Data Entry & Validation

Downloads: http://bit.ly/centric-ng5

Agenda

Datum	Onderwerp
1-2-2017	TypeScript Introduction
15-2-2017	Advanced TypeScript
1-3-2017	Angular Introduction
15-3-2017	Angular Building Blocks
29-3-2017	Components
12-4-2017	RxJS and Ajax
26-4-2017	Data Entry
10-5-2017	Single Page Applications

Downloads: http://bit.ly/centric-ng5

What are we going to cover?

Different ways of creating input forms.

- Template forms
- Reactive forms

Validating input.

- Validators
- Providing feedback

Template based forms

Template based forms are easy to create.

- Uses ngModel for “two” way data binding and ngForm to check on the status

Easy to get started with.

- All form logic is in the template making it hard to test

Template based component

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

@Component({
  selector: 'app-template-movie',
  templateUrl: './template-movie.component.html'
})
export class TemplateMovieComponent {

  @Input() movie;

  save() {
    console.log('Template save', this.movie);
  }
}
```

Template based form

```
<form (ngSubmit)="save()">
  <div>
    <label>Title</label>
    <input type="text" [(ngModel)]="movie.title" name="title">
  </div>
  <div>
    <label>Overview</label>
    <textarea [(ngModel)]="movie.overview"
                cols="100" rows="10" name="overview"></textarea>
  </div>

  <button>Save</button>
</form>
```

Reactive forms

Reactive forms is a completely different way of creating HTML forms.

- Not using the ngModel directive

The main building blocks are **FormControl**, **FormGroup** and **FormArray**.

- Import the **ReactiveFormsModule** from **@angular/forms**
- An HTML template is still used but only for layout purposes
- The **valueChanges** is a reactive stream of all data changes made

Reactive forms and template forms can be used together in the same application.

Reactive component

```
import { Component, OnInit, Input } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({ selector: 'app-reactive-movie', templateUrl: './reactive-movie.component.html'})
export class ReactiveMovieComponent implements OnInit {
  @Input() movie;
  movieForm = new FormGroup({
    title: new FormControl(),
    overview: new FormControl()
  });
  ngOnInit() {
    this.movieForm.setValue({ title: this.movie.title, overview: this.movie.overview });
    this.movieForm.valueChanges.subscribe(movie => console.log('Changing', movie));
  }
  save() {
    console.log('Reactive save', this.movieForm.value);
  }
}
```

Reactive template

```
<form (ngSubmit)="save()" [formGroup]="movieForm">
  <div>
    <label>Title</label>
    <input type="text" formControlName="title">
  </div>
  <div>
    <label>Overview</label>
    <textarea cols="100"
              rows="10" formControlName="overview">
    </textarea>
  </div>

  <button>Save</button>
</form>
```

Using FormBuilder

A FormBuilder is a convenience class to construct reactive forms.

Using a FormBuilder

```
class ReactiveMovieComponent implements OnInit {

  @Input() movie;
   movieForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {}

  ngOnInit() {
    this.movieForm = this.formBuilder.group({
      title:this.movie.title,
      overview: this.movie.overview
    });
  }
}
```

Comparison

Template forms	Reactive forms
Easy to setup	Harder to setup
Data binding and validation mixed with layout	Data binding and validation separate from layout
Changes are asynchronous	Changes are synchronous
Imperative code	Reactive code
Often better for simple forms	Often better for complex data entry forms

Input validation

Data entry usually needs to be validated on the client.

- Provide fast feedback to the user

Beware: All validations must be redone on the server.

- There is no guarantee that the HTTP request originated from the client application

Validation with template driven forms

Template driven forms use the HTML5 validation attributes.

- Make sure to add a **novalidate** attribute to the HTML **form** element

Angular has directives for the HTML5 validations like **required** and **maxLength**.

- Useful for basic validation

Use form variables to store ngForm and ngModel references.

- Used to check on the validity state of the form

Validating the title

