Section 15: Handling Forms in Angular Apps: Table of Contents

[182. Module Introduction 2](#_Toc64638025)

[183. Why do we Need Angular’s Help? 2](#_Toc64638026)

[184. Template-Drive (TD) vs Reactive Approach 2](#_Toc64638027)

[186. TD: Creating the Form and Registering the Controls 3](#_Toc64638028)

[187. TD: Submitting and Using the Form 3](#_Toc64638029)

[189. TD: Accessing the Form with @ViewChild 4](#_Toc64638030)

[190. TD: Adding Validation to check User Input 4](#_Toc64638031)

[191. Built-in Validators & Using HTML5 Validation 5](#_Toc64638032)

[192. TD: Using the Form State 5](#_Toc64638033)

[193. TD: Outputting Validation Error Messages 6](#_Toc64638034)

[194. TD: Set Default Values with ngModel Property Binding 6](#_Toc64638035)

[195. TD: Using ngModel with Two-Way-Binding 7](#_Toc64638036)

[196. TD: Grouping Form Controls 7](#_Toc64638037)

[197. TD: Handling Radio Buttons 8](#_Toc64638038)

[198. TD: Setting and Patching Form Values 8](#_Toc64638039)

[200. TD: Resetting Forms 9](#_Toc64638040)

[202. Reactive: Setup 9](#_Toc64638041)

[203. Reactive: Creating a Form in Code 10](#_Toc64638042)

[204. Reactive: Syncing HTML and Form 10](#_Toc64638043)

[205. Reactive: Submitting the Form 11](#_Toc64638044)

[206. Reactive: Adding Validation 11](#_Toc64638045)

[207. Reactive: Getting Access to Controls 11](#_Toc64638046)

[208. Reactive: Grouping Controls 12](#_Toc64638047)

[209. Fixing a Bug 12](#_Toc64638048)

[210. Reactive: Arrays of Form Controls (FormArray) 13](#_Toc64638049)

[211. Reactive: Creating Custom Validators 14](#_Toc64638050)

[212. Reactive: Using Error Codes 14](#_Toc64638051)

[213. Reactive: Creating a Custom Async Validator 15](#_Toc64638052)

[214. Reactive: Reacting to Status or Value Changes 15](#_Toc64638053)

[215. Reactive: Setting and Patching Values 15](#_Toc64638054)

[216. [OPTIONAL] Assignment Solution 16](#_Toc64638055)

# 182. Module Introduction

With single-page applications like those made in Angular, there is no option to submit forms to the server normally. Forms need to be handled through Angular, which Angular provides many tools for assisting with. Things can then be submitted to the server via an Http service.

Angular not only helps with getting user-entered values, but also will help with checking if an input is valid and changing the style of the form for invalid controls.

# 183. Why do we Need Angular’s Help?

The key features of a form object might look like this for a simple form:

1. {
2. value: {
3. name: 'Max',
4. email: 'test@test.com'
5. }
6. valid: true
7. }

This type of format would make entered data easy to manipulate and validate as needed. Luckily, Angular provides a means of representing user-submitted data in this manner.

# 184. Template-Drive (TD) vs Reactive Approach

Angular offers two approaches for handling forms:

1. Template-Driven (TD) - you simply setup the form in the template (HTML) and Angular will automatically infer the structure of the form via a Form Object from the DOM.

2. Reactive - you define the structure of the form in Typescript, setup the HTML, and manually connect them. Gives you greater control/fine-tuning ability.

# 186. TD: Creating the Form and Registering the Controls

In order to handle forms with Angular, the FormsModule needs to be imported from @angular/forms. In the template, the tag <form></form> essentially serves as a selector for some angular directive, which then creates a javascript object representation of the form for you. However, controls are not automatically loaded into object because you might be using 3rd party controls Angular is not familiar with, or a control that doesn't store data. In order to add a control to the object, you need to include the ngModel directive in addition to the HTML name attribute on the element as follows:

1. <input
2. type="text"
3. id="username"
4. class="form-control"
5. ngModel
6. name="username">

# 187. TD: Submitting and Using the Form

In order to submit a form an get access to user-entered data, the Angular event directive, (ngSubmit) can be used set in an expression to a custom function on the form element tag. Angular intercepts the way HTML usually handles form submissions so that requests are not sent to a server when a form is submitted. A local reference can be used to get access to the form element (which is of type HTMLFormElement) in the function. However, you can access to the data exposed by the form element via Angular by setting the local reference equal to ngForm.

