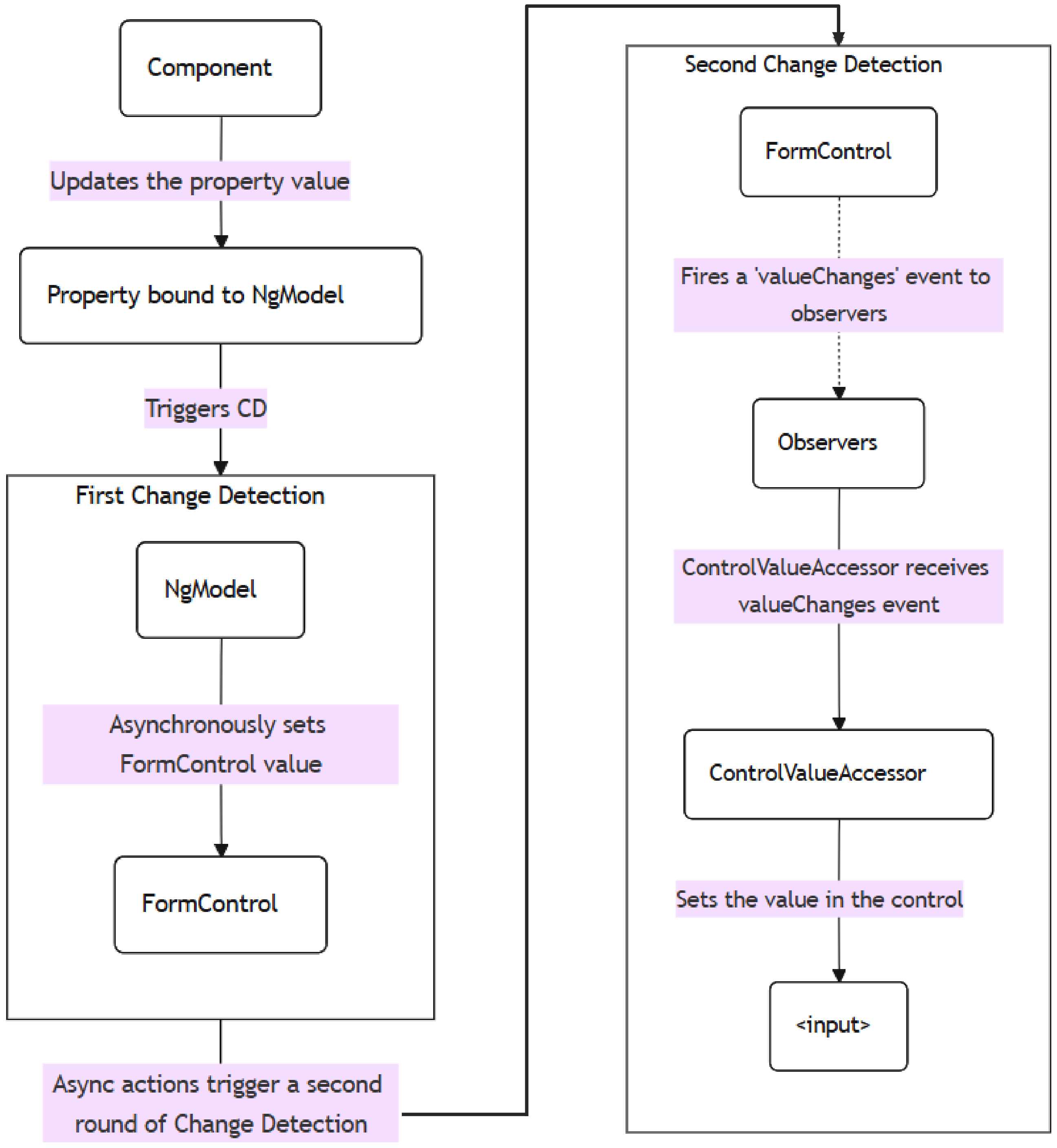
```
import { Component } from '@angular/core';

@Component({
    selector: 'app-template-favorite-color',
    template: `
        Favorite Color: <input type="text"
        [(ngModel)]="favoriteColor">
        `
})
export class FavoriteColorComponent {
        favoriteColor = '';
}
```

