Reactive Forms

Agenda

- Recap Reactive Forms Revisited
  - FormGroup, Form Builder, FormControl, Forms Array
- Form Validation
- Implementing Custom Validators
- Cascading Form Controls
- Dynamic Form Generation & Validation (Manual & ngx-formly)
- Declarative Binding in Reactive Forms using RxJS

Reactive Forms

Reactive Forms

Reactive forms provide a model-driven approach to handling form inputs

Use an explicit and immutable approach to managing the state of a form at a given point in time

Are built around observable streams, where form inputs and values are provided as streams

Depend on ReactiveFormsComponent import in Module

Uses FormGroup, FormControl, FormArray elements

FormGroups are used to Group FormControls

FormControl

Represents a Control in a Form

- Provide information about state: touched, dirty, valid
- Can be subscribed to
- Support sync & async validation

Can be bound using the following syntax:

- [formControl] = "email"
- formControlName="email"

Can be used WITH and WITHOUT a FormGroup

„Replaces“ ngModel in Reactive Forms

```
name = new FormControl('Giro', [Validators.required]);
postal = new FormControl('3544');
```

```
<mat-form-field fxFlex="1 1 auto">
  <input matInput type="text" placeholder="Name" [formControl]="name" />
  <mat-error *ngIf="!name.valid">
    Name is required
  </mat-error>
</mat-form-field>
```

FormGroup

Defines a form with a fixed set of controls that you can manage together

Can be created manually or using FormBuilder

Can be nested with other form groups to create more complex forms

Provide information about state: touched, dirty, valid

```
initForm() {
  this.personForm = new FormGroup({
    name: new FormControl(this.person.name),
```

```
<form
  [formGroup]="personForm"
  (ngSubmit)="savePerson(personForm)"
  novalidate
  fxLayout="column"
>
  <mat-form-field>
    <input matInput type="text" placeholder="Name" formControlName="name" />
  </mat-form-field>
```

FormBuilder

Creating form control instances manually can become repetitive when dealing with multiple forms

Helper Class that allows us to explicitly declare forms in our components and reduce boilerplate

FormBuilder must be injected using DI

Forms can contain multible FormGroups

```
this.personForm = this.fb.group({
  name: [this.person.name, Validators.required],
  age: [this.person.age],
  gender: [this.person.gender],
  email: [this.person.email],
  wealth: [this.person.wealth],
});
```

FormArray

Defines a dynamic form, where you can add and remove controls at run time

In contrast do FormGroup, you don't need to define a key for each control by name, so this is a great option if you don't know the number of child values in advance

Bound using:

- formArrayName="skillsGrp"

FormControl vs FormControlName

Both Sync a FormControl in an existing FormGroup to a form control element by name.

- FormControl
  - Uses a dotted notation to access a FormControl in the FormGroup
  - [formControl]="myForm.controls.firstName"
- FormControlName
  - Uses a reference to the UNIQUE FormControl Name in the FormGroup
  - formControlName="firstName"

Setting & Updating Data

There are two ways to update the model value:

- setValue()
  - set a new value the FormGroup and replaces all values the FormGroup
  - Typically used to initialize a Form
- patchValue()
  - Set a value for a specific FormControl
  - Typically used to party update a Form
- reset()
  - Resets the form control, marking it pristine and untouched, and setting the value to null.
  - Can write NEW VALUES!!!

Subscribing to Changes & Accessing Data

Three main methods to listen to changes:

- valueChanges()
  - Raised by the Angular forms whenever the value of the FormControl, FormGroup or FormArray changes
- form.get('NAME').value
- statusChanges()
  - Raised by the Angular forms whenever the Angular calculates the validation status of the FormControl, FormGroup or FormArray
- registerOnChange()
  - Used by formControl to register a callback that is expected to be triggered every time the native form control is updated

Validation

HTML Validation

HTML5 provides input types that expect data in a specific format,

You can also apply your own custom rules to many input fields by using a regular expression

```
Age:
<input type="number" size="6" name="age" min="18" max="99" value="21" />
Website: <input type="url" name="website" required placeholder="Insert URL" />

//URL
<input type="url" pattern="https?://.+" />
//IPv4 Address
<input type="text" pattern="\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" />
//Price
<input type="text" pattern="\d+(\.\d{2})?" />
//Long /Lat
<input type="text" pattern="-?\d{1,3}\.\d+" />
```

```
input {
    border: solid 1px;
}
input:invalid {
    border-color: #f00;
}
input:valid {
    border-color: #0f0;
}
```

