



Angular Form Essentials



A practical guide on how to build
forms with Angular.

Cory Rylan

Angular Form Essentials

A practical beginners guide on how to build forms with Angular.

Cory Rylan

This book is for sale at <http://leanpub.com/angular-form-essentials>

This version was published on 2020-06-28



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2020 Cory Rylan

Contents

Introduction	1
About the Author	1
What is this Book About?	1
Technical Prerequisites	1
Source Code Examples	2
Chapter 1 - HTML Forms	3
HTML	3
Events	4
Properties	5
Chapter 2 - Reactive Forms	6
Reactive Forms Module	6
Form Control Validation	8
Chapter 3 - Form Group API	12
Form Builder	15
Chapter 4 - Custom Form Validation	18
Optional Chaining and Dynamic Control Accessors	20
Custom Form Level Validation	21
Chapter 5 - Advanced Form Types	24
Inline/Text Based Inputs	24
Checkbox	27
Select	28
Distance	29
Radio	29
Chapter 6 - Accessibility with Angular Forms	31
HTML Form Basics	31
Angular Reactive Forms	33
Multiple Angular Form Groups	35
Accessible Form Validation	37
Accessible Inline Form Validation	37

CONTENTS

Angular Form Group Validation	41
Accessible Form Validation Summary	44
Accessible Form Status Messages	46
Chapter 7 - Async Form Data	50
Form SetValue and PatchValue	57
Chapter 8 - Dynamic Form Controls	60
Dynamic Select Input	60
Dynamic Radio Lists	63
Chapter 9 - Dynamic Form Controls with Form Array	67
Validation	70
Chapter 10 - Async Form Validation	73
Chapter 11 - Template Form API	78
Chapter 12 - Custom Form Controls	82
Custom Form Controls with Reactive Forms	83
Custom Form Controls with Template Forms and NgModel	83
Building a Custom Form Control	84
Chapter 13 - Observables and RxJS	94
JavaScript Promise	94
RxJS Observables	95
Unsubscribing	95
Operators	97
Chapter 14 - Reactive Forms	99
ValueChanges Observable	102
HttpClient	102
SwitchMap Operator	103
Debounce Time	106
Chapter 15 - Reusable Form Components	110
Sub Forms with Control Value Accessor	112
Reusable Password Creation Form	117
Chapter 16 - Validation Management	122
Chapter 17 - Code Examples	129

Introduction

About the Author

Cory Rylan is a Front End Web Developer and Google Developer Expert for Angular and Web technologies. He specializes in high performance Progressive Web Apps and enterprise level web app development. He currently works at VMware on the [Clarity UI](https://clarity.design)¹ team. Clarity is an open source Angular Component library and Design System. He speaks at conferences, meet-ups and helps teach Angular Bootcamp a three day in person class. When not speaking and teaching he is writing all about Angular and Web Components on his blog at coryrylan.com². You can also follow him on Twitter [@coryrylan](https://twitter.com/coryrylan)³;

What is this Book About?

This book's goal is to provide an introduction and practical guide of how to create and use Angular Forms. By the end of this book readers can feel confident in creating complex and custom Angular forms by leveraging the powerful Reactive Forms API.

Technical Prerequisites

The code examples will primarily use the Angular CLI. This book covers beginner to intermediate web specific technologies in HTML, CSS and JavaScript. Some of the topics used in this book are listed below.

- HTML
 - HTML5 tags
 - HTML inputs and forms
- CSS
 - Basic CSS properties for styling forms
- JavaScript/TypeScript
 - Familiar with ES2015 JavaScript syntax (classes, template strings, modules)
 - TypeScript basics such as static typing
 - Basic use of NPM (node package manager)
 - Angular CLI and basic Angular 2+ knowledge

¹<https://clarity.design>

²<https://coryrylan.com>

³<https://twitter.com/coryrylan>

Source Code Examples

You can find a link to access and download the source code of all examples in the last chapter of this book.

Chapter 1 - HTML Forms

HTML forms can be a user's worst nightmare when using the Web. Forms with poor accessibility or poor user experience can lead to lost data, frustration or the user leaving our application altogether.

As a professional building for the Web, we have a responsibility to build forms that help the user accomplish their goals and make them accessible to everyone.

This book will teach you how to build high-quality forms in Angular effectively. First, we need to cover some basics of HTML forms so we can distinguish what is built into the browser and what Angular provides for us.

HTML

HTML forms provide much functionality without any JavaScript framework. There are many different input types such as text, color, numbers, and even native datepickers. With the introduction of HTML5, we also got built-in browser validation such as the `required` attribute. Let's look at a basic HTML form.

HTML Forms

Name

Basic HTML Form

```
1 <form>
2   <label for="name">Name</label>
3   <input id="name" />
4   <button>Submit</button>
5 </form>
```

This form is about as simple as you can get with creating an HTML form. The form tag semantically marks that this content is a form. The form tag helps screen readers and gives us a way to listen to submit events we will see shortly.

Our form has just a single text input. By default with HTML inputs use the `type="text"` attribute if not defined by the developer. We provide some basic accessibility by associating an HTML label element to the input. With labels, we use the `for` attribute and give it an `id` name that exists on the input. Labels are essential for accessibility as it provides a way for screen readers to read what the input is for users with vision impairments.

Events

With Angular, we could use just the template binding syntax to use this form without using the Angular forms API at all. Let's take a look.


```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-html-forms-example',
5   templateUrl: './html-forms-example.component.html'
6 })
7 export class HtmlFormsExampleComponent {
8   initialValue = 'Hello World';
9
10  submit(event) {
11    console.log(event);
12  }
13 }
```

```
1 <form (submit)="submit($event)">
2   <label for="name">Name</label>
3   <input id="name" [value]="initialValue" />
4   <button>Submit</button>
5 </form>
```

With Angular, we can listen to any event with the event binding syntax (`eventName`). On our form element, we can listen to the submit event of the form. The submit event occurs whenever the user pushes the enter key or the submit button. By default buttons inside a form element triggers the submit event to fire unless there is a `type="button"` attribute on the button element.

When the submit event fires, we can capture the event value using the `$event` keyword to pass the value into our `submit()` method we defined in our Angular component.

Properties

```
1 <input id="name" [value]="initialValue" />
```

With Angular's template syntax, we can also set properties of HTML elements. To do this, we use the `[myprop]` property binding syntax. In this example, we set the `[value]` property of the HTML input to the `initialValue` property we defined in our TypeScript.

Using just Angular's template syntax, we can accomplish quite a bit of functionality with basic HTML forms. What about more complex scenarios such as validation, error handling and accessibility? In our next section, we will learn the basics of the Angular Reactive Forms Module.

Chapter 2 - Reactive Forms

The Angular forms API has two primary APIs to choose from Template Forms Module and the Reactive Forms Module. This book will primarily focus on the Reactive Forms Module. The Template Forms Module will be covered in a later chapter.

The Reactive Forms Module has many benefits for building forms. The Reactive Forms API allows us to keep a lot of the logic to construct and validate our forms out of our component template and in the moves it to the TypeScript making it easier to debug and test. The Reactive Forms Module also provides an excellent API for listening to form updates and changes, hence the “Reactive” name. We will cover the Reactive parts of the API in a later chapter in detail.

To start let’s take a look at a basic HTML input using the Reactive Forms API.

Reactive Forms Module

To start using the Reactive Forms in our Angular application, we need to import the `ReactiveFormsModule` into our application and add it to the `AppModule`.

```
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { ReactiveFormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6
7 @NgModule({
8   declarations: [AppComponent],
9   imports: [
10     CommonModule,
11     ReactiveFormsModule
12   ]
13 })
14 export class AppModule { }
```

Make sure to import the `ReactiveFormsModule` and not `FormsModule` from the `@angular/forms` package. The `FormsModule` is for the template based API we will cover in a later chapter.

Now that we have the correct module we can start using the forms API. In our application component, we are going to create a single form control instance. This form control is responsible for managing the state of our text input including the value and validation of the input.

```

1  import { Component } from '@angular/core';
2  import { FormControl } from '@angular/forms';
3
4  @Component({
5    selector: 'app-form-control-example',
6    templateUrl: './form-control-example.component.html'
7  })
8  export class FormControlExampleComponent {
9    name = new FormControl('Angular Forms Rock!');
10 }

```

The `FormControl` class from `@angular/forms` helps us manage our text input. When we instantiate the `FormControl` we can pass it an initial value to render into our HTML input. In our example, the text box is set with the value of “Angular Forms Rock!”. Let’s take a look at the component template next.

```

1  <label for="name">Name</label>
2  <input [formControl]="name" id="name" />
3
4  <p>Value: {{name.value}}</p>

```

In our template, we have a single HTML input defined. Notice we do not have an HTML form tag on the page. When creating a single input, the API does not require you to use a form tag. In later chapters when we cover multi-input forms and accessibility, we will see why we want to default to using a form element.

In the example, we associate the input with the form control instance by using the `[formControl]="name"` binding. When binding to the `[formControl]` property we pass in the property in our Form Control instance we created in the TypeScript which was `name`. Once this assignment is made, we can access all kinds of information from the form control.

```

1  {
2    "validator": null,
3    "asyncValidator": null,
4    "pristine": true,
5    "touched": false,
6    "value": "Angular Forms Rock!",
7    "status": "VALID",
8    "errors": null,
9    "valueChanges": { ... },
10   "statusChanges": { ... }
11 }

```

We can see this information includes the value of the input, validation errors and whether the user has updated the input. We will dig into this in more detail soon. Next, let's add a validator to require the user to add a name in our form.

Form Control Validation

Continuing to build on our previous example we will add our first validation rule for a new input. Client-side validation provides a better user experience for our forms. Client-side validation allows us to help the user fill out the form correctly and provide immediate feedback. Client-side validation, however, is just that, for user experiences.

Because the code we write in Angular is running on the client's machine, this means we cannot trust it. The user has access to the code and browser and can manipulate or disable our client-side validation. Always validate your inputs on the server side of your application.

Now for our use case, we want to create an email input and make it required for the user to fill out. We also want to be able to provide a UI message to express any mistakes the user may have made while adding their email address. First, we need to create our `FormControl` instance like in our previous section.

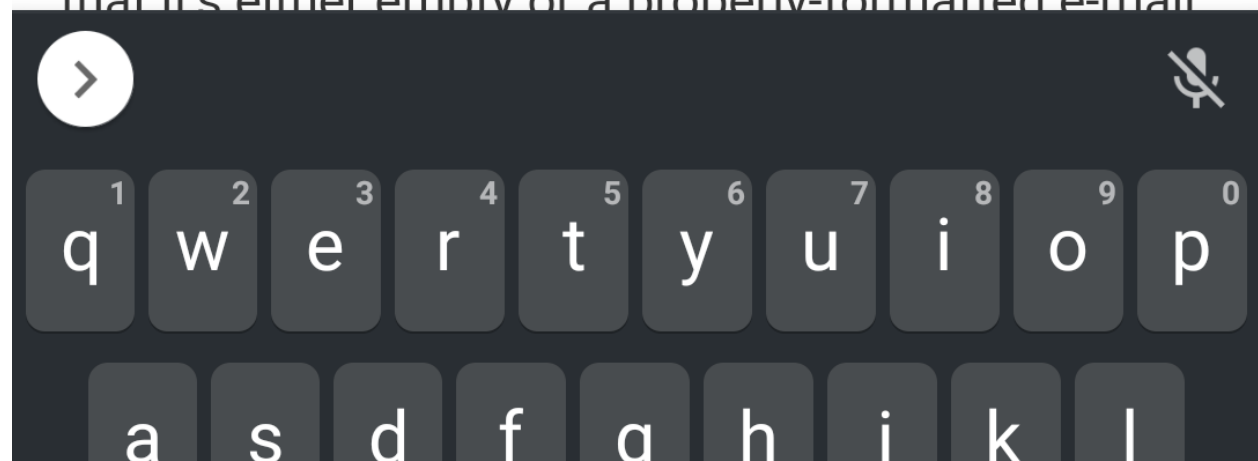
```
1 import { Component } from '@angular/core';
2 import { FormControl } from '@angular/forms';
3
4 @Component({
5   selector: 'app-input-validation-example',
6   templateUrl: './input-validation-example.component.html'
7 })
8 export class InputValidationExampleComponent {
9   email = new FormControl('');
10 }
```

```
1 <label for="email">Email</label>
2 <input [formControl]="email" id="email" type="email" />
```

Notice in our template we have `type="email"` defined on our input. This attribute is an HTML5 feature. By defining the input type to email some on-screen keyboards, especially on mobile devices, will show a keyboard that is designed entering email addresses. This includes making the `@` character easier to find.



The input value is automatically validated to ensure that it's either empty or a properly-formatted e-mail



Now that we have our email input created we can add our first validation rule. In our TypeScript component, we can import the built-in validators utility service from `@angular/forms`.

```
1 import { Component } from '@angular/core';
2 import { FormControl, Validators } from '@angular/forms';
3
4 @Component({
5   selector: 'app-input-validation-example',
6   templateUrl: './input-validation-example.component.html'
7 })
8 export class InputValidationExampleComponent {
9   email = new FormControl('', Validators.required);
10 }
```

The second parameter of our `FormControl` can take one to many validator functions. When the user enters in a value for our input Angular executes the validator functions in the order they are listed. A validator function takes in the input value and returns an error message if it does not meet the validation rule. In this example, the validator returns the `required` error if the user has not added any text to the input.

```
1 <label for="email">Email</label>
2 <input [formControl]="email" id="email" type="email" />
3
4 <div *ngIf="email.touched && email.hasError('required')" class="error">
5   email is required
6 </div>
```

When we show the message to the user, we want to wait until the user has “touched” or focused the input. We don’t want to show the message immediately as soon as the user sees the input. In the template, we prevent the message from being seen under two conditions. We conditionally show our validation message checking `email.touched` and if `email.hasError('required')`.

