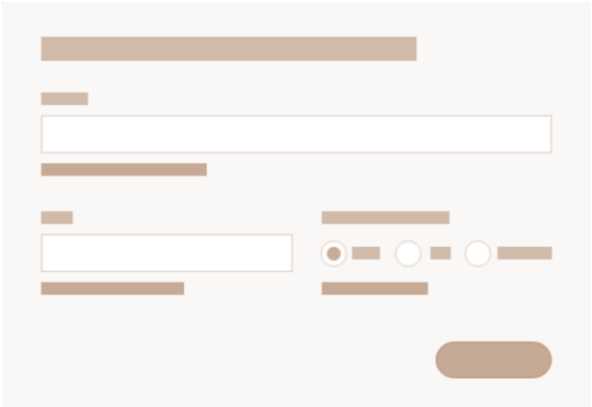
8

Working with forms

FormControl and FormGroup, Template driven, Model driven forms, Data validation

What's in a Form ?

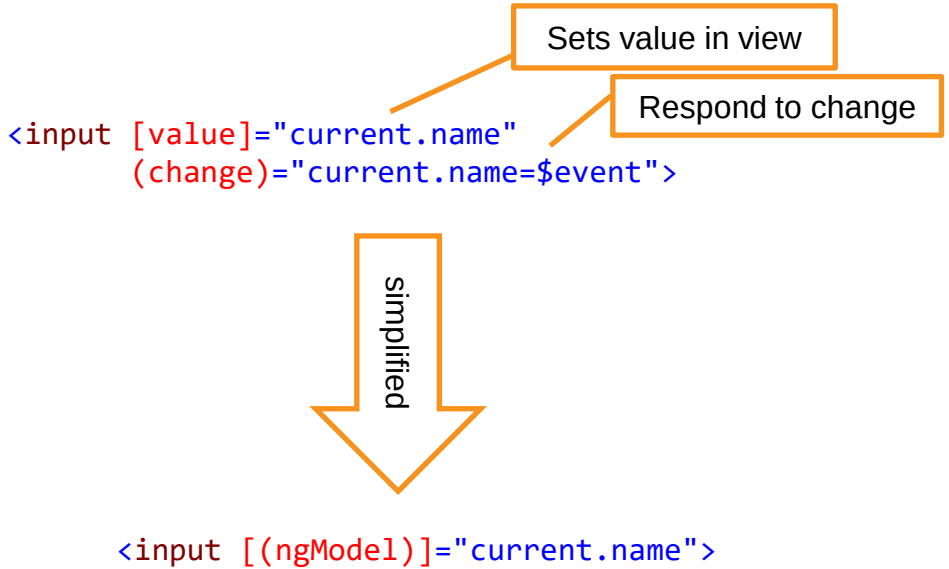
- HTML elements
  - input, select, checkbox, radio, slider, ...
- Values (Data source)
- Change tracking
  - Get notified when user change inputs
- Data validation
  - Control the user inputs
- Visual feedback
  - Display errors
  - Disable buttons



ngModel (Reminder)

Two-way binding

- A combination between one-way binding with event handling
- Typically used with **ngModel**
- Format : **[(ngModel])=“property”**



FormControl

- **FormControl** is a interface that represents a single input field
  - It keeps track of the state of the input
  - It controls the input data validation

- It has the following properties
  - value
  - dirty/pristine: indicates if the value has been changed
  - touched/untouched: indicates if the input has been visited
  - valid: true/false
  - errors: key/value pairs
  - validator: function
  - ...

FormGroup

- **FormGroup** aggregate controls
  - Wraps controls into an interface
  - Exposes aggregated properties from the FormControls
  - The entire form is a FormGroup

- Example

```
let personInfo = new FormGroup({
     firstName: new FormControl("Johnny"),
     lastName: new FormControl("Bravo")
   });
```

  - personInfo.value → `{firsName, lastName}`
  - personInfo.dirty → true if **one** the controls is dirty
  - personInfo.valid → true if **all** the controls are valid

Angular forms

In Angular you can build forms in two different ways :

- **Template-Driven**
  - FormsModule
  - Based on template (HTML)
  - ngForm, ngModel
  - Validation directives
  - Controls are refered in template
  - CSS-based feedback
  - *ngIf-based feedback
- **Model-Driven**
  - ReactiveFormsModule
  - Based on model (TS)
  - FormGroup, FormControl
  - Validation functions
  - Controls are refered in code
  - CSS-based feedback
  - *ngIf-based feedback

Angular forms

In Angular you can build forms in two different ways :