Angular Validation

Typically you want Angular to take over validation instead on HTML validation

HTML Validation can be disabled using a novalidate (HTML 5) attribute

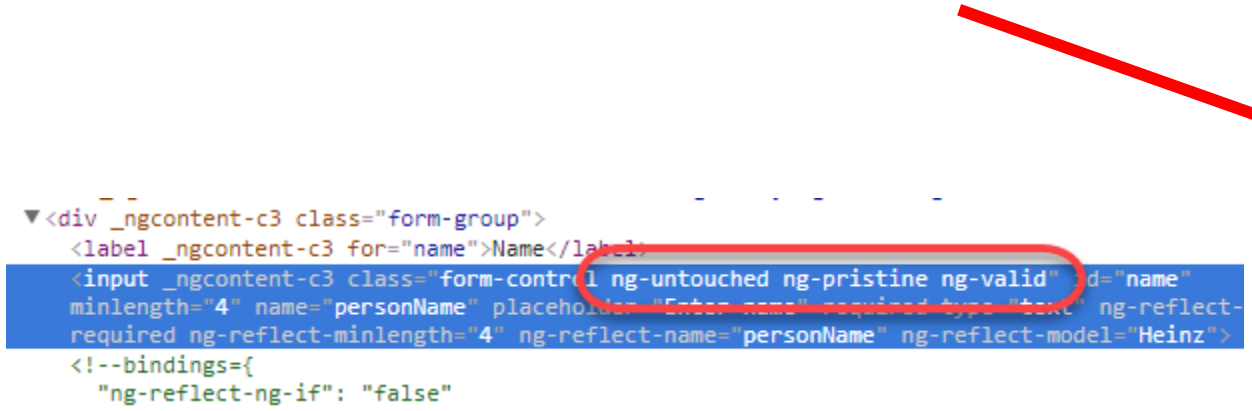
Angular provides several Validators

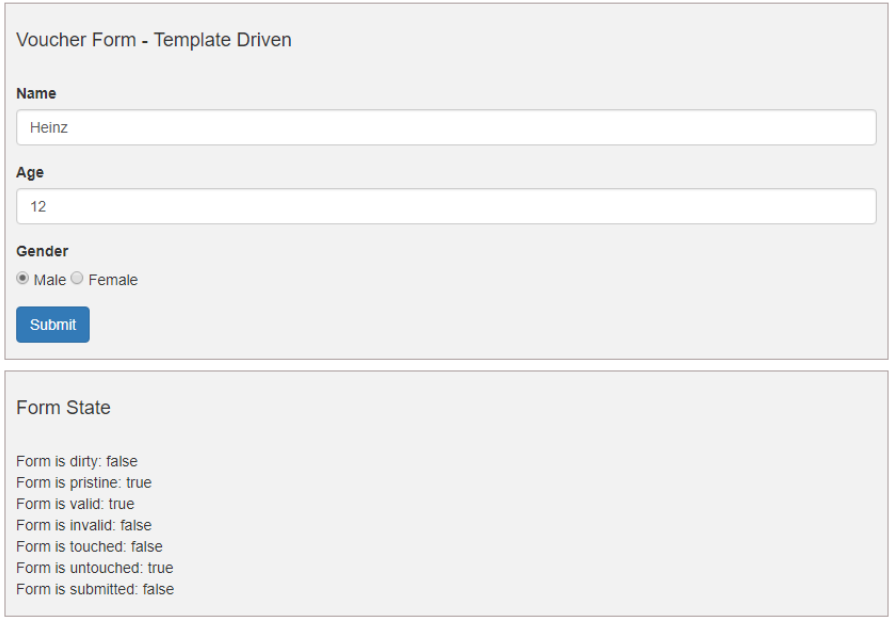
- EmailValidator
- RequiredValidator, CheckboxRequiredValidator
- AsyncValidator
- Min- / Max-LenghtValidator
- PatternValidator

Form | Control State

Informs about the current State of a Form | Control

```
<mat-card-content>
  Form is dirty: {{ personForm.dirty }}<br />
  Form is pristine ('unberührt'): {{ personForm.pristine }}<br />
  Form is valid: {{ personForm.valid }}<br />
  Form is invalid: {{ personForm.invalid }}<br />
  Form is touched: {{ personForm.touched }}<br />
  Form is untouched: {{ personForm.untouched }}<br />
</mat-card-content>
```