We can add additional validators to our input. The Angular `Validators` service also has a built-in email validator that checks if a given input value contains a valid email format.

```
1 import { Component } from '@angular/core';
2 import { FormControl, Validators } from '@angular/forms';
3
4 @Component({
5   selector: 'app-input-validation-example',
6   templateUrl: './input-validation-example.component.html'
7 })
8 export class InputValidationExampleComponent {
9   email = new FormControl('', [Validators.required, Validators.email]);
10 }
```

In our TypeScript, we update our form control to contain an array of validators. In our template, we can show a second error message to the user.

```
1 <label for="email">Email</label>
2 <input [formControl]="email" id="email" type="email" />
3
4 <div *ngIf="email.touched && email.hasError('required')" class="error">
5   email is required
6 </div>
7
8 <div *ngIf="email.touched && email.hasError('email')" class="error">
9   invalid email address
10 </div>
```

The error messages show in the order that the validators are listed in the form control validator array. In later chapters, we will show how to create our own custom validator functions. In the next chapter, we will cover how to create more complex multi-input forms with the Form Group API.

Chapter 3 - Form Group API

Now that we have a basic understanding of Angular Form Controls we can take what we know and build out more complex forms with multiple inputs. With Angular Reactive Forms, we can manage multiple Form Control instances by using the `FormGroup` class. Let's take a look at what our example form looks like.

Form Group

First Name

Last Name

Email

Submit

Form Group Example

With our form, we have three form controls, first name, last name, and email address. We also have necessary validation against the email form control.


```

1  import { Component } from '@angular/core';
2  import { FormControl, Validators, FormGroup } from '@angular/forms';
3
4  @Component({
5    selector: 'app-form-group-example',
6    templateUrl: './form-group-example.component.html'
7  })
8  export class FormGroupExampleComponent {
9    form = new FormGroup({
10      firstName: new FormControl(''),
11      lastName: new FormControl(''),
12      email: new FormControl('', [Validators.required, Validators.email]),
13    });
14
15    get email() {
16      return this.form.controls.email;
17    }
18
19    submit() {
20      console.log(this.form);
21    }
22  }

```

The `FormGroup` class imported from the `@angular/forms` package. The `FormGroup` class takes a JavaScript object where each key is the name of the individual `FormControl` instance. The `FormGroup` instance is stored as a property of the component class. We reference this property in our template to describe to Angular which form belongs to which property on our component.

We also use a `get email()` to shorthand the email control in our template. This allows us to simply refer to the email control as `email` in our template instead of having to repeatedly type out `this.form.controls.email` in our form template.

```

1  <form [formGroup]="form" (ngSubmit)="submit()">
2    <label for="first-name">First Name</label>
3    <input formControlName="firstName" id="first-name" />
4
5    <label for="last-name">Last Name</label>
6    <input formControlName="lastName" id="last-name" />
7
8    <label for="email">Email</label>
9    <input formControlName="email" id="email" />
10
11  <div *ngIf="email.touched && email.hasError('required')" class="error">

```

```

12     email is required
13 </div>
14
15 <div *ngIf="email.touched && email.hasError('email')" class="error">
16     invalid email address
17 </div>
18
19 <button>Submit</button>
20 </form>

```

In the template of our component we now have an HTML form element. We bind our form group to this form element with the `[formGroup]` property and pass a reference to the form property name that holds the `FormGroup` instance. Now that our inputs are in a form element and managed by Angular we can listen to submit events with `(ngSubmit)`. If the user clicks our default submit button or hits the enter key, it triggers the submit event for our form. By using a form element our form is also more accessible as it makes it very clear to screen readers the intent of the HTML and the inputs associated with it.

If we log out the form object, we can see the many properties the `FormGroup` instance provides us.

```

1  {
2    validator: null,
3    asyncValidator: null,
4    pristine: true,
5    touched: false,
6    asyncValidator: null,
7    controls: { firstName: FormControl, lastName: FormControl, email: FormControl }
8    dirty: (...)
9    disabled: (...)
10   enabled: (...)
11   errors: null
12   invalid: (...)
13   parent: (...)
14   pending: (...)
15   pristine: true
16   root: (...)
17   status: "INVALID"
18   statusChanges: EventEmitter { ... }
19   touched: false
20   untouched: (...)
21   updateOn: (...)
22   valid: (...)
23   validator: null

```

```
24   value: { firstName: "", lastName: "", email: "" }
25   valueChanges: EventEmitter { ... }
26 }
```

We can see by using the `FormGroup` we can check if the entire form is valid or dirty. We can also drill down into the `controls` property and access the individual `FormControl` instances of the form. Some of these properties such as `valueChanges` we will dig into more detail in a later chapter.

Form Builder

As our forms grow in size and complexity declaring form groups and form controls in our TypeScript can start to become verbose. To help counter this, Angular provides a utility service called `FormBuilder`. The `FormBuilder` service allows us to declare a form group composed of form controls with a less verbose API. Let's take a look at an example.

```
1  <form [formGroup]="form" (ngSubmit)="submit()">
2    <label for="firstName">First Name</label>
3    <input formControlName="firstName" id="firstName" />
4
5    <label for="lastName">Last Name</label>
6    <input formControlName="lastName" id="lastName" />
7
8    <label for="email">Email</label>
9    <input formControlName="email" id="email" />
10
11   <div *ngIf="email.touched && email.hasError('required')" class="error">
12     email is required
13   </div>
14
15   <div *ngIf="email.touched && email.hasError('email')" class="error">
16     invalid email address
17   </div>
18
19   <button>Submit</button>
20 </form>
```

Our template in our previous example stays precisely the same when using the `Form Builder` service.

```

1  import { Component } from '@angular/core';
2  import { Validators, FormGroup, FormBuilder } from '@angular/forms';
3
4  @Component({
5    selector: 'app-form-builder-example',
6    templateUrl: './form-builder-example.component.html'
7  })
8  export class FormBuilderExampleComponent {
9    form: FormGroup;
10
11   constructor(private formBuilder: FormBuilder) {
12     this.form = this.formBuilder.group({
13       firstName: [],
14       lastName: [],
15       email: ['', [Validators.required, Validators.email]],
16     });
17   }
18
19   submit() {
20     console.log(this.form.value);
21   }
22 }

```

We import the `FormBuilder` service from the `@angular/forms` package. Because `FormBuilder` is a service, we use Angular's dependency injection system to use it. Angular uses dependency injection to manage dependencies in our application. Dependency Injection allows us to decouple dependencies as well as make unit testing significantly easier as we can swap dependencies on the fly. To get an in-depth guide to dependency injection check out the Angular documentation.

When using a service with Angular, we request an instance of the service by adding it to the component constructor.

```

1  ...
2  export class FormBuilderExampleComponent {
3    form: FormGroup;
4
5    constructor(private formBuilder: FormBuilder) { }
6
7    ...
8  }
9  ...

```

When Angular creates an instance of our component, it sees that the constructor requires an instance of the `FormBuilder` service. Angular makes sure an instance is created and passed into

the constructor. We define the `private` keyword in the constructor as a shortcut. TypeScript will automatically take the `formBuilder` parameter and make it a `private` property of our class instance.

Now that we have an instance of the `FormBuilder` we can create our form group instance.

```
1  ...
2  export class FormBuilderExampleComponent {
3      form: FormGroup;
4
5      constructor(private formBuilder: FormBuilder) {
6          this.form = this.formBuilder.group({
7              firstName: [],
8              lastName: [],
9              email: ['', [Validators.required, Validators.email]],
10         });
11     }
12 }
13 ...
```

On the `FormBuilder` service there are a few methods. The method we use is `.group()`. The `group` method takes a JavaScript object with each key being the name of a form control instance. Our form has three form inputs. Instead of having to call the constructor of the `FormControl` instance for each input we can pass in the parameters directly. The `FormBuilder` expects each control to pass back an array as the parameter. The array's first value is the default value for that input. The second value in the array is one to many validators that are applied to that given input.

By using the `FormBuilder` service, we can quickly create forms and apply complex validation rules with a concise syntax.

Chapter 4 - Custom Form Validation

Angular provides many useful built-in form Validators such as the required and email validator. Sometimes in our applications, we need to enforce our own custom validation rules. Luckily for us, this is very straightforward with the Angular forms API.

In our example use case our form has a new password input field. Our password requirements state that a password must be at least 6 characters, a capital letter, and one number. We can build our custom password validator to check that the input meets these requirements. First, let's start with our component to see how we use a custom validator.

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder, Validators } from '@angular/forms';
3
4  import { passwordValidator } from './validators';
5
6  @Component({
7    selector: 'app-custom-validation-example',
8    templateUrl: './custom-validation-example.component.html'
9  })
10 export class CustomValidationExampleComponent {
11   form: FormGroup;
12
13   constructor(private formBuilder: FormBuilder) {
14     this.form = this.formBuilder.group({
15       password: ['', [Validators.minLength(6), passwordValidator]]
16     });
17   }
18
19   get password() {
20     return this.form.controls.password;
21   }
22
23   submit() { ... }
24 }
```

Our form component looks similar to our previous examples. We use the built-in validators class to set a minimum length requirement. The second validator is our custom passwordValidator function. This function is imported from a separate file containing our custom validator logic. Let's take a look at what the custom validator API looks like.

```

1  import { FormControl, ValidationErrors } from '@angular/forms';
2
3  export function passwordValidator(control: FormControl): ValidationErrors | null {
4    if (!containsNumber(control.value) || !containsUppercaseLetter(control.value)) {
5      return { invalidPassword: true };
6    } else {
7      return null;
8    }
9  }
10
11  ...

```

Our custom validator is just an ordinary JavaScript function. When Angular calls our validator, it passes a reference to our form control. With the form control reference, we can get the value and run our checks. The API expects one of two possible values returned. If valid it returns `null`, else return an object with the error name property marked `true`.

In our example, we check if the input contains a number or an uppercase letter, if not return our error message. In this example, we have some additional functions to figure out if the input meets these cases.

```

1  import { FormControl, ValidationErrors } from '@angular/forms';
2
3  export function passwordValidator(control: FormControl): ValidationErrors | null {
4    if (!containsNumber(control.value) || !containsUppercaseLetter(control.value)) {
5      return { invalidPassword: true };
6    } else {
7      return null;
8    }
9  }
10
11  function containsUppercaseLetter(value: string) {
12    return value.split('').find(v => v === v.toUpperCase() && !isNumber(v)) !== undefi\
13    ned;
14  }
15
16  function containsNumber(value: string) {
17    return value.split('').find(v => isNumber(v)) !== undefined;
18  }
19
20  function isNumber(value: any) {
21    return !isNaN(parseInt(value, 10));
22  }

```

Lastly, our template references the error message we defined in the custom validator to show the user any potential issues.

```

1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="password">Password</label>
3   <input formControlName="password" id="password" type="password" />
4
5   <ng-container *ngIf="password.touched">
6     <div *ngIf="password.hasError('minlength')" class="error">
7       Password must be at least six characters long.
8     </div>
9     <div *ngIf="password.hasError('invalidPassword')" class="error">
10      Passwords must contain at least one uppercase character and one number.
11    </div>
12  </ng-container>
13
14  <button>Submit</button>
15 </form>

```

Because custom validators are functions, we export easily to be reused across our application for any number of forms.

Optional Chaining and Dynamic Control Accessors

Alternatively we can use `this.form.get('password')` to get a reference to our control. Using `.get()` we can dynamically access the controls without using the `.controls` object reference. This can sometimes shorten the syntax in our templates. If your project is in strict mode, then Optional Chaining syntax `?.` will be necessary. The Optional Chaining syntax will check that the object is not null or undefined before evaluating the next statement, so an exception is not thrown. Since `.get()` is dynamic, we use the Optional Chaining to ensure the control is available.

```

1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="password">Password</label>
3   <input formControlName="password" id="password" type="password" />
4
5   <ng-container *ngIf="form.get('password')?.touched">
6     <div *ngIf="form.get('password')?.hasError('minlength')" class="error">
7       Password must be at least six characters long.
8     </div>
9     <div *ngIf="form.get('password')?.hasError('invalidPassword')" class="error">
10      Passwords must contain at least one uppercase character and one number.

```



```
11     </div>
12 </ng-container>
13
14     <button>Submit</button>
15 </form>
```

Optional chaining is an available feature in JavaScript and supported in TypeScript as of version 3.7.

Custom Form Level Validation

Building on top of what we have learned so far, we can take our custom validation one step further. When creating a new password, we typically would want the user to confirm the password by re-entering the password in a second password input. We can use Angular's custom validation API to check that the passwords match.

With Angular, we can apply a custom validator not just a single form control but an entire form allowing us to check the value of multiple form controls. Let's take a look and see an example.

```
1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="password">Password (required)</label>
3   <input formControlName="password" id="password" type="password" />
4   <div *ngIf="passwordIsValid" class="error">
5     Password must be at least 6 characters.
6   </div>
7
8   <label for="confirm">Confirm Password (required)</label>
9   <input formControlName="confirm" id="confirm" type="password" />
10
11   <div *ngIf="passwordsDoNotMatch" class="error">
12     Passwords must match.
13   </div>
14
15   <button>Create Account</button>
16 </form>
```

Looking at the template our form is pretty standard to what we have seen. At the bottom we do have a validation message that shows whenever the `passwordsDoNotMatch` property is true. Let's take a look at the TypeScript next.

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder, Validators } from '@angular/forms';
3
4  import { matchingInputsValidator } from './validators';
5
6  @Component({
7    selector: 'app-form-group-validation-example',
8    templateUrl: './form-group-validation-example.component.html'
9  })
10 export class FormGroupValidationExampleComponent {
11   form: FormGroup;
12
13   constructor(private formBuilder: FormBuilder) {
14     this.form = this.formBuilder.group({
15       password: ['', [Validators.required, Validators.minLength(6)]],
16       confirm: ['', Validators.required]
17     }, { validator: matchingInputsValidator('password', 'confirm') });
18   }
19
20   submit() {
21     console.log(this.form.value);
22   }
23
24   get passwordIsValid() {
25     return this.form.controls.password.valid && this.form.controls.password.touched;
26   }
27
28   get passwordsDoNotMatch() {
29     return this.form.errors && this.form.errors.mismatch && this.form.controls.confirm.touched;
30   }
31 }
32
33 }
```

The password form is constructed with the `FormBuilder` service like in our previous examples. With the `group` method we can pass a secondary options object. This options object can take a validator function. In this case, we pass it a custom validator we created called `matchingInputsValidator`. This validator takes two parameters, each for the two input names we want to compare.

```
1 import { FormGroup, ValidationErrors } from '@angular/forms';
2
3 export function matchingInputsValidator(firstKey: string, secondKey: string): ValidatorFn {
4   return (control: AbstractControl): ValidationErrors | null => {
5     if (control.get(firstKey)?.value !== control.get(secondKey)?.value) {
6       return { mismatch: true };
7     } else {
8       return null;
9     }
10  };
11 }
12 }
```

Angular expects a function that it can pass a form `AbstractControl` as the parameter and get back any potential validation errors. Since we have the entire `FormGroup` and not just a single `FormControl` we can compare any two values or conditions. In this example, we use a factory function pattern that returns another internal function. This pattern is necessary, so we can return a function that knows the names of the two input values we want to compare.