```
<form (ngSubmit)="save(movieForm)" #movieForm="ngForm" novalidate>
  <div>
    <label>Title</label>
    <input type="text" [(ngModel)]="movie.title"
           #title="ngModel" name="title" required maxlength="10">

    <div *ngIf="title.errors && title.errors.required">
      Error: The title is required
    </div>
  </div>
  <div>
    <label>Overview</label>
    <textarea [(ngModel)]="movie.overview"
              cols="100" rows="10" name="overview"></textarea>
  </div>

  <button [disabled]="movieForm.pristine || movieForm.invalid">Save</button>
</form>
```

Validation with reactive forms

Validation is added to the FormControl instances.

- Or with the FormBuilder

Use one from the **Validators** class for common validations:

- Required
- Min length
- Max length
- Email
- Regular expression pattern

With the form builder each field is an array property:

- Index 0 is the value
- Index 1 are the synchronous validators
- Index 2 are the asynchronous validators

Using validators

```
ngOnInit() {
  this.movieForm = this.formBuilder.group({
    title: [this.movie.title, [
      Validators.required,
      Validators.maxLength(10)
    ]],
    overview: this.movie.overview
  });
  this.validateTitle();
  this.movieForm.valueChanges
    .subscribe(() => this.validateTitle());
}
```

Validating the title

```
validateTitle() {
  const titleErrors = this.movieForm.controls['title'].errors
  this.titleError = null;
  if (!titleErrors) {
    return;
  }

  const { required, maxlength } = titleErrors;
  if (required) {
    this.titleError = 'Error: The title is required'
  } else if (maxlength) {
    this.titleError = `Error: The title is too long.
        Maximum ${maxlength.requiredLength},
        current ${maxlength.actualLength}`
  }
}
```

Custom validators

Validators are functions that receive the **AbstractControl** as a parameter.

- Wrap with a directive for template forms

Return a **ValidationErrors** object to indicate an error.

- The value will be added to the control **errors** object
- Return **null** to indicate that there is no error

Synchronous validator example

```
const notDots = (control: AbstractControl) => {
  const value: string = control.value
    .replace(/\./g, '');
  if (value) {
    return null;
  }
  else {
    return {
      notDots: true
    };
  }
};
```

Testing the validator

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

describe('noDots', () => {
  it('should fail ..', () => {
    const control = new FormControl('..');
    const result = notDots(control);
    expect(result).toEqual({
      notDots: true
    });
  });
});
```

The validator directive

```
@Directive({
  selector: '[notDots]',
  providers: [{
    provide: NG_VALIDATORS,
    useExisting: NotDotsDirective,
    multi: true }]
})
export class NotDotsDirective implements Validator {
  validate(control: AbstractControl): ValidationErrors | null {
    if (control && control.value) {
      return notDots(control);
    } else {
      return null;
    }
  }
}
```

Asynchronous validators

Asynchronous validators fire after the synchronous ones are complete.

- Can do http request etc.

Add them as the third element with the **FormBuilder**.

Asynchronous validators return a **promise** or **observable**.

- The status is **PENDING** until the promise resolves or the observable completes

The promise eventually resolves with the result of the validation.

- The same kind of value as a synchronous validator

Asynchronous validator example

```
const notDots = (control: AbstractControl) => {
  const value: string = control.value.replace(/\./g, '');

  return new Promise(resolve => {
    setTimeout(() => {
      if (value) {
        resolve(null);
      }
      else {
        resolve({
          notDots: true
        });
      }
    }, 1000);
  })
};
```

Asynchronous validator test

```
describe('noDots', () => {
  it('should fail ..', async(() => {
    const control = new FormControl('..');
    notDots(control)
      .then(result => expect(result)
        .toEqual({
          notDots: true
        }));
  }));
});
```

Conclusion

Template forms are easy to get started with.

- But reactive forms are more powerful when things become complex
- Use the FormBuilder to construct reactive forms

Validation can be done with the inbuild validators.

- Create custom validators where needed
- Validators can be asynchronous where needed

Reactive forms use the validators as is.

- Template forms need an extra decorator