HOA SEN UNIVERSITY

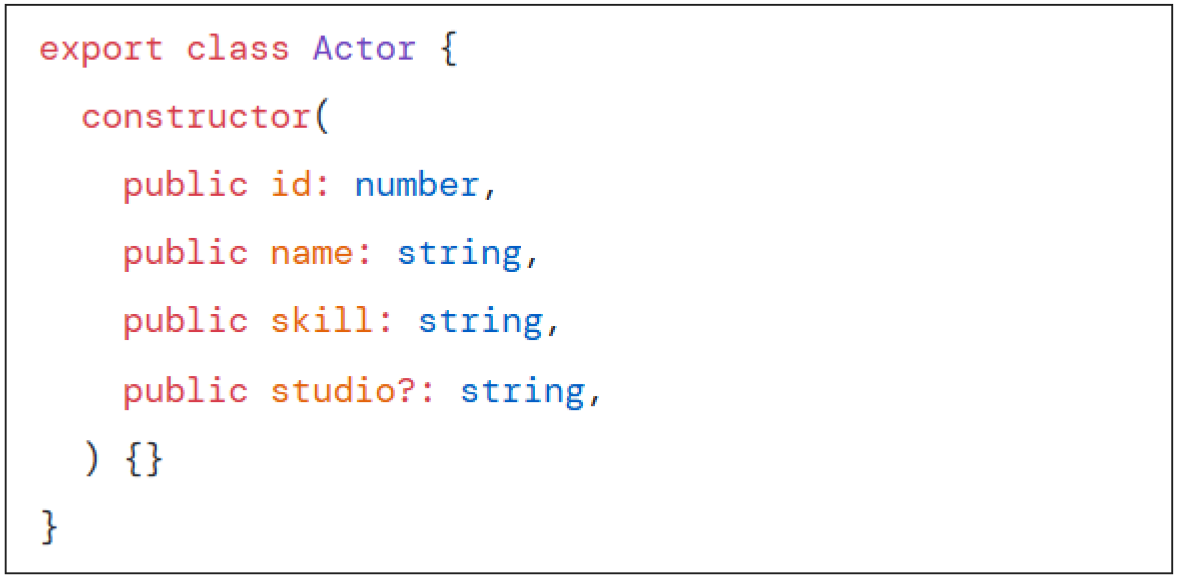
www.hoasen.edu.vn

The view-to-model diagram


User
Types in the input box
<input>
Fires the 'input' event
ControlValueAccessor
Calls setValue() on the FormControl
FormControl
Fires a 'valueChanges' event to observers
Observers
Calls viewToModelUpdate()
NgModel
Emits an ngModelChange event
Component
Updates the value of the two-way bound property
Two-way binding