- **Template-Driven**

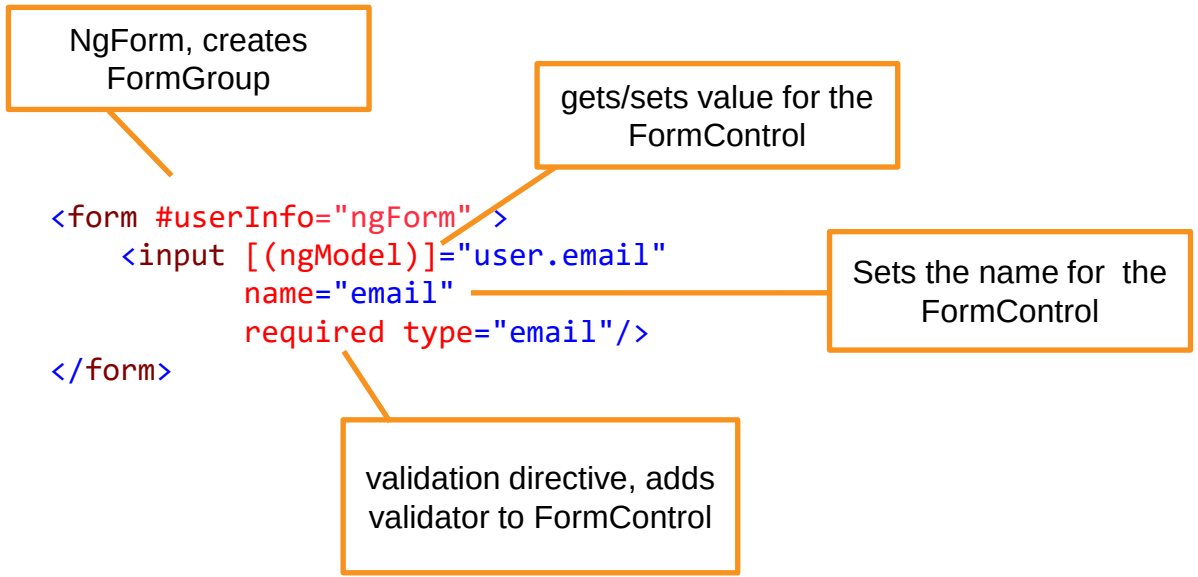
```
<form #tshirtForm="ngForm" novalidate>

    <input [(ngModel)]="tshirt.email"
           name="email"
           #email="ngModel"
           required
           type="email">
```

- **Model-Driven**

```
this.mForm = fb.group({
  'size': [TShirtSize.Medium],
  'print': ['Wazup?',
            Validators.required],
  'email': ['e@ma.il']
});
```

Template driven

- With Template-Driven forms we use **directives** to create the form groups, controls and validators



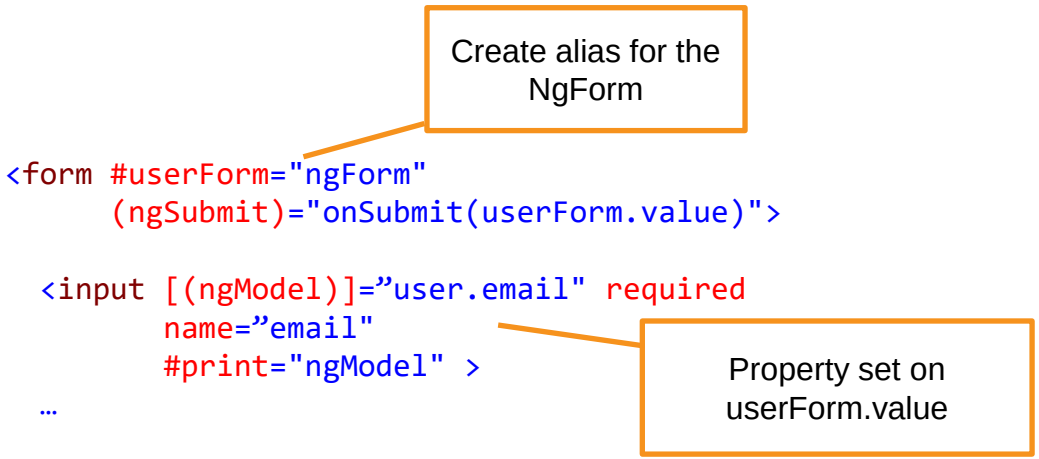
Template driven

- FormsControls and FormsGroup are pimarily used **in the view** for validation
  - Use local template variables with “ngForm” and “ngModel”
  - This makes NgForm and NgModel responsible for poviding the reference

```
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm.value)" >

    <input [(ngModel)]=”user.email"
           name="email" #email="ngModel"
           required type="email">
    <p *ngIf="!email.valid && email.dirty">
      Email is required
    </p>

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

</form>
```

NgForm

- Uses "**form**" as selector
- Groups FormControls into a FormGroup called ngForm
- (ngSubmit) event for submission



NgModel

When used with ngModel

- Creates a form control for the form element
- name is mendatory
- Two-way binding is not mandatory

```
<form #userForm="ngForm"
      (ngSubmit)="onSubmit(userForm.value)">

  <input name="print" ngModel>
  <input name="email" ngModel>
  …
```

```
userForm.value = {
  print: 'value from first input',
  email: 'value from second input'
};
```