In our next chapter, we will cover all the various input types Angular forms can support.

Chapter 5 - Advanced Form Types

Angular forms support a wide variety of input types. So far we have covered a couple of input types such as `type="text"` and `type="password"`. This chapter will be a brief overview of how to create forms using additional input types such as selects, checkboxes, radios and more.

Inline/Text Based Inputs

This example we will see how to bind to various inline/text style inputs such as `type text`, `number`, `color` and `date`.

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder } from '@angular/forms';
3
4  @Component({
5    selector: 'app-advanced-form-types-example',
6    templateUrl: './advanced-form-types-example.component.html'
7  })
8  export class AdvancedFormTypesExampleComponent {
9    form: FormGroup;
10
11    constructor(private formBuilder: FormBuilder) {
12      this.form = this.formBuilder.group({
13        name: [''],
14        color: ['#ff0000'],
15        password: [''],
16        age: [100],
17        date: [new Date()]
18      });
19    }
20
21    submit() {
22      console.log(this.form.value);
23    }
24  }
```

With many of the inline/text-based input types we can bind them to our forms just like any regular text input using `formControlName`.

```
1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="name">Name (type="text")</label>
3   <input formControlName="name" id="name" />
4
5   <label for="color">Favorite Color (type="color")</label>
6   <input formControlName="color" id="color" type="color" />
7
8   <label for="password">Password (type="password")</label>
9   <input formControlName="password" id="password" type="password" />
10
11  <label for="age">Age (type="number")</label>
12  <input formControlName="age" id="age" type="number" />
13
14  <label for="date">Date (type="date")</label>
15  <input formControlName="date" id="date" type="date" />
16 </form>
```

The more advanced HTML5 input types work seamlessly with Angular forms. We can get native datepickers and color pickers from most modern browsers.

Advanced Form Types

Date (type="date")

April 2019 ▼

◀

●

▶

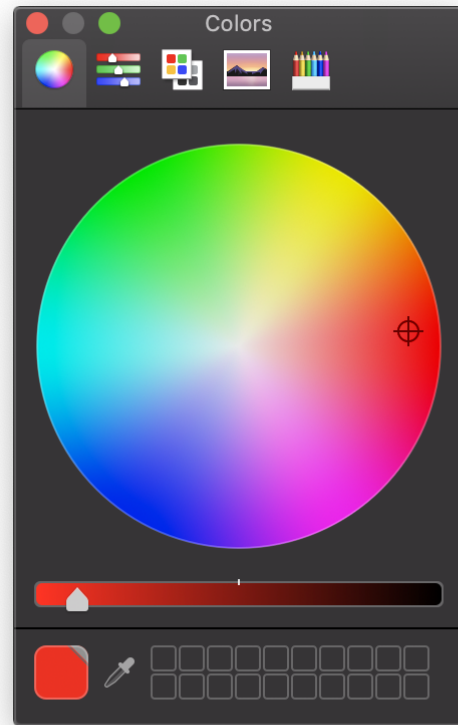
Sun	Mon	Tue	Wed	Thu	Fri	Sat
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4

Datepicker Input Example

Advanced Form Types

Favorite Color (type="color")

Submit



Color Picker Input Example

Checkbox

Checkbox while not text-based works the same way as binding to a text input.

```
1 <input formControlName="subscribe" type="checkbox" id="checkbox" />
2 <label for="checkbox">Subscribe (type="checkbox")</label>
```

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder } from '@angular/forms';
3
4  @Component({
5    selector: 'app-advanced-form-types-example',
6    templateUrl: './advanced-form-types-example.component.html'
7  })
8  export class AdvancedFormTypesExampleComponent {
9    form: FormGroup;
10
11    constructor(private formBuilder: FormBuilder) {
12      this.form = this.formBuilder.group({
13        subscribe: [true]
14      });
15    }
16
17    submit() {
18      console.log(this.form.value);
19    }
20  }
```

When binding to a checkbox Angular returns a boolean value as the control value.

Select

Selects bind with `formControlName` just like our previous examples but also have a bit more syntax in the template to define.

```
1  <label for="memory">Memory (select)</label>
2  <select formControlName="memory" id="memory" >
3    <option value="16">16 Gigabytes</option>
4    <option value="32">32 Gigabytes</option>
5    <option value="64">64 Gigabytes</option>
6    <option value="128">128 Gigabytes</option>
7  </select>
```



```
1 import { Component } from '@angular/core';
2 import { FormGroup, FormBuilder } from '@angular/forms';
3
4 @Component({
5   selector: 'app-advanced-form-types-example',
6   templateUrl: './advanced-form-types-example.component.html'
7 })
8 export class AdvancedFormTypesExampleComponent {
9   form: FormGroup;
10
11   constructor(private formBuilder: FormBuilder) {
12     this.form = this.formBuilder.group({
13       memory: ['32']
14     });
15   }
16
17   submit() {
18     console.log(this.form.value);
19   }
20 }
```

When setting the options of the select, we have to make sure to provide the value attribute for Angular to use and assign to the control value. Note that when using the value attribute the value returned always is a string type. In a later chapter, we will cover in detail how to create selects dynamically from async data.

Distance

The distance input type uses `formControlName` to bind the `FormControl`.

```
1 <label for="distance">Distance (type="range")</label>
2 <input formControlName="distance" id="distance" type="range" min="0" max="100" />
```

On the range input, you can use the `min` and `max` attributes to set a range of values. You can dynamically set these attributes by using the attribute binding syntax: `[attr.min]="minValue"`.

Radio

Radio inputs are one of the more complex built in input types to bind to a `FormControl`.

```
1 <span class="label">Region (type="radio")</span>
2 <label>
3   <input formControlName="region" type="radio" name="region" value="north-america" />
4   North America
5 </label>
6 <label>
7   <input formControlName="region" type="radio" name="region" value="south-america" />
8   South America
9 </label>
10 <label>
11   <input formControlName="region" type="radio" name="region" value="europe" />
12   Europe
13 </label>
```

```
1 import { Component } from '@angular/core';
2 import { FormGroup, FormBuilder } from '@angular/forms';
3
4 @Component({
5   selector: 'app-advanced-form-types-example',
6   templateUrl: './advanced-form-types-example.component.html'
7 })
8 export class AdvancedFormTypesExampleComponent {
9   form: FormGroup;
10
11   constructor(private formBuilder: FormBuilder) {
12     this.form = this.formBuilder.group({
13       region: ['south-america']
14     });
15   }
16
17   submit() {
18     console.log(this.form.value);
19   }
20 }
```

On each input radio type, we use the `formControlName` with the same shared `FormControl` property. We set the `name` attribute to associate the radios as a single group to toggle. Lastly, we set the `value` attribute for Angular to set the `FormControl` value.

Angular smoothly works out of the box with all HTML input types. In a later chapter, we will cover how to create custom Angular components that can bind to form controls as custom inputs.

Chapter 6 - Accessibility with Angular Forms

Using Angular and accessibility (a11y) best practices we can create complex forms that any user can use. When building accessible HTML forms, we have several things we must consider. Some users have mobility issues and can only use the Web via keyboard or other non-mouse inputs. Other users may have vision disabilities that require the use of a screen reader or have difficulties with color blindness. With well-structured HTML and Angular, we can make our forms accessible for all users.

In this chapter, we will break down several critical components of making an Angular form accessible.

- HTML Form Basics
- Angular Reactive Forms
- Multiple Angular Form Groups
- Accessible Form Validation
- Accessible Inline Form Validation
- Angular Form Group Validation
- Accessible Form Validation Summary
- Accessible Form Status Messages

Many of these concepts will apply to making accessible forms in other tools like [React](#)⁴ [Vue](#)⁵.

HTML Form Basics

In our example, we will slowly build up to a sophisticated multi-step user sign up form by using accessibility best practices from the start. Our form has three groups of information for the user to fill out, a name, contact information and, account information.

⁴<https://reactjs.org/>

⁵<https://vuejs.org/>

Legal Name
First Name

Last Name

Contact Information
Email (required)

Phone Number

Account Information
Password (required)

Confirm Password (required)

Create Account

Accessible Forms with Angular

First, we will start with creating a basic HTML form with first name and last name with no Angular interactivity just yet.

```
1 <form>
2   <label for="first">First Name</label>
3   <input id="first" />
4
5   <label for="last">Last Name</label>
6   <input id="last" />
7
8   <button>Create Account</button>
9 </form>
```

The first step for accessible forms is to provide a label for each input in the form. Defining a label allows screen readers to read aloud the description of what the purpose of the input is. Two popular and easily available screen readers to test with are [Voiceover](https://www.apple.com/accessibility/mac/vision/)⁶ (Mac OS) and [Chrome Vox](https://www.chromevox.com/)⁷ (Chrome OS / Chrome).

We associate a label with the `for` attribute and match it to an `id` attribute of an existing input. Another benefit of assigning a label to an input is when the user clicks on a label the browser focuses the associated input.

Another critical aspect of building not only accessible forms but better user experiences is to have a clear call to action with your form submit button. Instead of generic messages such as “save” or “submit” be specific to what the action is accomplishing for example, “Create Account” or “Update Your Profile”.

Angular Reactive Forms

Now that we have the basics of our form defined we can start wiring up our form to Angular to make it interactive. In this example, we will continue to use the Angular Reactive Forms module.

First, we need to import the `ReactiveFormsModule` into our application module.

```
1 import { NgModule } from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3 import { ReactiveFormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6
7 @NgModule({
8   imports:      [BrowserModule, ReactiveFormsModule],
9   declarations: [AppComponent],
10  bootstrap:    [AppComponent]
```

⁶<https://www.apple.com/accessibility/mac/vision/>

⁷<https://www.chromevox.com/>

```
11  })
12  export class AppModule { }
```

Once imported we can now start using the Reactive Forms API to build our sign up form.

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormControl } from '@angular/forms';
3
4  @Component({
5    selector: 'my-app',
6    templateUrl: './app.component.html',
7    styleUrls: [ './app.component.css' ]
8  })
9  export class AppComponent {
10    form = new FormGroup({
11      name: new FormGroup({
12        first: new FormControl(''),
13        last: new FormControl('')
14      })
15    });
16
17    submit() {
18      console.log(this.form.value);
19    }
20  }
```

When creating our form, we use a combination of the `FormGroup` class and the `FormControl` class. `FormGroup` allows us to create a group or collection of related `FormControl` inputs. A `FormControl` represents a single form input in our HTML template. If we look, we see that our example uses two `FormGroup` instances.

```
1  form = new FormGroup({
2    name: new FormGroup({
3      first: new FormControl(''),
4      last: new FormControl('')
5    })
6  });
```

The first group is the form element itself and contains all inputs of the form. Our second form group is grouping the two name inputs, first and last. We group these inputs to make it easier to manage and validate as we add additional steps and groups such as contact information and password validation. We will see how to add additional groups in our example soon.

When we create a group each property of the group corresponds to a single input so our example first and last name. Each property is an instance of `FormControl`.

Next we need to assign our new Angular form to the form in our HTML template:

```
1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <fieldset formGroupName="name">
3     <legend>Legal Name</legend>
4     <label for="first">First Name</label>
5     <input formControlName="first" id="first" />
6
7     <label for="last">Last Name</label>
8     <input formControlName="last" id="last" />
9   </fieldset>
10
11   <button>Create Account</button>
12 </form>
```

The first `[formGroup]` binding on the HTML form tag assigns the `form` property of our component linking the Angular form to the HTML form.

In our form, we have added the `fieldset` and `legend` HTML tags. The `fieldset` and `legend` tags allow us to describe a collection of related inputs and a descriptive label. `Fieldset` is commonly used with collections of inputs such as radios and checkboxes. While not required, `fieldset` and `legend` are also important for large forms to make it clear what the inputs are doing and how they are interrelated. By adding a `fieldset` and `legend` we also further assist anyone using a screen reader to understand better what the form is trying to accomplish when visual cues such as white space are not available. On each `fieldset` we can attach our inner form group so Angular's form groups can associate to a given `fieldset`.

Lastly, we need to associate each input to the given `FormControl` in our TypeScript. Using `formControlName`, we can assign the control name to the given input which assigns any default value as well as applies the appropriate validation if any.

Multiple Angular Form Groups

Now our form is functional we can start to add additional form groups to our form. By having multiple groups, we can make it easier to validate our forms as well as improve the accessibility of our form. Our sign up form currently has just a first and last name, let's go ahead and add a new group called the contact group. The contact group contains the inputs for the user's email and phone number. Let's start with taking a look at the TypeScript.