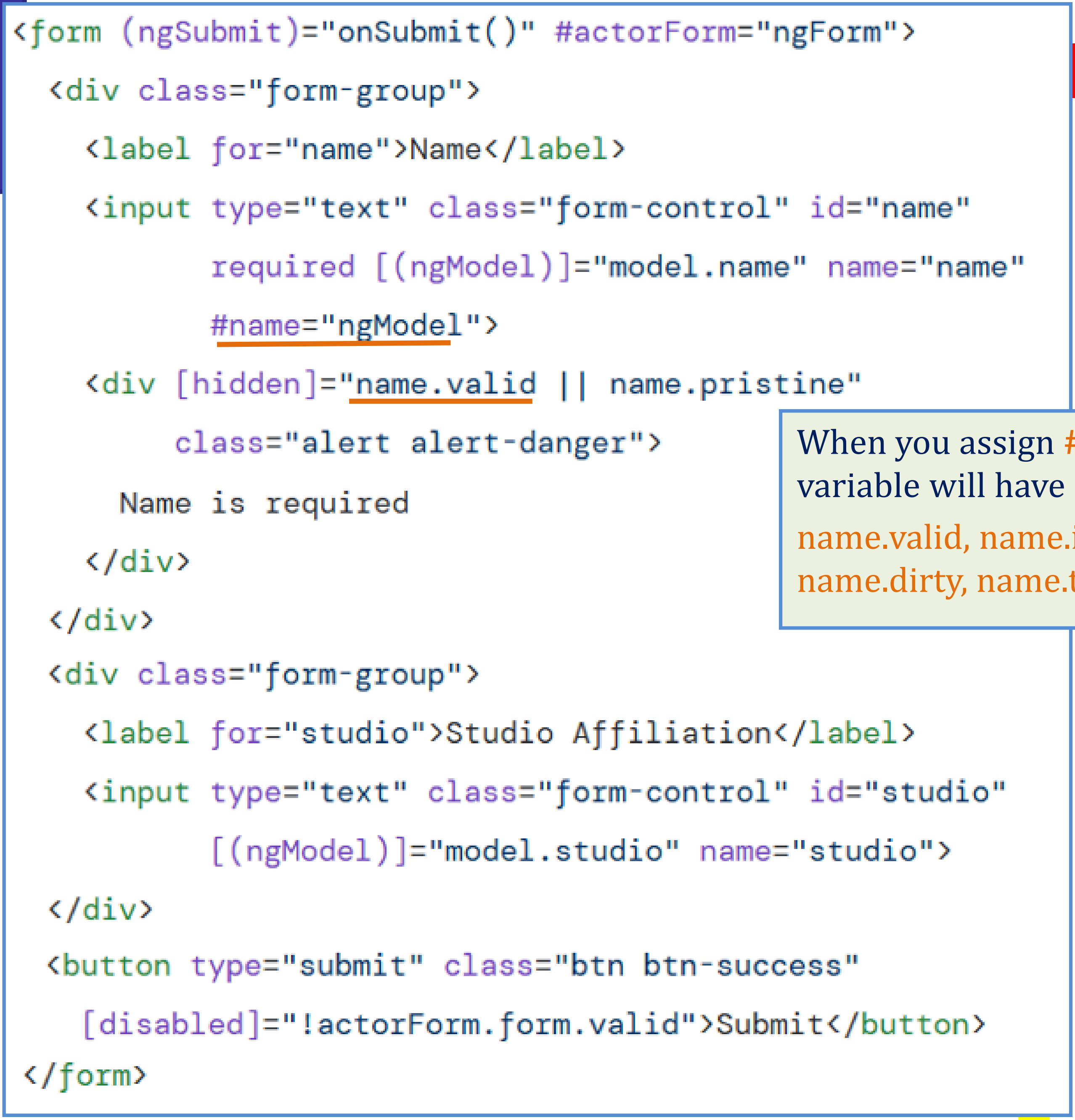

Component
Updates the property value
Property bound to NgModel
Triggers CD
First Change Detection
NgModel
Asynchronously sets FormControl value
FormControl
Async actions trigger a second round of Change Detection
Second Change Detection
FormControl
Fires a 'valueChanges' event to observers
Observers
ControlValueAccessor receives valueChanges event
ControlValueAccessor
Sets the value in the control
<input>

The model-to-view diagram



```
export class Actor {
  constructor(
    public id: number,
    public name: string,
    public skill: string,
    public studio?: string,
  ) {}
}
```

```
import {Actor} from '../actor';

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

import {JsonPipe} from '@angular/common';


@Component({

  selector: 'app-actor-form',

  templateUrl: './actor-form.component.html',

  imports: [FormsModule, JsonPipe],

})

export class ActorFormComponent {

  skills = ['Method Acting', 'Singing', 'Dancing', 'Swordfighting'];


  model = new Actor(18, 'Tom Cruise', this.skills[3], 'CW Productions');


  submitted = false;


  onSubmit() {

    this.submitted = true;

  }


...

}
```



```
<form (ngSubmit)="onSubmit()" #actorForm="ngForm">
  <div class="form-group">
    <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>
  </div>
  <div class="form-group">
    <label for="studio">Studio Affiliation</label>
    <input type="text" class="form-control" id="studio"
           [(ngModel)]="model.studio" name="studio">
  </div>
  <button type="submit" class="btn btn-success"
    [disabled]="!actorForm.form.valid">Submit</button>
</form>
```

When you assign **#name="ngModel"**, name variable will have properties such as:
name.valid, name.invalid, name.pristine, name.dirty, name.touched, name.untouched

Track control states

States	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

```
<input class="form-control ng-untouched ng-pristine ng-valid">;
```

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

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

Demo

HOA SEN UNIVERSITY

www.hoasen.edu.vn

References