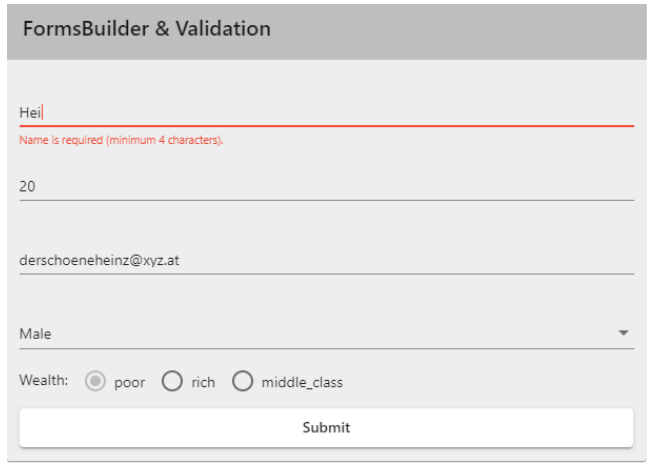


Validating Reactive Forms

Reative Forms support 2 types of Validation

- synchronous
- asynchronous

on FormControls & FormGroups




```
<mat-form-field>
  <input matInput type="text" placeholder="Name" formControlName="name" />
  <mat-error
    *ngIf="
      personForm.get('name').errors != undefined &&
      personForm.controls['name'].errors != undefined &&
      personForm.controls['name'].errors['nameError']
    "
  >
    The Name Hugo is not allowed
  </mat-error>
  <mat-error *ngIf="violatesMinLenght()">
    Name is required (minimum 4 characters).
  </mat-error>
</mat-form-field>
```

```
private initForm() {
  this.personForm = this.fb.group({
    name: [
      this.person.name,
      [Validators.required, Validators.minLength(4), this.validateName],
    ],
```

FormControl & Validation

The FormControl Constructor accepts 1 mandatory and 3 optional params

- Field to bind to
- Sync Validators Function or Array
- Asyn Validatiors Function orArray
- Event to Validate ... click, blur, ...

Example:

- city = new FormControl('Idolsberg', [Validators.maxLength(15)], [this.cityInServiceAreaValidator]);

Trigger Validation using Code

When a Form is submitte the ngSubmit event is triggered

Validation can also be triggered using Code – 2 methods available

- FormGroup .updateValueAndValidity()
- FormControl.updateValueAndValidity()

```
validateForm() {
  this.personForm.updateValueAndValidity();
  this.personForm.controls.name.updateValueAndValidity();
}
```

Custom Validators

Custom Validators

The process for creating custom validators is:

- Create a class or service derived from Validator or AsyncValidator
  - For sync Validators you can also use a pure function instead
- Implement the validate() method
- Return null for valid, or an ValidationErrors object for invalid
- Async Validators return an Observable<ValidationErrors> instead
- Add the class to Module Declarations
- Add the class to the component Provider list

Custom synchronous Validators - Function

- Create a pure Function with a FormControl as Input Param that returns a JSON with
  - a Key representing your custom err name and
  - a Boolean
- Do your check and return this JSON or null
- Bind your custom Validator

```
validateName(control: FormControl): { [s: string]: boolean } {
  if (control.value === 'Hugo') {
    return { nameError: true };
  }
  return null;
}
```

```
this.personForm = this.fb.group({
  name: [
    this.person.name,
    [Validators.required, Validators.minLength(4), this.validateName],
  ],
```

Custom asynchronous Validators

Async Validators have to be implemented as Angular Service

The Service can have injections and must implement the validate method inherited from AsyncValidator

Just like the sync Validator it must return a JSON

- a Key representing your custom err name and
- a Boolean

```
@Injectable({ providedIn: 'root' })
export class AsyncMailExistsValidator implements AsyncValidator {
  constructor(private ps: PersonService) {}

  validate(
    ctrl: AbstractControl
  ): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> {
    return this.ps.checkMailExists(ctrl.value).pipe(
      map((exists) => {
        return exists ? { mailexists: true } : null;
      }),
      catchError(() => null)
    );
  }
}
```

Dynamic Forms

Dynamic Forms

Two Choices to create Dynamic Forms

- Manually
  - Requires a Meta Object holding the Form Data
  - Lots of Coding & Boilerplate
  - Everything in your Hand
- Using an Open Source Form Generator
  - Everything ready to use
  - ie ngx-formly

```
createForm() {
  const controls: any = {};
  const cols = this.getColumns();

  cols.forEach((c) => {
    controls[c.Name] = new FormControl(
      this.data.row[c.Name] || '',
      this.getValidators(c)
    );
  });
  this.resForm = this.fb.group(controls);
}

getColumns(): ColDef[] {
  const result = [];
  this.data.source.EditColumns.forEach((ec) => {
    result.push(this.data.source.Columns.find((c) => c.Name == ec));
  });
  return result;
}
```

ngx-formly

Formly is a dynamic (JSON powered) form library for Angular that allows dynamic generation of Forms

Available for:

- Angular Material, Bootstrap
- Ionic
- ...

Base Elements are

- the <formly-form> tag and
- the FormlyFieldConfig

```
form = new FormGroup({});
model = {};
fields: FormlyFieldConfig[] = [
  {
    key: 'input',
    type: 'input',
    templateOptions: {
      label: 'Input',
      placeholder: 'Input placeholder',
      required: true,
    },
  },
  {
    key: 'textarea',
    type: 'textarea',
    templateOptions: {
      label: 'Textarea',
      placeholder: 'Textarea placeholder',
      required: true,
    },
  },
```