```

1  import { Component } from '@angular/core';
2  import { FormGroup, FormControl, Validators, ValidationErrors } from '@angular/forms\
3  ';
4
5  @Component({
6    selector: 'my-app',
7    templateUrl: './app.component.html',
8    styleUrls: [ './app.component.css' ]
9  })
10 export class AppComponent {
11   form = new FormGroup({
12     name: new FormGroup({
13       first: new FormControl(''),
14       last: new FormControl('')
15     }),
16     contact: new FormGroup({
17       email: new FormControl('', [Validators.required, Validators.email]),
18       phone: new FormControl('')
19     })
20   });
21
22   submit() {
23     console.log(this.form.value);
24   }
25 }

```

We added a new property contact which contains a new FormGroup instance. This group contains the email and phone number FormControls for our next step in our sign up form. We could use the FormBuilder service to build the form but to emphasize better how the form is instantiated we will manually instantiate each input. Next, we need to create the HTML and wire up our new FormGroup.

```

1  <form [formGroup]="form" (ngSubmit)="submit()">
2    <fieldset formGroupName="name">
3      <legend>Legal Name</legend>
4      <label for="first">First Name</label>
5      <input formControlName="first" id="first" />
6
7      <label for="last">Last Name</label>
8      <input formControlName="last" id="last" />
9    </fieldset>
10
11   <fieldset formGroupName="contact">
12     <legend>Contact Information</legend>

```



```

13     <label for="email">Email (required)</label>
14     <input formControlName="email" id="email" type="email" />
15
16     <label for="phone">Phone Number</label>
17     <input formControlName="phone" id="phone" />
18 </fieldset>
19 <button>Create Account</button>
20 </form>

```

Now that we have our second group added we need to handle some validation logic in our template for the new required email input.

Accessible Form Validation

In our example form, we now have an email input that is required and must match an email address format. We have to convey this information clearly to our users as well as properly handle screen readers. Let's take a look at our updated template for the email input.

```

1 <legend>Contact Information</legend>
2 <label for="email">Email (required)</label>
3 <input formControlName="email" id="email" type="email" required />

```

Because the email input is required, there are a couple of additional things we need to do for user experience and accessibility. First, in our label for the email input, we add the word required. By adding required in the label makes it easier for users upfront to know they should add an email address versus getting to the end of the form and having to backtrack to previous inputs. By explicitly using the word “required” this also helps screen reader users in the same way.

Do **not** use patterns such as marking required fields with an "*" asterisk, when read by a screen reader does not convey to the user that the input is required. We can also add the required attribute to our input as this helps screen readers additionally understand the input that is required for the user to enter. I would love to see Angular be able to dynamically add the required attribute whenever the `Validators.required` is used on an Angular form to improve accessibility out of the box.

Note it is also essential to take advantage of HTML5 input types when appropriate. In our use case, we use `type="email"` on the email input. By using the proper input type, we can take advantage of native input behavior such as a better on-screen keyboard optimized for email inputs.

Accessible Inline Form Validation

Next, we need to add our validation message and set validation state for accessibility on our email input. Inline validation allows us to notify the user quickly that something needs to be corrected to prevent them from submitting the form and having backtrack to previous inputs.

Contact Information

Email (required)

Please enter a valid email address.

Phone Number

Accessible Inline Form Validation with Angular

Let's take a look at our updated template and TypeScript for determining if the input is invalid.

```
1 <label for="email">Email (required)</label>
2 <input
3   [attr.aria-invalid]="emailIsInvalidAndTouched"
4   FormControlName="email"
5   id="email"
6   type="email"
7   required />
```

If the input is invalid and the user has focused the input, then we show the appropriate message to the user below our email input. On our input, we set the `aria-invalid` attribute⁸ to true whenever the input is invalid as this helps notify screen readers that the input is invalid and needs the user's attention.

Unfortunately here due to the nesting behavior of the `FormGroup` the syntax to check the input validity is rather verbose, so we abstract it into a getter properties in our component so we don't duplicate the logic in our template.

⁸https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques/Using_the_aria-invalid_attribute

```
1  // Be careful when using get() properties as they are called multiple time per change detection cycle
2
3  // Example putting an async HTTP call in the get() would cause several HTTP requests\
4  in the matter of seconds hurting performance
5  get emailIsValid() {
6      return (this.form.controls.contact as FormGroup).controls.email.invalid;
7  }
8
9  get emailIsValidAndTouched() {
10     return this.emailIsValid && (this.form.controls.contact as FormGroup).controls.\
11     email.touched;
12 }
13
14 get passwordIsValid() {
15     return (this.form.controls.password as FormGroup).controls.password.invalid;
16 }
17
18 get passwordIsValidAndTouched() {
19     return this.passwordIsValid && (this.form.controls.password as FormGroup).contro\
20     ls.password.touched;
21 }
22
23 get passwordsDoNotMatch() {
24     return this.form.controls.password.errors.mismatch;
25 }
26
27 get passwordsDoNotMatchAndTouched() {
28     return this.passwordsDoNotMatch && (this.form.controls.password as FormGroup).cont\
29     rols.confirm.touched;
30 }
```

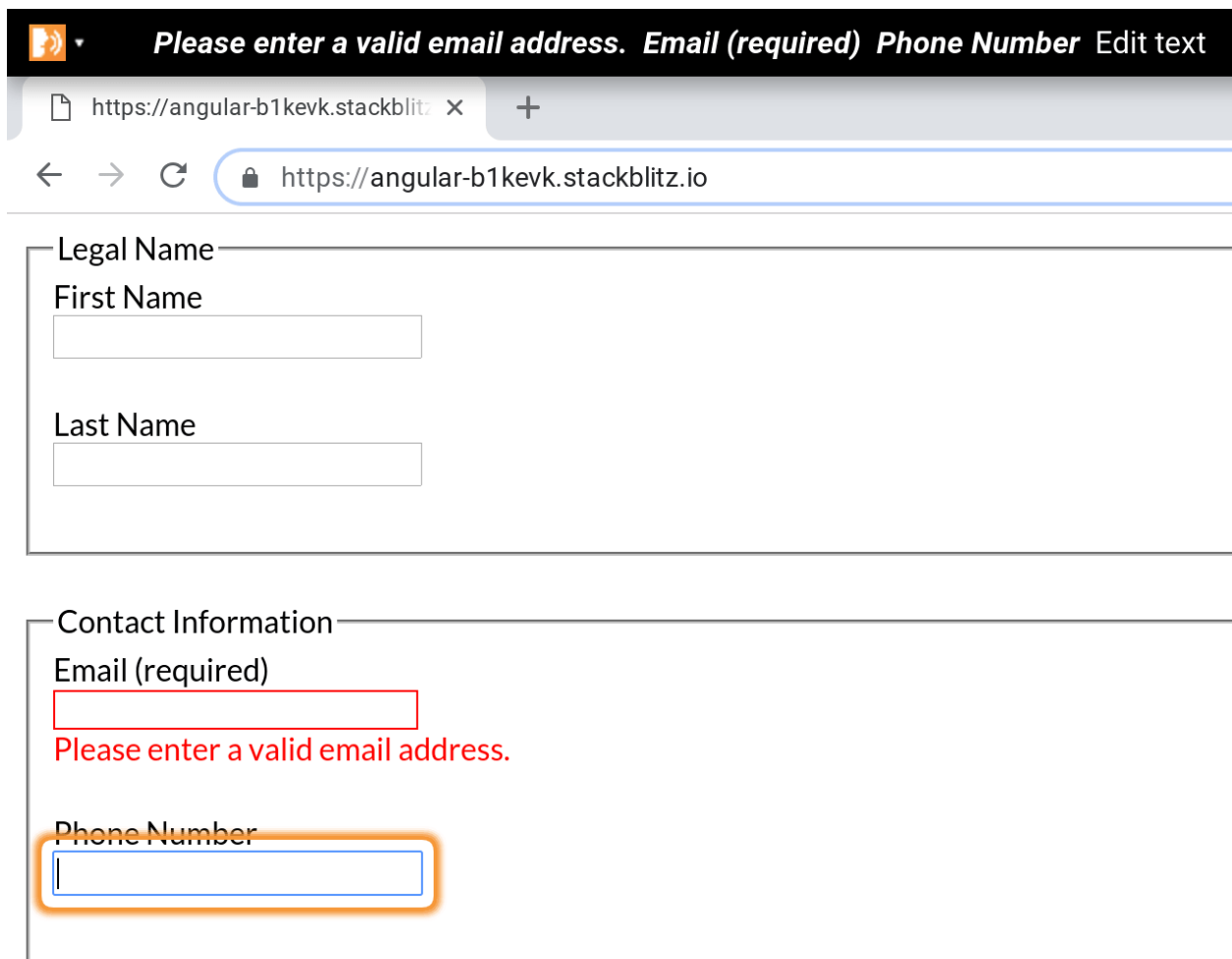
Now that we have the logic in place to determine if the email input is invalid and touched we can add our message to show the user when our email input is invalid. We will use the non touched checks later in our example.

```
1 <label for="email" id="email-label">Email (required)</label>
2 <input
3   [attr.aria-invalid]="emailIsInvalidAndTouched"
4   FormControlName="email"
5   id="email"
6   type="email"
7   required />
8
9 <div
10  *ngIf="emailIsInvalidAndTouched"
11  role="alert"
12  aria-describedby="email-label"
13  tabindex="0"
14  class="error">
15   Please enter a valid email address.
16 </div>
```

First, we need to add a new id attribute `id="email-label"` to our label. This id is used to associate our error message to the label. Next, we have our Error message which contains several attributes. First, we have our `*ngIf` which toggles the element in the DOM whenever the email is invalid. Next, we have the `role="alert"` attribute which triggers the screen reader to immediately read the content in the element to notify the user that an error has occurred.

The next attribute `aria-describedby="email-label"`⁹ allows the screen reader to notify that the given validation/alert message belongs or is associated with the email label. This association makes it easier to understand that the error is for the email input. Lastly, we have the inline validation message set to have a `tabindex="0"`. Setting the tab index makes the validation message focusable making it easier to discover as the user tabs through our form to fix any validation issues.

⁹<https://webaim.org/techniques/formvalidation/>



Please enter a valid email address. Email (required) Phone Number Edit text

https://angular-b1kevk.stackblitz.io

Legal Name

First Name

Last Name

Contact Information

Email (required)

Please enter a valid email address.

Phone Number

Accessible Inline Form Validation with Angular

Using [Chrome Vox](https://www.chromevox.com/)¹⁰ we can see that the screen reader successfully reads the inline validation error message and the associated label. Do not try to refocus the input with the validation error as it can be confusing to have navigation jumping without users direct input. Allow the user to tab back and refocus the input as necessary.

Angular Form Group Validation

The next step in our sign up form builds on more of what we have covered by adding a password and confirm password input.

¹⁰<https://www.chromevox.com/>

```
1 <fieldset formGroupName="password">
2   <legend>Account Information</legend>
3   <label for="password" id="password-label">Password (required)</label>
4   <input
5     [attr.aria-invalid]="passwordIsInvalidAndTouched"
6     formControlName="password"
7     id="password"
8     type="password"
9     required />
10  <div
11    *ngIf="passwordIsInvalidAndTouched"
12    role="alert"
13    aria-describedby="password-label"
14    tabindex="0"
15    class="error">
16    Password must be at least 6 characters.
17  </div>
18
19  <label for="confirm" id="confirm-label">Confirm Password (required)</label>
20  <input formControlName="confirm" id="confirm" type="password" required />
21
22  <div
23    *ngIf="passwordsDoNotMatchAndTouched"
24    role="alert"
25    aria-describedby="confirm-label"
26    tabindex="0"
27    class="error">
28    Passwords must match.
29  </div>
30 </fieldset>
```

The last step in the form is slightly more complex as it must compare two inputs, password and confirm password for validation. In the TypeScript, we can create a `FormGroup` validator that allows us to validate multiple `FormControls` instead of a single `FormControl`.

```

1  form = new FormGroup({
2    name: new FormGroup({
3      first: new FormControl(''),
4      last: new FormControl('')
5    }),
6    contact: new FormGroup({
7      email: new FormControl('', [Validators.required, Validators.email]),
8      phone: new FormControl('')
9    }),
10   password: new FormGroup({
11     password: new FormControl('', [Validators.required, Validators.minLength(6)]),
12     confirm: new FormControl('', Validators.required)
13   }, matchingInputsValidator('password', 'confirm', 'mismatch'))
14 });
15
16 get passwordsDoNotMatch() {
17   return this.form.controls.password.errors?.mismatch && (this.form.controls.password\
18   rd as FormGroup).controls.confirm.touched;
19 }
20
21 // Angular validators expect a function to be passed as the parameter so it can execu\
22 ute,
23 // here we create a function dynamically to return to the form for Angular
24 export function matchingInputsValidator(firstKey: string, secondKey: string, errorName\
25 me: string) {
26   return function (group: FormGroup): ValidationErrors | undefined {
27     if (group.controls[firstKey].value !== group.controls[secondKey].value) {
28       return {
29         [errorName]: true
30       };
31     }
32   };
33 }

```

Now that we have all three groups of our form created, it's essential to test that the tab flow makes sense to the user. Starting at the top of your page test to see if tabbing through the page results in a linear expected experience. Users that have mobility disabilities may only be able to navigate the page via keyboard commands so tab order is essential to test with your forms.

Accessible Form Validation Summary

Now that we have our form functional and the inline validation working we need to handle displaying a validation summary. A validation summary displays a list of errors whenever a user has submitted the form and may have skipped input fields that need to be corrected.

The form has the following errors that need to be corrected:

Please enter a valid email address.

Password must be at least 6 characters.

Passwords must match.

Accessible Inline Form Validation with Angular

To display our validation summary, we need to track when the user has submitted the form and check the validation of our inputs. Let's take a look at our updated submit method.