HTML:  
<form (ngSubmit)="onSubmit(f)" #f="ngForm">

To access this in the typescript, you should change your input type assignment to the custom submit function from HTMLFormElement to NgForm (imported from @angular/forms). Ex:

1. onSubmit(form: NgForm) {
2. console.log(form);
3. }

# 189. TD: Accessing the Form with @ViewChild

Another way to gain access to form data is with @ViewChild (imported from @angular/core). To do this, we simply pass in the local reference name on the form to @ViewChild and assign the element object as a variable of type NgForm. This can be useful in places where you want to access the Form object prior to submitting the form.

1. @ViewChild('f') signupForm: NgForm;
2. onSubmit() {
3. console.log(this.signupForm);
4. }

# 190. TD: Adding Validation to check User Input

Validators are directives in Angular that can be placed on form controls in the template to ensure that specific criteria are met in order for the form to be valid. required and email are two examples of validators that can be used.

Controls with the required directive must be completed for the form to be valid upon submission. And similarly, input controls with the email directive must contains a valid email address for the form to be valid upon submission. If all validator criteria are not met, then the NgForm object's valid property will return false. Likewise, the individual FormControl objects themselves will have valid properties that return true or false depending on whether the validator criteria is met.

Template ex:

1. <input
2. type="email"
3. id="email"
4. class="form-control"
5. ngModel
6. name="email"
7. required
8. email>

# 191. Built-in Validators & Using HTML5 Validation

Which Validators do ship with Angular?

Check out the Validators class: <https://angular.io/api/forms/Validators> - these are all built-in validators, though that are the methods which actually get executed (and which you later can add when using the reactive approach).

For the template-driven approach, you need the directives. You can find out their names, by searching for "validator" in the official docs: <https://angular.io/api?type=directive> - everything marked with "D" is a directive and can be added to your template.

Additionally, you might also want to enable HTML5 validation (by default, Angular disables it). You can do so by adding the ngNativeValidate to a control in your template.

# 192. TD: Using the Form State

One way to use the form state is by disabling the submit button until the form as a whole has been deemed valid. This is easy enough to do by using the local reference for the NgForm and binding the disabled property of the button to the valid property of the form. Ex:

1. <button
2. class="btn btn-primary"
3. type="submit"
4. [disabled]="!f.valid">
5. Submit
6. </button>

Another way to use the form state is when Angular compiles the DOM: there are CSS classes that are added to the form controls based on things such as whether the initial value has changed (ng-pristine, ng-dirty), whether a control has been touched at all (ng-touched, ng-untouched), or whether a control is valid or not (ng-valid, ng-invalid). Using these classes, we can style the form accordingly. For example, we can add a red border when an input control has been touched but is still invalid:

1. input.ng-invalid.ng-touched {
2. border: 1px solid red;
3. }

# 193. TD: Outputting Validation Error Messages

Individual controls can have access granted by assigning a local reference on the control and setting it equal to "ngModel", which gives access to the control information initially exposed by ngModel. With the local reference, you can then do things like add help notes when a control hasn't been completed properly. Ex:

1. <input
2. type="email"
3. id="email"
4. class="form-control"
5. ngModel
6. name="email"
7. required
8. email
9. #email="ngModel">
10. <span
11. class="help-block"
12. \*ngIf="!email.valid && email.touched">
13. Please enter a valid email!
14. </span>

# 194. TD: Set Default Values with ngModel Property Binding

To set a default value for a control, all you must do is put brackets around the ngModel directive and one-way bind it to the default value desired. Ex:

Template:

1. <select
2. id="secret"
3. class="form-control"
4. [ngModel]="defaultQuestion"
5. name="secret">
6. <option value="pet">Your first Pet?</option>
7. <option value="teacher">Your first teacher?</option>
8. </select>

Typescript:

1. defaultQuestion = 'teacher';