FormsModule

- All necessary directives for Template-Driven forms can be found in **FormsModule**
  - Import this module into the current module to use them

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

@NgModule({
  imports: [FormsModule]
})
export class AppModule { }
```

Model driven (Reactive forms)

- With Model-driven forms you create form controls, form groups and validator in the **class component**

```
export class ModelFormComponent {

  userForm: FormGroup;

  constructor() {
    this.userForm = new FormGroup({
      'username': new FormControl('Wazup?', Validators.required),
      'email': new FormControl('e@ma.il')
    });
  }
}
```

Model driven

- We use formGroup and formControlName to bind to the defined form groups and form controls

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

  <input formControlName="email"
         type="email">

</form>
```

Model driven

- Use **FormBuilder** to create form groups in more concise way

```
userForm: FormGroup;

constructor(private fb: FormBuilder) {
   this.userForm = fb.group({
     'username': ['Wazup?', Validators.required],
     'email': ['e@ma.il']
   });

   this.email = this.userForm.controls['print'];
}
```

Model driven

- Validation is primarily determined in the component
  - You can use valueChanges to monitor and react to changes
  - Use Validators.compose (or an array) to set multiple validators

```
constructor(private fb: FormBuilder) {
  this.userForm = fb.group({
    'print': ['Wazup?', Validators.required],
    'email': ['e@ma.il', [Validators.required, this.emailValidator]]
  });

  this.print = this.userForm.controls['print'];

  this.print.valueChanges.subscribe((value: string) => {
    console.log('checking value: ', value);
  });
}
```

Model driven

- Because FormControls are easy to access from code,
  - Model-driven prefered when dealing with complex validation
- Syntax is slightly less convenient in the templte

```
<form [formGroup]="mForm" novalidate>

  <input formControlName="email" type="email" >

  <p *ngIf="!mForm.controls.email.valid &&
            mForm.controls.email.dirty">
    Email is required
  </p>

  …
<form>
```

ReactiveFormsModule

- All necessary directives for Model-Driven forms can be found in **ReactiveFormsModule**
  - Import the module to build model-driven forms

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

@NgModule({
  imports: [ReactiveFormsModule]
})
export class AppModule { }
```

Form validation

Template-driven forms

- A validator is the process of checking if the information provided by the user is “correct”
  - correct = respect the constraints of the given validator

- In template-driven forms validators are set up in the template using native HTML validation attributes
  - `required` : Input need to be filled
  - `minlength, maxlength`: minimum , maximum length of the textual input
  - `min, max`: minimum, maximum value of the numerical input
  - `pattern`: specifies a regular expression
  - `email`: a valid email (respect the email pattern)

Form validation

Template-driven forms

```
<input [(ngModel)]="email" required email>
```

Required field

A valid email format

```
<input [(ngModel)]="password" required pattern="(?=.*[a-z])(?=.*[A-
Z])(?=.*[0-9])(?=.*[$@$!%*?&])[A-Za-z\d$@$!%*?&].{8,}" >
```

- At least 8 characters in length
- Lowercase letters
- Uppercase letters
- Numbers
- Special characters

Form validation

Model-driven forms

- In template-driven forms validators are specified in the model using built-in Angular validators functions provided by `Validators` class
  - `required` : Input need to be filled
  - `requiredTrue` : Input need to be filled with true value (generally used with checkbox)
  - `minlength, maxlength`: minimum , maximum length of the textual input
  - `min, max`: minimum, maximum value of the numerical input
  - `pattern`: specifies a regular expression
  - `email`: a valid email (respect the email pattern)
  - `compose`: a special validator that can take multiple validators

Form validation

Model-driven forms

```
this.form = this.formBuilder.group({
            email: ['', Validators.compose(Validators.required, Validators.email)],
            password: ['', [Validators.required, Validators.pattern('<regex>')]
})
```

Required field

Valid email format

Use an array ([ ]) or compose() to set multiple validators

Must match the given pattern (regex expression)

Feedback

- Use properties of the control to provide feedback
  - Using the form control

```
<p *ngIf="!email.valid && email.dirty">Invalid</p>

<p *ngIf="email.errors?.required && email.dirty">Required</p>
```

  - Using the form group

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

Feedback

- Angular **automatically** adds CSS classes for validation
  - `ng-valid`: is set when the control is valid
  - `ng-invalid`: is set when the control is invalid
  - `ng-pristine`: is set when the value has not been changed yet
  - `ng-dirty`: is set when the value has been changed
  - `ng-touched`: is set when the user has visited the input
  - `ng-untouched`: is set when the user have not interacted with the input yet
  - `ng-submitted`: is set when onSubmit is called

Feedback

- Set CSS rules for the previous classes

Example

```
<input ...  required >
```

```
input.ng-invalid.ng-dirty {
  outline: 2px solid red;
}
```

```
input.ng-valid.ng-dirty {
  outline: 2px solid green;
}
```

LAB 8

Login page