```
1 <div
2   *ngIf="this.form.invalid && formSubmitted"
3   role="alert"
4   class="status-error">
5   The form has the following errors that need to be corrected:
6
7   <div *ngIf="emailIsValid">
8     Please enter a valid email address.
9   </div>
10
11  <div *ngIf="passwordIsValid">
12    Password must be at least 6 characters.
13  </div>
14
15  <div (click)="confirmRef.focus()" *ngIf="passwordsDoNotMatch">
16    Passwords must match.
17  </div>
18 </div>
```


In our template, we show the summary only if the form has been submitted and there is at least one form validation error. This works well, but we can further improve the user experience and accessibility of the validation summary.

With each error message, we can create a click event to focus the input that needs attention. You could achieve this with in-page anchors with regular HTML, but with Angular and client-side routing we have to do something slightly more complicated.

We update each error message to a button with a click event. On each input that we want to navigate to; we add a special Angular template syntax called a `template reference variable`.

```

1  ...
2  <!-- The '#' creates a template reference variable we can use later in our template \
3  -->
4  <input
5      #emailRef
6      [attr.aria-invalid]="emailIsInvalidAndTouched"
7      formControlName="email"
8      id="email"
9      type="email"
10     required />
11
12  ...
13
14  <button (click)="emailRef.focus()" *ngIf="emailIsInvalid">
15     Please enter a valid email address.
16  </button>
17  ...

```

For each input, we want to focus explicitly, we will need to create a unique template reference variable for each corresponding input.

```

1  <div
2      *ngIf="this.form.invalid && formSubmitted"
3      role="alert"
4      class="status-error">
5      The form has the following errors that need to be corrected:
6
7      <button (click)="emailRef.focus()" *ngIf="emailIsInvalid">
8          Please enter a valid email address.
9      </button>
10
11     <button (click)="passwordRef.focus()" *ngIf="passwordIsInvalid">
12         Password must be at least 6 characters.

```

```
13     </button>
14
15     <button (click)="confirmRef.focus()" *ngIf="passwordsDoNotMatch">
16         Passwords must match.
17     </button>
18 </div>
```

Now for any user when the validation summary shows the user can quickly navigate to the input that needs to be fixed easily.

Accessible Form Status Messages

The last step for our accessible form is to provide status updates to the user when the form is processing data. We covered error statuses; next, we create a pending status and success status for when data is saved asynchronously.

Create Account

Saving profile

Profile successfully created

Account Information

Password (required)

Confirm Password (required)

Accessible Inline Form Validation with Angular

Here is an image showing both the pending and success status messages of our form. Notice the pending message the text is black instead of the matching orange border. We don't match the colors to make sure the text has enough color contrast, so it's clear and easily visible for all users. You can test your colors to see if they are accessible by using contrast-ratio.com¹¹.

To start creating our status messages, we are going to do a little bit of refactoring and create a TypeScript enum to represent the state of our form.

¹¹<https://contrast-ratio.com>

```
1  export enum FormStatus {
2    Initial,
3    Success,
4    Pending,
5    Error
6  }
7
8  ...
9
10 export class AppComponent {
11   // bind the enum to the component so we can reference it in our template
12   FormStatus = FormStatus;
13   // initialize the status
14   formStatus = FormStatus.Initial;
15   ...
16 }
```

Now we can update our submit method to emulate our async event similar to saving the form data to an API endpoint.

```
1  submit() {
2    this.formStatus = FormStatus.Pending;
3
4    if (this.form.valid) {
5      setTimeout(() => { // simulate a slow async http/api call
6        this.formStatus = FormStatus.Success;
7        console.log(this.form.value);
8      }, 3000);
9    } else {
10     this.formStatus = FormStatus.Error;
11   }
12 }
```

Now when our form is submitted, we can set the form to the pending state and show a pending message to the user. When our API call completes, we can then display a success message.

```
1 <div *ngIf="formStatus === FormStatus.Pending" role="status" class="status-pending">
2   Saving profile
3 </div>
4
5 <div *ngIf="formStatus === FormStatus.Success" role="status" class="status-success">
6   Profile successfully created
7 </div>
```

When displaying our status messages, we use the `role="status"` attribute. The main difference between `role="status"` and `role="alert"` is that `role="status"` will wait if the screen reader is paused before reading the status aloud. The `role="alert"` immediately causes the screen reader to read the alert even if the reader is in the middle of reading other content on the page. By using `role="status"` we can be more considerate of the user and not interrupt or stop what they are doing.

Getting accessibility right can be challenging but using Angular we can create dynamic client-side JavaScript forms while keeping them accessible to all users.

Chapter 7 - Async Form Data

In real-world Angular applications we commonly deal with asynchronous data usually from some kind of API. When dealing with forms, we often have to initialize the form input with existing data the user had entered at an earlier point in time. When doing this with asynchronous data, it can be a bit more complicated. This chapter we will cover how to handle async data for a good user experience as well as how to pre-populate a form with the data.

For our example, we have a simple user profile form that contains three inputs, a first name, last name and about input.

Forms with Async Data

First Name

Last Name

About

Example Form for pre-populating forms

This form will need to be pre-populated with data the user had saved earlier in their user account. First, let's take a look at building our form.

```
1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="firstname">First Name</label>
3   <input id="firstname" formControlName="firstName" />
4   <div *ngIf="firstName.hasError('required') && firstName.touched" class="error">
5     Required
6   </div>
7
8   <label for="lastname">Last Name</label>
9   <input id="lastname" formControlName="lastName" />
10  <div *ngIf="lastName.hasError('required') && lastName.touched" class="error">
```

```

11     Required
12 </div>
13
14 <label for="about">About</label>
15 <textarea id="about" formControlName="about"></textarea>
16
17 <button>Save Profile</button>
18 </form>

1 import { Component, OnInit } from '@angular/core';
2 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3
4 @Component({
5   selector: 'app-forms-with-async-data-example',
6   templateUrl: './forms-with-async-data-example.component.html'
7 })
8 export class FormsWithAsyncDataExampleComponent implements OnInit {
9   form: FormGroup;
10
11   get firstName() {
12     return this.form.controls.firstName;
13   }
14
15   get lastName() {
16     return this.form.controls.lastName;
17   }
18
19   constructor(private formBuilder: FormBuilder) { }
20
21   ngOnInit() {
22     this.form = this.formBuilder.group({
23       firstName: ['', Validators.required],
24       lastName: ['', Validators.required],
25       about: []
26     });
27   }
28
29   submit() { ... }
30 }

```

Our form is pretty straightforward; we use the `FormBuilder` to initialize and create the form. For this use case, we need to load in some asynchronous data and populate the form with that data. To do this, we have a `UserService` that will fetch or simulate fetching our data from an API.


```
1  import { Injectable } from '@angular/core';
2  import { of } from 'rxjs';
3  import { delay } from 'rxjs/operators';
4
5  export interface User {
6    id: number;
7    firstName: string;
8    lastName: string;
9    about: string;
10 }
11
12 const fakeData: User = {
13   id: 0,
14   firstName: 'Cory',
15   lastName: 'Rylan',
16   about: 'Web Developer'
17 };
18
19 @Injectable()
20 export class UserService {
21   constructor() { }
22
23   loadUser() {
24     // A real HTTP request will use the HttpClientService, https://angular.io/guide/\
25     http for more details
26     // Here we use RxJS to simulate an async response
27     return of<User>(fakeData).pipe(
28       delay(2000)
29     );
30   }
31 }
```

The `UserService` has a single method `loadUser()` which returns an `Observable` of user data. `Observables` are widely used throughout Angular to manage and handle async data. Typically we would use the Angular HTTP Client Service to fetch this data from an API. Angular uses an `Observable` library called `RxJS`. We use `RxJS` simulate an async operation with a particular function called an operator to manipulate the data.

The `delay` operator delays the `Observable` to emit the async value after a specific time span. For this example that is a delay of two seconds. We delay this to simulate a slow connection, and this emphasizes how it is essential to handle this slow async data properly.

In the component, we can now inject our `UserService` using Angular's dependency injection.

```
1  import { Component, OnInit } from '@angular/core';
2  import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3
4  import { UserService, User } from '../user.service';
5
6  @Component({
7    selector: 'app-forms-with-async-data-example',
8    templateUrl: './forms-with-async-data-example.component.html'
9  })
10 export class FormsWithAsyncDataExampleComponent implements OnInit {
11   form: FormGroup;
12   user: Observable<User>;
13
14   constructor(
15     private formBuilder: FormBuilder,
16     private userService: UserService) { }
17
18   ngOnInit() {
19     this.form = this.formBuilder.group({
20       firstName: ['', Validators.required],
21       lastName: ['', Validators.required],
22       about: []
23     });
24
25     this.userService.loadUser().subscribe(userData => {
26       console.log(userData);
27     });
28   }
29
30   submit() { ... }
31 }
```

With Observables, we can subscribe to value updates and receive those update in a return function. While this works, Angular provides a nice syntax to subscribe for us in our template.

```
1  import { Component, OnInit } from '@angular/core';
2  import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3  import { Observable } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5
6  import { UserService, User } from '../user.service';
7
8  @Component({
9    selector: 'app-forms-with-async-data-example',
10    templateUrl: '../forms-with-async-data-example.component.html'
11  })
12  export class FormsWithAsyncDataExampleComponent implements OnInit {
13    form: FormGroup;
14    user: Observable<User>;
15
16    constructor(
17      private formBuilder: FormBuilder,
18      private userService: UserService) { }
19
20    ngOnInit() {
21      this.form = this.formBuilder.group({
22        firstName: ['', Validators.required],
23        lastName: ['', Validators.required],
24        about: []
25      });
26
27      this.user = this.userService.loadUser();
28    }
29
30    submit() { ... }
31  }
```

Instead of subscribing to the data updates we assign the Observable as a property of our class. In our template, we use a special pipe syntax that subscribes to the data and makes it available to our template.

```

1 <form *ngIf="user | async; let user;" [formGroup]="form" (ngSubmit)="submit()">
2   <label for="firstname">First Name</label>
3   <input id="firstname" formControlName="firstName" />
4   <div *ngIf="firstName.hasError('required') && firstName.touched" class="error">
5     Required
6   </div>
7
8   <label for="lastname">Last Name</label>
9   <input id="lastname" formControlName="lastName" />
10  <div *ngIf="lastName.hasError('required') && lastName.touched" class="error">
11    Required
12  </div>
13
14  <label for="about">About</label>
15  <textarea id="about" formControlName="about"></textarea>
16
17  <button>Save Profile</button>
18
19  <label>Async Data:</label>
20  <!-- show data in json format -->
21  <pre>{{user | json}}</pre>
22 </form>

```

We subscribe to the user observable with the async pipe (`user | async`). Angular subscribes to the Observable and re-renders the template if the Observable emits any new event values.

We do this in the `*ngIf` so we can hide the form from the user until the data is loaded. We don't want to show the form early where the user could enter data and then replace any data they had entered with the async data. By hiding it, we ensure the user won't lose data the might of tried to enter.

The `let user;` statement in our `*ngIf` takes whatever value the observable emits and assign it as a local variable in our template to be able to display as we need.

Because we hide the form from the user until the data is loaded we want to provide some feedback for the user to let them know the data is loading. This feedback can be in the form of a message or a spinner of some kind. Angular provides an additional `else` syntax to the existing `*ngFor` syntax to accomplish this.

```

1 <form *ngIf="user | async; let user; else loading" [formGroup]="form" (ngSubmit)="su\
2 bmit()">
3   ...
4 </form>
5
6 <ng-template #loading>
7   Loading User Data...
8 </ng-template>

```

In the template, we can define an `ng-template` element. This creates a subset template that won't render unless specifically directed to. In the `*ngIf` statement, we can add an additional `else`. We can give the template a special unique identifier called a template ref. In this example, the template ref is `#loading`, the `#` denotes the template ref followed by the name.

```

1 ngIf="user | async; let user; else loading"

```

If there is a user loaded render the form, else show the `loading` template. Now that we show a loading message as the data is being loaded and swap it for the form when the data has loaded we need to populate the form with the loaded data.

Form SetValue and PatchValue

To populate the form, we can use a combination of RxJS Observable operators and the Form API to dynamically update the form values.

```

1 import { tap } from 'rxjs/operators';
2
3 ...
4
5 ngOnInit() {
6   this.form = this.formBuilder.group({
7     firstName: ['', Validators.required],
8     lastName: ['', Validators.required],
9     about: []
10  });
11
12   this.user = this.userService.loadUser().pipe(
13     tap(user => this.form.patchValue(user))
14   );
15 }

```

Because we allow Angular to subscribe to the Observable for us with the async pipe, we can't subscribe in the TypeScript. When using RxJS Observables, we can chain one to many operator functions in the pipe() method of our Observable. We will cover Observables more in-depth in a later chapter.

We can use the tap operator function for our use case. The tap operator allows us to “tap” into the stream of events. Every time the Observable emits a value the value runs through our tap operator.

The tap operator allows us to handle side effects or changes that do not directly affect the value of the Observable. In our use case, our side effect is setting the form value but not changing the value in the Observable. Tap passes the original value unaltered onto the Observable.

Now that we can tap into the value of our async user data we can update the form to have an initial value. To update the form we can use two different methods.

```
1 // update all controls
2 this.form.setValue({ firstName: 'Cory', lastName: 'Rylan', about: 'Web Developer' });
3
4 // update subset of form controls
5 this.form.patchValue({ firstName: 'Cory' });
```

The setValue requires us to pass an object with a key value for each corresponding control name on our form. If we forget one of the control names, it throws an error. The patchValue method, however, allows us to set just a subset of controls on our form. So our final code example looks like the following.