# 195. TD: Using ngModel with Two-Way-Binding

On form controls, ngModel can be added as an attribute alone to pass along control data to NgForm. ngModel can also be used with one-way binding to set a default value for a control. Unsurprisingly, ngModel can yet be used with two-way binding as well, in order to get access to the user-inputted data in real-time. However, it should be noted that submitting a form will take a snapshot of the data at the time, and information changed after the form was submitted will not be included in the NgForm object stored. Two-way binding example:

1. <div class="form-group">
2. <textarea
3. name="questionAnswer"
4. rows="3"
5. class="form-control"
6. [(ngModel)]="answer">
7. </textarea>
8. </div>
9. <p>Your reply: {{ answer }} </p>

# 196. TD: Grouping Form Controls

In order to group form controls together, you can simply wrap them in a div and then add the ngModelGroup directive as an attribute on the div element. Since a name is not applied to a group, you will set ngModelGroup equal to a name of your choice. The form object for the group can be accessed in the same way that it is accessed for a form control, by setting a local reference equal to "ngModelGroup". Ex:

1. <div
2. id="user-data"
3. ngModelGroup="userData"
4. #userData="ngModelGroup">
5. ...
6. </div>
7. <p \*ngIf="!userData.valid && userData.touched">User Data is invalid!</p>

# 197. TD: Handling Radio Buttons

To handle a radio button form control, you'll first want to have your options included in an array in the typescript files. Once this is done, you can use \*ngFor on a div of class "radio" to iterate through the array of radio button options. Inside the div, you can create your input control and set the type, name, value (equal to for loop iteration value), and include your ngModel and validator directives. Note that you'll need to use String Interpolation to get the option names to show up next to the buttons. Ex:

1. <div class="radio" \*ngFor="let gender of genders">
2. <label>
3. <input
4. type="radio"
5. name="gender"
6. ngModel
7. [value]="gender"
8. required>
9. {{ gender }}
10. </label>
11. </div>

# 198. TD: Setting and Patching Form Values

In a case where we might want to insert a value into a form control at the click of a button, we can utilize a method with the NgForm @ViewChild instance that sets the values of the form. In this case, we use the setValue() method, which takes an object that must contain values for all of the properties in the form. With this, we can set a value by clicking a button tied to a function.

1. @ViewChild('f') signupForm: NgForm;
3. suggestUserName() {
4. const suggestedName = 'Superuser';
5. this.signupForm.setValue({
6. userData: {
7. username: suggestedName,
8. email: ''
9. },
10. secret: 'pet',
11. questionAnswer: '',
12. gender: 'Male'
13. })
14. }

One problem with setValue() is that it will overwrite any values that have been entered, which is bad if you are only trying to change the value of a single control. Therefore, a better method is to access the form object and then the patchValue() method on the @ViewChild instance. With this, you input an object that contains the values of just the form controls you wish to change. This way, other fields aren't overwritten. Ex:

1. @ViewChild('f') signupForm: NgForm;
3. suggestUserName() {
4. const suggestedName = 'Superuser';
5. this.signupForm.form.patchValue({
6. userData: {
7. username: suggestedName
8. }
9. });
10. }

# 200. TD: Resetting Forms

Another method that can be accessed on an NgForm instance in the reset() method, which can be used to clear a form after it has been submitted. However, it should be noticed that this will clear even the default values. Your defaults can be restored by passing a complete object of the form values into the input argument for the reset() method, similar as to what was done with the setValue() method. Ex:

1. onSubmit() {
2. this.submitted = true;
3. this.user.username = this.signupForm.value.userData.username;
4. this.user.email = this.signupForm.value.userData.email;
5. this.user.secretQuestion = this.signupForm.value.secret;
6. this.user.answer = this.signupForm.value.questionAnswer;
7. this.user.gender = this.signupForm.value.gender;
9. this.signupForm.reset();
10. }

# 202. Reactive: Setup

To get started with creating a reactive form, you need to make sure you have the ReactiveFormsModule imported from @angular/forms in the app module in place of the regular FormsModule.

A variable of type FormGroup is used as the base for our form. This makes sense since a form at the end of the day is just a group of controls. Like the ReactiveFormsModule, FormGroup also needs to be imported from @angular/forms.

