Angular 5 Training Course

Exercise J-forms

- A **FormControl** describes one input field/control.
- Each field have one or more **states**: pristine, dirty, touched, valid.
- We manage multiple FormControls in a **FormGroup**.
- We can use a built-in helper class called **FormBuilder** to simplify coding Angular forms.
- Custom validation can be added with **Validators**.

Setup

- Open the **useful/forms** project.
- Rebuild it

```
npm install
ng serve --open
```

- A form has been defined in **app.component.html**.
- Styles have been provided in **app.component.css**.
- Import statements have been added to **app.component.ts**.
- Angular form modules have been imported in **app.module.ts**.
- Note the **feedback spans** in the form are initially hidden.

FormBuilder

- **FormBuilder** is an Angular helper class for working with FormControls and FormGroups.
- Pass FormBuilder into the **constructor**:

```
constructor(fb: FormBuilder) {}
```

- Define a **FormGroup** variable:

```
film :FormGroup;
```

- Populate this FormGroup with data from an object using FormBuilder.

```
constructor(fb: FormBuilder) {
    let config = { title:"Jaws" , director:"Spielberg" };
    this.film = fb.group( config );
}
```

- To connect this code to the form defined in the template, define a [formGroup] and use it:

```
<form class="films" [formGroup]="film">
```

- Define [formControl] **directives** on each input field.

```
[formControl]="film.controls.title"
[formControl]="film.controls.director"
```

- Add a submit event to the form.

```
<form (ngSubmit)="addFilm(film)" ..>
```

- Add code to listen for this event.

```
addFilm( film ) {
console.log( film.value );
console.log( film.valid );
```

}

- An object containing the form will be logged to the console.

Basic validation

- We will add custom validation to the form using Angular **Validators**.
- Change the configuration object. Each field is now initially empty and is a required field.

```
   let config = {
    title :    [ "" , Validators.required ],
    director : [ "" , Validators.required ]
}
```

- The expression **film.controls.title.valid** returns true if any content has been added to the title field.
- Use **ngIf directives** to conditional reveal feedback spans.

```
<span *ngIf="!film.controls.title.valid">Required</span>
<span *ngIf="!film.controls.director.valid">Required</span>
```

Touched state

- The required feedback displays before the user has entered any value. We only want feedback to appear if the user leaves an edited field empty.
- The **touched** property only becomes true if the user has focused into the field, and then clicked/tabbed out of the field. Extend the expression on the ngIf.

```
<span *ngIf="!film.controls.title.valid &&
film.controls.title.touched">Required</span>
```

Concise names for form controls

- To make this code less **verbose**, define a variable that points to a specific control, and then use it in the view.

```
dc:AbstractControl;
tc:AbstractControl;
```

- In the constructor assign it a value.

```
this.dc = this.film.controls.director;
this.tc = this.film.controls.title;
```

- Apply this shorthand in the template

```
  <input type="text" [formControl]="tc">
```

<span *ngIf="!tc.valid && tc.touched">Required

Custom validation

- We can define functions to perform custom validation
- This checkName function requires the director name to be at least two words.
  checkName(d:FormControl) {

```
let check = d.value.trim().split(" ").length >= 2;

if( !check ) {
    return { shortName : "At least 2 names" }
}
```
}

- We can apply the custom validation using the config object.

```
let config = {
  title :    [ "" , Validators.required ],
  director : [ "" , this.checkName ]
}
```

- Create a function that tests for a specific error and returns its associated error message.

```
getError( f:FormControl ) {

    if( f.hasError("shortName")) {
        return f.errors.shortName;
    }
    return "";

}
```

- Use this function in the error-span:

```
  <span *ngIf="!dc.valid && dc.touched">
  {{ getError(dc) }}</span>
```

Monitor the form using an Observable

- FormGroups are **Observables**: we can subscribe to the **valueChanges** property to monitor the state of a form.
- Log all changes to the form.

```
monitorForm() {
    this.film.valueChanges
    .subscribe( data => console.log( JSON.stringify( data )))
}
```

- Call the function in the constructor

```
this.monitorForm();
```

- We can add a filter to only log valid states of the form.
- This requires we import the Observable filter function

```
import 'rxjs/add/operator/filter';

monitorForm() {
    this.film.valueChanges
    .filter( f => this.film.valid )
    .subscribe( f => console.log( f ))
}
```

Reset form on submit

- If the form is valid and the user clicks submit, we can reset the contents of the form.

```
onSubmit( film ) {
if( film.valid ) {
    let empty = { title: "", director: ""};
    this.film.reset( empty );
}
```

}

- We can also set the active state and visual look of the submit button in the template.

```
[disabled]="!film.valid"
[ngClass]="{'submit' : film.valid }"
```

Debugging tools

- We can display the state of the form on-screen for debugging.

```
<pre>{{film.value | json }}</pre>
```

```
Valid? {{film.valid }}
```

- The **dirty-state** is true if the user has changed the contents of a field. It remains true even if the user edits the field to contain nothing.
- The **pristine-state** is the opposite of the dirty-state.
- The **touched-state** is true if the user has focused on a field, and then focused on another field, by clicking/tabbing out. The touched-state can become true without changing the fields contents.
- The **valid-state** becomes true if it passes the custom validation set.

```
<pre>Title
    dirty-state <span>{{ film.controls.title.dirty }}</span>
    pristine-state <span>{{ film.controls.title.pristine }}
</span>
    touched <span>{{ film.controls.title.touched }}</span>
    valid <span>{{ film.controls.title.valid }}</span>
</pre>
```