```
1 import { Component, OnInit } from '@angular/core';
2 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { tap } from 'rxjs/operators';
5
6 import { UserService, User } from '../user.service';
7
8 @Component({
9   selector: 'app-forms-with-async-data-example',
10   templateUrl: '../forms-with-async-data-example.component.html'
11 })
12 export class FormsWithAsyncDataExampleComponent implements OnInit {
13   form: FormGroup;
14   user: Observable<User>;
15
16   constructor(
17     private formBuilder: FormBuilder,
18     private userService: UserService) { }
```

```

19
20   ngOnInit() {
21     this.form = this.formBuilder.group({
22       firstName: ['', Validators.required],
23       lastName: ['', Validators.required],
24       about: []
25     });
26
27     this.user = this.userService.loadUser().pipe(
28       tap(user => this.form.patchValue(user))
29     );
30   }
31
32   submit() {
33     if (this.form.valid) {
34       console.log(this.form.value);
35     }
36   }
37 }

```



```

1  <form *ngIf="user | async; else loading" [formGroup]="form" (ngSubmit)="submit()">
2    <label for="firstname">First Name</label>
3    <input id="firstname" formControlName="firstName" />
4    <div *ngIf="firstName.hasError('required') && firstName.touched" class="error">
5      Required
6    </div>
7
8    <label for="lastname">Last Name</label>
9    <input id="lastname" formControlName="lastName" />
10   <div *ngIf="lastName.hasError('required') && lastName.touched" class="error">
11     Required
12   </div>
13
14   <label for="about">About</label>
15   <textarea id="about" formControlName="about"></textarea>
16
17   <button>Save Profile</button>
18 </form>
19
20 <ng-template #loading>
21   Loading User Data...
22 </ng-template>

```

Chapter 8 - Dynamic Form Controls

In the previous chapter, we covered how to handle and deal with asynchronous data in Angular forms. This chapter we will cover how to use async data to build form controls dynamically such as selects and radio lists.

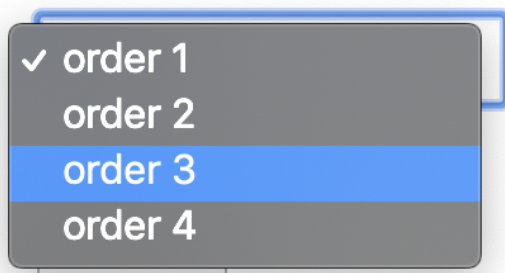
Dynamic Select Input

When creating a select dropdown in HTML, we have several tags we must define in the template.

```
1 <select>
2   <option value="1">Order 1</option>
3   <option value="2">Order 2</option>
4   <option value="3">Order 3</option>
5   <option value="4">Order 4</option>
6 </select>
```

Dynamic Form Controls Example

Select Order



Example HTML Select Input

Often when creating inputs like a select, we need to create the options based on dynamic data such as a list from an API. This example we load some data asynchronously and dynamically build the select input.

We have an `OrderService` that returns our test order data and make the data async by using an `Observable` and `RxJS` operators.


```

1  import { Injectable } from '@angular/core';
2  import { of } from 'rxjs';
3  import { delay } from 'rxjs/operators';
4
5  export interface Order {
6    id: number;
7    name: string;
8  }
9
10 const fakeData: Order[] = [
11   { id: 100, name: 'order 1' },
12   { id: 200, name: 'order 2' },
13   { id: 300, name: 'order 3' },
14   { id: 400, name: 'order 4' }
15 ];
16
17 @Injectable({
18   providedIn: 'root'
19 })
20 export class OrderService {
21   loadOrders() {
22     // A real HTTP request will use the HTTPClientService, https://angular.io/guide/\
23     http for more details
24     // Here we use RxJS to simulate an async response
25     return of<Order[]>(fakeData).pipe(
26       delay(1000)
27     );
28   }
29 }

```

In the Angular component, we use the FormBuilder service to instantiate our form.

```

1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder } from '@angular/forms';
3  import { Observable } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5
6  import { OrderService, Order } from './order.service';
7
8  @Component({
9    selector: 'app-dynamic-form-controls-example',
10    templateUrl: './dynamic-form-controls-example.component.html'
11  })

```

```

12 export class DynamicFormControlsExampleComponent {
13   form: FormGroup;
14   orders: Observable<Order[]>;
15
16   constructor(private FormBuilder: FormBuilder, private orderService: OrderService) {
17     this.form = this.formBuilder.group({
18       selectOrder: []
19     });
20
21     this.orders = this.orderService.loadOrders().pipe(
22       tap(orders => this.form.patchValue({ selectOrder: orders[0].id }))
23     );
24   }
25
26   submit() { ... }
27 }

```

Like our previous example, we bind the Observable data from the Service to the component and use the async pipe to subscribe to it. We use the tap operator to access the value and set the forms initial value to be the first id value in the orders list.

In the HTML template, we use the `ngIf` to show the data when loaded and the async pipe to subscribe to the data.

```

1 <form *ngIf="orders | async; let orders; else loading" [formGroup]="form" (ngSubmit)\
2 ="submit()">
3   <!-- dynamic select will be here -->
4
5   <button>submit</button>
6 </form>
7 <ng-template #loading>
8   Loading Orders...
9 </ng-template>

```

In the `*ngIf` we subscribe with the async pipe and assign the value to the orders variable using the `let` syntax. If there is no value, then we use the `else` condition to show the loading template.

Now that the basic structure of the form created we can now build our dynamic select input.

```
1 <form *ngIf="orders | async; let orders; else loading" [formGroup]="form" (ngSubmit)\  
2 ="submit()">  
3  
4   <label for="orders">Select Order</label>  
5   <select formControlName="selectOrder" id="orders">  
6     <option *ngFor="let order of orders; let i = index" [value]="orders[i].id">  
7       {{orders[i].name}}  
8     </option>  
9   </select>  
10  
11   <button>submit</button>  
12 </form>
```

Select inputs just like regular text inputs are associated with a form control using the `formControlName` directive. In the select we simple use a `*ngFor` loop to create a single `<option>` tag for each of the orders. The inner text of each option displays the order name. The value of the option is set by binding to the `[value]` property of the option and passing it the order id. Just like any other form control, we can get the value back from the form object.

```
1 submit() {  
2   // with selects and radios the value is always returned as a string type  
3   console.log('select order: ', parseInt(this.form.value.selectOrder, 10));  
4 }
```

An important note when using a select in HTML the value always returns as a string. In this use case, we set the value from a numeric ID so we must convert that back to a number when getting the form value.

Dynamic Radio Lists

Radio lists function very similarly to our previous select example. We use the same component and list of async order data to create a radio list dynamically.

Dynamic Form Controls Example

Radio Orders

☒ order 1

☐ order 2

☐ order 3

☐ order 4

submit

Example HTML Radio Input

```
1 import { Component } from '@angular/core';
2 import { FormGroup, FormBuilder } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { tap } from 'rxjs/operators';
5
6 import { OrderService, Order } from './order.service';
7
8 @Component({
9   selector: 'app-dynamic-form-controls-example',
10  templateUrl: './dynamic-form-controls-example.component.html'
11 })
12 export class DynamicFormControlsExampleComponent {
13   form: FormGroup;
14   orders: Observable<Order[]>;
15
16   constructor(private formBuilder: FormBuilder, private orderService: OrderService) {
17     this.form = this.formBuilder.group({
18       radioOrder: []
```

```

19     });
20
21     this.orders = this.orderService.loadOrders().pipe(
22       tap(orders => this.form.patchValue({ radioOrder: orders[0].id }))
23     );
24   }
25
26   submit() { ... }
27 }

```

In our template, we see that the radio is not too different than a select.

```

1 <span class="label">Radio Orders</span>
2 <label *ngFor="let order of orders; let i = index">
3   <input type="radio" formControlName="radioOrder" name="radioOrder" [value]="orders\
4 [i].id" />
5   {{orders[i].name}}
6 </label>

```

When creating our radio, we loop over each order creating an input with type="radio". Each radio is bound to the same formControlName. We set the value of each radio to the order id of that iteration. To associate the radios into a group HTML requires a name attribute on each radio of the group. This example the name id radioOrder, every radio with this name is associated with the same group.

Finally just like all of our other form controls we can access the value in our submit event.

```

1 import { Component } from '@angular/core';
2 import { FormGroup, FormBuilder } from '@angular/forms';
3 import { Observable } from 'rxjs';
4 import { tap } from 'rxjs/operators';
5
6 import { OrderService, Order } from './order.service';
7
8 @Component({
9   selector: 'app-dynamic-form-controls-example',
10  templateUrl: './dynamic-form-controls-example.component.html'
11 })
12 export class DynamicFormControlsExampleComponent {
13   form: FormGroup;
14   orders: Observable<Order[]>;
15
16   constructor(private formBuilder: FormBuilder, private orderService: OrderService) {
17     this.form = this.formBuilder.group({

```

```
18     radioOrder: []
19   });
20
21   this.orders = this.orderService.loadOrders().pipe(
22     tap(orders => this.form.patchValue({ radioOrder: orders[0].id }))
23   );
24 }
25
26 submit() {
27   // with selects and radios the value is always returned as a string type
28   console.log('radio order: ', parseInt(this.form.value.radioOrder, 10));
29 }
30 }
```

Radio inputs like selects return a string type as attributes like the value attribute are restricted to strings in HTML.

Chapter 9 - Dynamic Form Controls with Form Array

The previous chapter showed how to build individual dynamic controls dynamically such as a select or radio list. Sometimes we need to dynamically add and remove entire form controls from the form. Angular forms provide a friendly API that allows us to add or remove controls as easily as it is to add or remove items from an array.

In our example, we want to create a checkbox list based on a list of orders we fetch from an API.

Form Array Example

Checkbox List Orders

☒ order 1

☐ order 2

☐ order 3

☐ order 4

submit

Dynamic Checkbox List

To create the checkbox list, we use the `FormBuilder` service.

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormArray, FormControl, FormBuilder } from '@angular/forms';
3  import { Observable } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5
6  import { minSelectedCheckboxes } from './validators';
7  import { Order, OrderService } from './order.service';
8
9  @Component({
10     selector: 'app-form-array-example',
11     templateUrl: './form-array-example.component.html'
12 })
13 export class FormArrayExampleComponent {
14     form: FormGroup;
15     orders: Observable<Order[]>;
16
17     get orderControls() {
18         return (this.form.controls.orders as FormArray).controls;
19     }
20
21     constructor(private formBuilder: FormBuilder, private orderService: OrderService) {
22         this.form = this.formBuilder.group({
23             orders: new FormArray([])
24         });
25
26         this.orders = this.orderService.loadOrders();
27     }
28 }
```

When creating the form instead of the `orders` property being a standard form control, we use the `FormArray` class provided by Angular. The `FormArray` class allows us to control a list of form controls easily. With the `FormArray` we can add and remove `FormControl` instances from this list.

Just like in our previous examples with dynamic forms we can use RxJS to “tap” into the values and allow Angular to subscribe to the `Observable` via the `async` pipe in the component template.


```

1  constructor(private FormBuilder: FormBuilder, private orderService: OrderService) {
2    this.form = this.formBuilder.group({
3      orders: new FormArray([], minSelectedCheckboxes(1))
4    });
5
6    this.orders = this.orderService.loadOrders()
7      .pipe(tap(orders => this.addCheckboxes(orders)));
8  }
9
10 private addCheckboxes(orders: Order[]) {
11   orders.map((_, i) => {
12     const control = new FormControl(i === 0); // if first item set to true, else false
13   })
14   se (this.form.controls.orders as FormArray).push(control);
15   });
16 }

```

In the `addCheckboxes` method we loop through each order and for each order push a new instance of a form control. Each control initializes with a default value of false except the first one we set to true so the first checkbox in the UI is checked.

```

1  <form *ngIf="orders | async; let orders; else loading" [formGroup]="form" (ngSubmit)\
2  ="submit(orders)" class="checkbox-list">
3    <span class="label">Checkbox List Orders</span>
4    <label formArrayName="orders" *ngFor="let order of orderControls; let i = index">
5      <input type="checkbox" [formControlName]="i">
6        {{orders[i].name}}
7    </label>
8    <br>
9
10   <button [disabled]="!form.valid">submit</button>
11 </form>
12
13 <ng-template #loading>
14   Loading Orders...
15 </ng-template>

```

In the template, we can dynamically create each checkbox by looping through the list of form controls. Using the index of the loop we can assign the index value as the `formControlName` for each checkbox. When we submit the form, we get back the same list of checkbox values.

```
1 submit(orders: Order[]) {
2     const selectedOrderIds = this.form.value.orders
3     // for each checkbox boolean if its truthy return the id of the corresponding or\
4     der
5     .map((o, i) => o ? orders[i].id : null)
6     // filter out any falsy/null values from the map so we only return the selected \
7     ids
8     .filter(v => v !== null);
9
10    console.log('checkbox list orders: ', selectedOrderIds);
11    // if first and last items are selected
12    // selectedOrderIds will return [100, 400]
13 }
```

When submitting the form, we pass in the reference of orders to the submit method. We need this list because the list of form control checkboxes returns a list of booleans. We have to do a little extra work to map the list of booleans to know the ID of each order that was selected.

Validation

To extend the problem use case further, we have a business requirement that the user must check at least one order on the checkbox list to submit the form. We can solve this with what we have learned about form arrays as well as custom validation covered in earlier chapters.

Form Array Example

Checkbox List Orders

☒ order 1

☐ order 2

☐ order 3

☐ order 4

At least one order must be selected

submit

Dynamic Checkbox List with Validation

When constructing the form, we can add a new validator to the form array.

```
1 this.form = this.formBuilder.group({  
2   orders: new FormArray([], minSelectedCheckboxes(1))  
3 });
```

The second parameter of a `FormArray` is similar to a `FormControl` in that it can take one to many validator functions. This particular validator function `minSelectedCheckboxes` is a custom validator function we have created.

```
1  import { FormArray, ValidatorFn } from '@angular/forms';
2
3  export function minSelectedCheckboxes(min = 1) {
4    const validator: ValidatorFn = (formArray: FormArray) => {
5      const totalSelected = formArray.controls
6        // return a list of control values (booleans)
7        .map(control => control.value)
8        // loop through and total up the number of values that are true
9        .reduce((prev, next) => next ? prev + next : prev, 0);
10
11      // if the number of total selected is greater than the min then the validator passes
12      return totalSelected >= min ? null : { required: true };
13    };
14
15    return validator;
16  }
```

The validator function receives a reference to the `FormArray` when called. With the form array, we can loop through each from control of each checkbox and check if the value is truthy. If there is at least one true value in the list then we know at least one checkbox was selected. If not, we can return our custom required error.