# 203. Reactive: Creating a Form in Code

A form should be initialized in a lifecycle hook that is called prior to the template being rendered. To initialize a form, you'll want to set your variable of type FormGroup equal to a new instance of the class, and create an object with the form controls entered as key-value pairs as the constructor input argument.

It can be useful to wrap the key names in single quotation marks to ensure that during minification, the property names are kept since they'll be referenced in the HTML code. You'll want to enter the values as new FormControl class instances, which can be imported from @angular/forms. The constructor can take null to leave the value empty, or it can pass a default value as the first argument. Ex:

1. ngOnInit() {
2. this.signupForm = new FormGroup({
3. 'username': new FormControl(null),
4. 'email': new FormControl(null),
5. 'gender': new FormControl('male')
6. });
7. }

# 204. Reactive: Syncing HTML and Form

With a form element present in the HTML, Angular will automatically try to create a new FormGroup object. This needs to be overwritten in order to sync the template form content with our custom form, and can be done by property binding the formGroup directive to our form object. Ex:

1. <form [formGroup]="signupForm">

To tie together the template controls and our custom FormControl object, we'll want to name the control template elements using formControlName instead of just name. Ex:

1. <input
2. type="text"
3. id="username"
4. formControlName="username"
5. class="form-control">

# 205. Reactive: Submitting the Form

As with the template-driven approach, we can submit the form by including a the ngSubmit directive on the form element and event binding it to a custom function. However, unlike with the template-driven approach, we don't need to pass in a local reference to the form element since our form was created in and can be accessed in the typescript. Ex:

1. <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">

# 206. Reactive: Adding Validation

To use validators with the reactive form approach, you can enter them in the second input argument to the constructor when creating a new FormControl instance. They are entered by entering Validators (imported from @angular/forms) and then calling the validator methods as references (without parenthesis). References to the methods are used so that the method can be executed whenever Angular detects that the content of the form as changed. For multiple validators, an array of them should be entered.

1. this.signupForm = new FormGroup({
2. 'username': new FormControl(null, Validators.required),
3. 'email': new FormControl(null, [Validators.required,
4. Validators.email]),
5. 'gender': new FormControl('male')
6. });

# 207. Reactive: Getting Access to Controls

The reactive approach has a slightly different way of accessing form control state values in the template. We cannot use local references set to ngModel since the reactive approach doesn't generate an ngModel for the data. Instead, we can access it via our typescript form object and calling the get() method with the control name as the input. The overall form's state would be obtained by simply remove the get() method call in the below example:

1. <span
2. \*ngIf="!signupForm.get('username').valid &&
3. signupForm.get('username').touched"
4. class="help-block">
5. Please enter a valid username!
6. </span>

# 208. Reactive: Grouping Controls

Grouping together controls in the reactive approach requires a few changes. First, the controls listed in the form object initialization should be inserted into another new FormGroup instance within the overall new form FormGroup instance. Ex:

1. this.signupForm = new FormGroup({
2. 'userData': new FormGroup({
3. 'username': new FormControl(null, Validators.required),
4. 'email': new FormControl(null, [Validators.required,
5. Validators.email]),
6. }),
7. });

We'll want to group our template control elements under a div with the formGroupName directive and make sure it's equal to the group name in the typescript. Ex: <div formGroupName="userData">...</div>

However, this might change our get() method calls. To adjust, we need to enter the group name before the control name separated by periods. Ex: signupForm.get('userData.username')

# 209. Fixing a Bug

In the next lecture, we'll add some code to access the controls of our form array:

\*ngFor="let hobbyControl of signupForm.get('hobbies').controls; let i = index"

This code will fail as of the latest Angular version.

You can fix it easily though. Outsource the "get the controls" logic into a method of your component code (the .ts file):

1. getControls() {
2. return (<FormArray>this.signupForm.get('hobbies')).controls;
3. }

In the template, you can then use:

\*ngFor="let hobbyControl of getControls(); let i = index"

**Alternatively**, you can set up a getter and use an alternative type casting syntax:

1. get controls() {
2. return (this.signupForm.get('hobbies') as FormArray).controls;
3. }

and then in the template:

\*ngFor="let hobbyControl of controls; let i = index"

This adjustment is required due to the way TS works and Angular parses your templates (it doesn't understand TS there).