Chapter 10 - Async Form Validation

Sometimes when validating an input, we need to check if the input is valid based on some condition on a backend server. Some examples include checking to see if a postal code or address is valid, or if a username has been taken. Fortunately for us, Angular forms have a built-in async validation API that makes this kind of check trivial.

In our example, we are going to create a user sign up form. This form will have an input that takes a username and will run an async API check to see if the username has already been taken.

Forms with Async Validation

Username

Checking username...

Async Validation Loading State

Forms with Async Validation

Username

Username taken

Async Validation Error State

First, let's take a look at our UsernameService to understand how we are getting the async data.

```
1 import { Injectable } from '@angular/core';
2 import { of } from 'rxjs';
3 import { delay } from 'rxjs/operators';
4
5 @Injectable()
6 export class UsernameService {
7   usernames = [
8     'luke',
9     'leia',
10    'han'
11  ];
12
13  usernameAvailable(username: string) {
14    // A real HTTP request will use the HTTPClientService, https://angular.io/guide/\
15    http for more details
16    // fake async task as if we made a http request to check the server if username \
17    has been taken
18    const match = this.usernames.find(n => n === username);
19    return of(match === undefined).pipe(delay(1000));
20  }
21 }
```

Like our previous service example, we are mocking out a fake async response and returning an

Observable of data to our component. This service simulates a check that would occur on the server. Here it is checking if the given username exists in the database and return if it is available or not. Next, let's take a look at creating our form instance in the component TypeScript.

```

1  import { Component, OnInit } from '@angular/core';
2  import { FormBuilder, FormGroup, Validators, ValidationErrors, FormControl } from '@\
3  angular/forms';
4  import { Observable } from 'rxjs';
5  import { map, tap } from 'rxjs/operators';
6
7  import { UsernameService } from './username.service';
8
9  @Component({
10   selector: 'app-async-validation-example',
11   templateUrl: './async-validation-example.component.html'
12 })
13 export class AsyncValidationExampleComponent implements OnInit {
14   form: FormGroup;
15
16   get userName() {
17     return this.form.controls.userName;
18   }
19
20   constructor(
21     private formBuilder: FormBuilder,
22     private usernameService: UsernameService) { }
23
24   ngOnInit() {
25     this.form = this.formBuilder.group({
26       userName: ['', Validators.required, usernameAvailableValidator(this.usernameSe\
27 rvice)]
28     });
29   }
30
31   submit() { ... }
32 }

```

Looking at the component and form instance, we can see that the custom async validator is the same API as a standard validator. This API makes it simple to use async validators with standard synchronous validators.

Notice how we inject the UsernameService and pass a reference to the usernameAvailableValidator function. Custom validators do not have dependency injection like components and services do.

Because of this, we must pass in a reference of the service as a parameter to the validator so it can use it to run the appropriate checks.

The async validator API looks very similar to a standard custom validator.

```

1 export function usernameAvailableValidator(usernameService: UsernameService) {
2   return (control: FormControl): Observable<ValidationErrors | null> => {
3     return usernameService.usernameAvailable(control.value).pipe(
4       map(usernameAvailable => usernameAvailable ? null : { username: 'username take\
5 n' })
6     );
7   };
8 }

```

The async validator API expects that instead of returning an error message or null for success cases you return an Observable or a Promise that returns the error value or null success value.

```

1 return usernameService.usernameAvailable(control.value).pipe(
2   map(usernameAvailable => usernameAvailable ? null : { username: 'username taken' })
3 );

```

Here in this example, we make an API request using the UsernameService and then using the RxJS operators we can map over the value. When we map the value, we essentially get to intercept the value before it gets to the subscriber and return a new value instead. Here we check if the user name is available to return null else return the username taken error code.

Now that the validator is set up we can show the validation messages in the template just like any other error message.

```

1 <form [formGroup]="form" (ngSubmit)="submit()">
2   <label for="userName">Username</label>
3   <input id="userName" formControlName="userName" />
4
5   <div *ngIf="userName.hasError('required') && userName.touched" class="error">
6     Required
7   </div>
8
9   <div *ngIf="userName.hasError('username') && userName.touched" class="error">
10    Username taken
11  </div>
12
13  <div *ngIf="userName?.pending" class="info">
14    Checking username...

```



```
15     </div>
16
17     <button>Save</button>
18 </form>
```

When using async validation, we get an additional property we can check on the `FormControl`. The `pending` property is true if the control is currently being validated. This helps to show the user some in progress message as async checks can take a noticeable amount of time.

Chapter 11 - Template Form API

So far throughout the book, we have been using the `ReactiveFormsModule` to build our Angular forms. Angular has two forms modules, the other we have not covered is the template-based forms called `FormsModule`.

Experience with large scale Angular applications the community recommends using the `ReactiveFormsModule` as it encourages more logic of our form to exist in the TypeScript. The template-based forms is similar to the older Angular JS 1.x forms where most of the logic to create a form is in the template.

This chapter we will briefly cover template driven forms although we don't recommend it, it is good to be familiar with the template form API. Understanding how template forms work will be helpful when you build your custom controls which we will cover in the next chapter.

In this example, we have a simple search input.

Template Forms

Search

Search Value: angular

Simple Search form with Template Form API

To start using the Template Forms API, we need to add the Angular Module to our application.

```
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { FormsModule } from '@angular/forms';
4
5  import { TemplateFormsExampleComponent } from './template-forms-example.component';
6
7  @NgModule({
8    declarations: [TemplateFormsExampleComponent],
9    imports: [
10      CommonModule,
11      FormsModule
12    ]
13  })
14  export class AppModule { }
```

In the TypeScript of the component, notice how little code there is.

```
1  import { Component } from '@angular/core';
2  import { NgForm } from '@angular/forms';
3
4  @Component({
5    selector: 'app-template-forms-example',
6    templateUrl: './template-forms-example.component.html'
7  })
8  export class TemplateFormsExampleComponent {
9    search = 'angular';
10
11    submit(form: NgForm) {
12      console.log(form.value);
13    }
14  }
```

Where most of the code lives for the form is in the template.

```

1 <h2>Template Forms</h2>
2
3 <form #exampleForm="ngForm" (ngSubmit)="submit(exampleForm)">
4   <label for="search">Search</label>
5   <input [(ngModel)]="search" name="search" type="search" required #searchControl="n\
6 gModel" />
7
8   <div *ngIf="searchControl.touched && searchControl.hasError('required')" class="er\
9 ror">
10     search is required
11   </div>
12
13   <button>Submit</button>
14 </form>
15
16 <p>Search Value: {{search}}</p>

```

In the template, we have a basic form element. To instantiate the form for template forms, we must use the # template variable syntax to create a variable reference we can use in the template. We bind it to the ngForm instance which connects the form to Angular's template form API. The form element uses the same standard (ngSubmit) event like we saw on the Reactive Forms.

```

1 <input [(ngModel)]="search" name="search" type="search" required #searchControl="ngM\
2 odel" />

```

The input uses a unique two-way binding syntax. This syntax often called “bananas in a box” essentially combines setting a property [prop] and listening to an event (event). This syntax combined with the ngModel directive allows our input to update the search property of the component automatically. This binding is convenient for smaller models but large data models Reactive Forms lend better as they don't mutate the model at all but return a new object when the form is submitted.

We can get a ngModel reference on the search input by using another template variable #searchControl="ngModel". We have a single validator required on the search input. When using template forms validation is initialized by placing specific Angular attribute directives on the input instead of providing a validator function. There are many built-in validation directives, and you can create your own custom validation directives as well.

```

1 <form #exampleForm="ngForm" (ngSubmit)="submit(exampleForm)">

```

When submitting the form, we can pass in a reference of the form to the submit method.

```
1 submit(form: NgForm) {  
2   console.log(form.value);  
3 }
```

Once we have a reference of the form in our TypeScript, we can do the various checks to see if the form is valid and to get the form values just like a form build with the `FormBuilder` API.

In the next chapter, we will take what we have learned and build a custom form control that will work with both the Template Forms API and the Reactive Forms API.

Chapter 12 - Custom Form Controls

Custom form controls/inputs are a typical pattern in complex Angular applications. It's common to want to encapsulate HTML, CSS, and accessibility in an input component to make it easier to use in forms throughout the application. Common examples of this are datepickers, switches, dropdowns, and typeaheads. All of these types of inputs are not native to HTML. Ideally, we would like them to integrate into Angular's form system easily.

In this chapter, we will show how to create a switch component (`app-switch`) which is essentially a checkbox with additional CSS and markup to get a physical switch like effect. This component will easily integrate into the new Angular Reactive and Template Form APIs. First, let's take a look at what our switch component will look like.

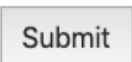
Custom Form Controls

NgModel



Value: false

Reactive Forms



Custom Toggle Control

The switch component mostly mimics the behavior of a checkbox. It toggles a boolean value in our forms. In this component, we use a native checkbox and some HTML and CSS to create the switch effect. We use a particular API Angular exposes to allow us to support both Template and Reactive Form API integration. Before diving into how to build the switch component let's take a look at what it looks like when using it in our Angular application.

Custom Form Controls with Reactive Forms

Let's take a quick look at how a Reactive Form would look with our custom app-switch component.

```

1 <h3>Reactive Forms</h3>
2 <form [formGroup]="myForm" (ngSubmit)="submit()">
3   <label for="switch-2">Switch 2</label>
4   <app-switch formControlName="mySwitch" id="switch-2"></app-switch>
5   <button>Submit</button>
6 </form>

1 import { Component, OnInit } from '@angular/core';
2 import { FormBuilder, FormGroup } from '@angular/forms';
3
4 @Component({
5   selector: 'app-custom-form-controls-example',
6   templateUrl: './custom-form-controls-example.component.html'
7 })
8 export class CustomFormControlsExampleComponent implements OnInit {
9   myForm: FormGroup
10
11   constructor(private formBuilder: FormBuilder) { }
12
13   ngOnInit() {
14     this.myForm = this.formBuilder.group({
15       mySwitch: [true]
16     });
17   }
18
19   submit() {
20     console.log(`Value: ${this.myForm.controls.mySwitch.value}`);
21   }
22 }

```

We can see our custom app-switch works seamlessly with the Reactive Forms/Form Builder API just like any other text input.

Custom Form Controls with Template Forms and NgModel

NgModel allows us to bind to an input with a two-way data binding syntax similar to Angular 1.x. We can use this same syntax when using a custom form control.

```

1 <h3>NgModel</h3>
2 <label for="switch-1">Switch 1</label>
3 <app-switch [(ngModel)]="value" id="switch-1"></app-switch><br />
4 <strong>Value:</strong> {{value}}

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-custom-form-controls-example',
5   templateUrl: './custom-form-controls-example.component.html'
6 })
7 export class CustomFormControlsExampleComponent implements OnInit {
8   value = false;
9
10  submit() {
11    console.log(`Value: ${this.value}`);
12  }
13 }

```

Now that we see what our custom form control looks like when using with Angular's two different form APIs let's dig into the code for how the app-switch is implemented.

Building a Custom Form Control

First, let's take a look at the template for our custom form control app-switch.

```

1 <div (click)="switch()" class="switch" [ngClass]="{ 'checked': value }" [attr.title]\
2 ="label">
3   <input type="checkbox" class="switch-input" [value]="value" [attr.checked]="valu\
4 e" [id]="ID">
5   <span class="switch-label" data-on="On" data-off="Off"></span>
6   <span class="switch-handle"></span>
7 </div>

```

In the component template, there are a few dynamic properties and events. There is a click event to toggle the value. We also bind to the value to set our checkbox value and our CSS class for styles.


```
1  .switch {
2      position: relative;
3      display: inline-block;
4      vertical-align: top;
5      width: 80px;
6      height: 36px;
7      padding: 3px;
8      margin-bottom: 8px;
9      background-color: none;
10     cursor: pointer;
11     overflow: visible;
12 }
13
14 .switch:focus-within {
15     outline: 2px solid Highlight;
16 }
17
18 .switch-input {
19     position: absolute;
20     top: 0;
21     left: 0;
22     opacity: 0;
23     padding: 0;
24     margin: 0;
25 }
26
27 .switch-label {
28     position: relative;
29     display: block;
30     height: 30px;
31     font-size: 16px;
32     font-weight: normal;
33     text-transform: uppercase;
34     background: #ccc;
35     border-radius: 4px;
36     transition: 0.15s ease-out;
37     transition-property: opacity background;
38     color: #2d2d2d;
39 }
40
41 .switch-label::before, .switch-label::after {
42     position: absolute;
43     top: 50%;
```

```
44     margin-top: -.5em;
45     line-height: 1;
46     transition: inherit;
47 }
48
49 .switch-label::before {
50     content: attr(data-off);
51     right: 11px;
52     color: #fff;
53 }
54
55 .switch-label::after {
56     content: attr(data-on);
57     left: 11px;
58     color: #fff;
59     opacity: 0;
60 }
61
62 .switch-handle {
63     position: absolute;
64     top: 9px;
65     left: 10px;
66     width: 18px;
67     height: 18px;
68     background: #fff;
69     border-radius: 10px;
70     box-shadow: 1px 1px 5px #2d2d2d;
71     transition: left 0.15s ease-out;
72 }
73
74 .checked .switch-label {
75     background: #4CAF50;
76     box-shadow: inset 0 1px 2px rgba(255, 255, 255, 0.5), inset 0 0 3px rgba(0, 0, 0, \
77 0.15);
78 }
79
80 .checked .switch-label::before {
81     opacity: 0;
82 }
83
84 .checked .switch-label::after {
85     opacity: 1;
86 }
```

```

87
88 .checked .switch-handle {
89   left: 50px;
90   box-shadow: -1px 1px 5px rgba(45, 45, 45, 0.41);
91 }