# 210. Reactive: Arrays of Form Controls (FormArray)

Let's say we want to enable to user to click a button and add controls programmatically. To do so, we first tie a button to a custom function, and enter the following code for example in the function (Although we also need to make sure we add a new FormArray instance in our form initialization: 'hobbies': new FormArray([])):

1. onAddHobby() {
2. const control = new FormControl(null, Validators.required);
3. (<FormArray>this.signupForm.get('hobbies')).push(control);
4. }

This lets us create a new control with each button click. We can then loop through our getControls() array (see lesson 209) to show the input for each control added (which use the index as the key, and the user-input as the value)

1. <div
2. class="form-group"
3. \*ngFor="let hobbyControl of getControls(); let i = index">
4. <input type="text" class="form-control" [formControlName]="i">
5. </div

# 211. Reactive: Creating Custom Validators

A validator is just a function that checks for the validity of a control. Below, it is shown how a custom validator receives the form control as an input, and should produce an object with a key value pair of a string error code and a boolean value. When a validator finds a forbidden control value, it needs to return an object with the error code string and true, otherwise it should always return false (Note that indexOf() returns -1 when it doesn't find the value being searched for).

1. forbiddenNames(control: FormControl): {[s: string]: boolean} {
2. if (this.forbiddenUsernames.indexOf(control.value) !== -1) {
3. return {'nameIsForbidden': true};
4. }
5. return null;
6. }

When entering the validator into the form initialization, it's important that you append .bind(this) onto the reference, which binds the validator to the class it's created in.

1. 'username': new FormControl(null, this.forbiddenNames.bind(this)),

# 212. Reactive: Using Error Codes

Each validator produces a string error code key with a boolean value when the validator check fails, which can be found under the errors property of a form control. These error codes can be checked for as done below to display specific error code messages. Ex:

1. <span \*ngIf="signupForm.get('userData.username').errors['required']">
2. This field is required!
3. </span>

# 213. Reactive: Creating a Custom Async Validator

When checking something like username availability, you may need to reach out to a web service in order to validate an entered name, which is an asynchronous operation. So we need async validators that are able to wait for a true or false response. As seen in the example below, async validators are similar to regular custom validators, except they expect promises rather than directly returning an error object.

1. forbiddenEmails(control: FormControl): Promise<any> | Observable<any> {
2. const promise = new Promise<any>((resolve, reject) => {
3. setTimeout(() => {
4. if (control.value === 'test@test.com') {
5. resolve({'emailIsForbidden': true});
6. } else {
7. resolve(null)
8. }
9. },1500);
10. });
11. return promise;
12. }

Unlike custom validators, async validators are entered as the third argument in a new FormControl's constructor.

# 214. Reactive: Reacting to Status or Value Changes

There are two observables that can be used to react to either status or value changes on a reactive form. The first is valueChanges, which returns an object of the form values every time a form control is updated. The second is statusChanges, which will return either VALID or INVALID depending on the validity of the form, as well as PENDING whenever an async validator is pending.

1. this.signupForm.valueChanges.subscribe(
2. (value) => console.log(value)
3. )
4. this.signupForm.statusChanges.subscribe(
5. (status) => console.log(status)
6. )

# 215. Reactive: Setting and Patching Values

Like with the template-drive approach, the reactive approach also lets you utilize the setValue(), patchValue(), and reset() methods on a form object.

# 216. [OPTIONAL] Assignment Solution

We have the ability to create our own custom validators bundle that is called the same way as Validators, and doesn't require the bind(this) method. To do so, just create a new typescript file with the class name of the bundle, and place your validator methods inside as static methods. Afterward, you just have to import your custom class. Ex:

custom-validators Typescript:

1. import { FormControl } from '@angular/forms';
2. import { Observable } from 'rxjs/Observable';
4. export class CustomValidators {
5. static invalidProjectName(control: FormControl): {[s: string]:
6. boolean} {
7. if (control.value === 'Test') {
8. return {'invalidProjectName': true};
9. }
10. return null;
11. }
12. }

Component Typescript:

1. import { CustomValidators } from './custom-validators';