```

In this chapter, we won't cover the CSS file for this component as it is not Angular specific, but you can dig into the source code in the included working code examples. Next, let's take a look at the app-switch component code and dig into the API.

```

1  import { Component, Input, forwardRef, HostBinding } from '@angular/core';
2  import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';
3
4  @Component({
5    selector: 'app-switch',
6    templateUrl: './switch.component.html',
7    styleUrls: ['./switch.component.css'],
8    providers: [
9      {
10       provide: NG_VALUE_ACCESSOR,
11       useExisting: forwardRef(() => SwitchComponent),
12       multi: true
13     }
14   ]
15 })
16 export class SwitchComponent implements ControlValueAccessor {
17   @HostBinding('attr.id')
18   externalId = '';
19
20   @Input()
21   set id(value: string) {
22     this._ID = value;
23     this.externalId = null;
24   }
25
26   get id() {
27     return this._ID;
28   }
29
30   private _ID = '';
31
32   @Input('value') _value = false;
33   onChange: any = () => {};

```

```
34   onTouched: any = () => {};  
35  
36   get value() {  
37     return this._value;  
38   }  
39  
40   set value(val) {  
41     this._value = val;  
42     this.onChange(val);  
43     this.onTouched();  
44   }  
45  
46   constructor() {}  
47  
48   registerOnChange(fn) {  
49     this.onChange = fn;  
50   }  
51  
52   writeValue(value) {  
53     if (value) {  
54       this.value = value;  
55     }  
56   }  
57  
58   registerOnTouched(fn) {  
59     this.onTouched = fn;  
60   }  
61  
62   switch() {  
63     this.value = !this.value;  
64   }  
65 }
```

A lot is going on here, let's break it down. First our imports and @Component decorator.

```

1  import { Component, Input, forwardRef } from '@angular/core';
2  import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';
3
4  @Component({
5    selector: 'app-switch',
6    templateUrl: './switch.component.html',
7    styleUrls: ['./switch.component.css'],
8    providers: [
9      {
10       provide: NG_VALUE_ACCESSOR,
11       useExisting: forwardRef(() => SwitchComponent),
12       multi: true
13     }
14   ]
15 })

```

The first part of our decorator is defining the component template, CSS, and selector. The API we are interested in is under providers. Under providers, we are telling the Angular DI to extend the existing `NG_VALUE_ACCESSOR` token and use `SwitchComponent` when requested. We then set `multi` to `true`. This mechanism enables `multi` providers. Essentially allowing multiple values for a single DI token. This allows natural extensions to existing APIs for developers. This essentially registers our custom component as a custom form control for Angular to process in our templates. Next, let's look at our component class.

```

1  export class SwitchComponent implements ControlValueAccessor {
2    @HostBinding('attr.id')
3    externalId = '';
4
5    @Input()
6    set id(value: string) {
7      this._ID = value;
8      this.externalId = null;
9    }
10
11    get id() {
12      return this._ID;
13    }
14
15    private _ID = '';
16
17    @Input('value') _value = false;
18    onChange: any = () => { };
19    onTouched: any = () => { };

```

```

20
21   get value() {
22       return this._value;
23   }
24
25   set value(val) {
26       this._value = val;
27       this.onChange(val);
28       this.onTouched();
29   }
30
31   registerOnChange(fn) {
32       this.onChange = fn;
33   }
34
35   registerOnTouched(fn) {
36       this.onTouched = fn;
37   }
38
39   writeValue(value) {
40       if (value) {
41           this.value = value;
42       }
43   }
44
45   switch() {
46       this.value = !this.value;
47   }
48 }

```

The first part of our class is the `ControlValueAccessor` interface we are extending. The `ControlValueAccessor` interface looks like this:

```

1  export interface ControlValueAccessor {
2      writeValue(obj: any) : void
3      registerOnChange(fn: any) : void
4      registerOnTouched(fn: any) : void
5  }

```

We will go over the purpose of each one of these methods below. Our component takes in a couple of different `@Inputs`.

```
1  export class SwitchComponent implements ControlValueAccessor {
2    @HostBinding('attr.id')
3    externalId = '';
4
5    @Input()
6    set id(value: string) {
7      this._ID = value;
8      this.externalId = null;
9    }
10
11    get id() {
12      return this._ID;
13    }
14
15    private _ID = '';
16
17    ...
18 }
```

The first input is the `id` input. We want to be able to set an `id` on the `app-switch` component and pass that down to the underlying checkbox in our switch component. Doing this allows developers using our component to assign a label element to the input to get the appropriate level of accessibility. There is one trick though, we must pass the `id` as an input to add to the checkbox and then remove the `id` on the `app-switch` element. We need to remove the `id` on the `app-switch` because it is not valid to have duplicate `id` attributes with the same value.

To remove the extra `id` we can use the `HostBinding` decorator to set the `attr.id` value to `null`. This allows us to receive the `id` value as an input, set the checkbox `id` internally then delete the `id` on the parent `app-switch` element. Next is the `@Input('input')` property.

```
1  export class SwitchComponent implements ControlValueAccessor {
2    @HostBinding('attr.id')
3    externalId = '';
4
5    @Input()
6    set id(value: string) {
7      this._ID = value;
8      this.externalId = null;
9    }
10
11    get id() {
12      return this._ID;
13    }
14 }
```

```
14
15   private _ID = '';
16
17   @Input('value') _value = false;
18   onChange: any = () => { };
19   onTouched: any = () => { };
20
21   get value() {
22     return this._value;
23   }
24
25   set value(val) {
26     this._value = val;
27     this.onChange(val);
28     this.onTouched();
29   }
30
31   registerOnChange(fn) {
32     this.onChange = fn;
33   }
34
35   registerOnTouched(fn) {
36     this.onTouched = fn;
37   }
38
39   writeValue(value) {
40     if (value) {
41       this.value = value;
42     }
43   }
44
45   switch() {
46     this.value = !this.value;
47   }
48 }
```

The `@Input('input')` allows us to take an input value named `input` and map it to the `_input` property. We will see the role of `onChange` and `onTouched` in shortly. Next, we have the following getters and setters.


```
1  get value() {  
2    return this._value;  
3  }  
4  
5  set value(val) {  
6    this._value = val;  
7    this.onChange(val);  
8    this.onTouched();  
9  }
```

Using getters and setters, we can set the value on the component in a private property named `_value`. This allows us to call `this.onChange(val)` and `.onTouched()`.

The next method `registerOnChange` passes in a callback function as a parameter for us to call whenever the value has changed. We set the property `onChange` to the callback, so we can call it whenever our setter on the `value` property is called. The `registerOnTouched` method passes back a callback to call whenever the user has touched the custom control. When we call this callback, it notifies Angular to apply the appropriate CSS classes and validation logic to our custom control.

```
1  registerOnChange(fn) {  
2    this.onChange = fn;  
3  }  
4  
5  registerOnTouched(fn) {  
6    this.onTouched = fn;  
7  }  
8  
9  writeValue(value) {  
10   if (value) {  
11     this.value = value;  
12   }  
13 }
```

The last method to implement from the `ControlValueAccessor` is `writeValue`. This `writeValue` is called by Angular when the value of the control is set either by a parent component or form. The final method `switch()` is called on the click event triggered from our `switch` component template.

Custom form controls are simply components that implement the `ControlValueAccessor` interface. By implementing this interface, our custom controls can now work with Template and Reactive Forms APIs seamlessly providing a great developer experience to those using our components.

Chapter 13 - Observables and RxJS

Throughout this book, we have heavily used the Reactive Forms API. If we pause and think for a moment, why is it called “Reactive”? Well Angular has several “Reactive” APIs, meaning they use Observables. Observables are a primitive object we can use in our Angular applications to handle events and asynchronous code.

Angular uses a library called RxJS to provide the Observable class primitive for its APIs. If you are familiar with JavaScript, you likely are familiar with using Promises for handling async code. Observables, however, offer us some distinct advantages over traditional JavaScript Promises. Let’s dive into Observables and RxJS to learn how they work and how they are useful for complex Angular forms!

JavaScript Promise

Most JavaScript applications use Promises to handle async code. Promises are great for async code and its all built into the browser. Let’s take a quick review with an example of a promise.

```
1  const promise = new Promise(resolve => {  
2    setTimeout(() => {  
3      resolve('Hello from a Promise!');  
4    }, 2000);  
5  });  
6  
7  promise.then(value => console.log(value));
```

We create promises by instantiating the class. In the class constructor, we pass in a function to handle the async task. In this example, we have a simple `setTimeout` function. The `setTimeout` executes our function after 2 seconds. Once executed the function calls `resolve` which tells our promise to pass the value 'Hello from a Promise!' back to the consumer of our promise. To listen for our promise value, we use the `then()` method on the promise instance.

Promises are great at handling single events in our application. Where promises fall short is when it comes to handling multiple events from a single source. For example we have many multi event APIs in the browser. A simple click event:

```
1 const button = document.querySelector('button');
2 button.addEventListener('click', (event) => console.log(event));
```

A simple click event listener is an async task that can fire multiple events over a time span. Observables are a primitive that can help manage streams of events such as our click event handler. Let's take a look at the Observable API.

RxJS Observables

Angular uses a library called RxJS to provide our Observable implementation.

```
1 import { Observable } from 'rxjs';
2
3 const observable = new Observable(observer => {
4   let count = 0;
5   setInterval(value => observer.next(count++), 1000);
6 });
7
8 observable.subscribe(value => console.log(value));
```

We import the Observable class from the RxJS package. Similar to the promise, we instantiate the Observable. In the constructor, we pass in a function that is responsible for defining the behavior of when the Observable should emit values. The Observable passes in an observer object which has a next() method. The next() method allows us to trigger an event whenever we choose.

In this example, instead of a setTimeout we have a setInterval that runs every 1000 milliseconds and trigger an event to fire. To listen to these events, we use the subscribe() method instead of then() like our promise example. If we look in the console, we see the following,

```
1 1
2 2
3 3
4 ...
```

As we can see, we can listen to multiple events now. Next, we need to learn how to stop listening to events when they are no longer applicable to our use case.

Unsubscribing

We also can unsubscribe from an Observable when we no longer need to receive events. When we subscribe to an Observable, we receive back a subscription object.

```
1  const subscription = observable.subscribe(value => console.log(value));
```

With the subscription object, we can unsubscribe from our Observable and stop receiving events.

```
1  const subscription = observable.subscribe(value => console.log(value));
2
3  setTimeout(( ) => subscription.unsubscribe(), 3000);
```

In this example, we would receive events for three seconds and then unsubscribe and no longer receive events. While the console log will no longer fire, we still have a bit of work in our source Observable to do.

```
1  import { Observable } from 'rxjs';
2
3  const observable = new Observable(observer => {
4    let count = 0;
5    setInterval(value => observer.next(count++), 1000);
6  });
7
8  observable.subscribe(value => console.log(value));
```

In our interval example, even though we unsubscribed the `setInterval` is still running internally in our Observable. We need to stop the interval when we no longer have any subscribers listening to our event. To do this, we create a cleanup function that our Observable calls for us.

```
1  import { Observable } from 'rxjs';
2
3  const observable = new Observable(observer => {
4    let count = 0;
5    const interval = setInterval(value => observer.next(count++), 1000);
6
7    return () => {
8      clearInterval(interval);
9    }
10 });
11
12 observable.subscribe(value => console.log(value));
```

In our constructor function, RxJS expects a return function as the cleanup function. When all subscribers have unsubscribed the Observable calls the cleanup function where we can stop any long running tasks preventing any memory leaks.

Operators

In RxJS we get not only the Observable class but also an extensive collection of utility functions called operators. Operators allow us to change or update events as they come into our subscribers. Let's take a look at a basic operator `map()`.

```
1 import { Observable } from 'rxjs';
2 import { map } from 'rxjs/operators';
3
4 const interval = new Observable(observer => {
5   let count = 0;
6   const interval = setInterval(value => observer.next(count++), 1000);
7
8   return () => {
9     clearInterval(interval);
10  }
11 });
12
13 interval.pipe(
14   map(value => value * value)
15 ).subscribe(value => console.log(value));
```

On our interval Observable, we can use the `pipe()` method. The `pipe()` method takes one to many operators as its parameters. As the Observable emits each event, it passes the event value to each one of our operators. Each operator has a different behavior to handle our events.

In our example we are using an operator called `map()` this is similar to the JavaScript array `map` method. Instead of looping over each value in an array and changing the value, we are looping over each event as it is emitted from the Observable.

```
1 import { map } from 'rxjs/operators';
2
3 interval.pipe(
4   map(value => value * value)
5 ).subscribe(value => console.log(value));
```

As each value emits, it is passed into the `map` operator where we can mutate the value before it gets to the subscriber. In this case, we square the value and return it as the new value.

```

1 // ----1----2----3----4---->
2 //      map => x * x
3 // ----1----4----9----16---->

```

We can also filter out only certain events we care about. Using the `filter()` operator, we can select specific events based on a certain condition we define.

```

1 import { map } from 'rxjs/operators';
2
3 interval.pipe(
4   filter(value => value % 2 === 0)
5 ).subscribe(value => console.log(value));

```

In this example, every event is passed through the `filter()` method. In our filter, we can define a condition that determines if the event is allowed to continue to the subscriber. In this case, we are using the modulo operator to determine if the value is even or odd. If it is even the statement is true, and the event continues.

```

1 // ----1----2----3----4---->
2 //      filter
3 // -----2-----4---->

```

With operators, we can combine and chain the behavior.

```

1 interval.pipe(
2   map(value => value * value),
3   filter(value => value % 2 === 0)
4 ).subscribe(value => console.log(value));
5
6 // ----1----2----3----4---->
7 //      map => x * x
8 // ----1----4----9----16---->
9 //      filter
10 // -----4-----16---->

```

Now that we know the basics of how Observable work we can dig into how Angular leverages them to make our UIs reactive to the events in our applications.

Chapter 14 - Reactive Forms

Now that we have a basic understanding of how Observables work, we can dig even deeper into RxJS and the Angular Reactive Forms API. To do this, we will build a type-ahead style search feature that will start querying image results as we type in a search form.

This search feature will have several technical requirements such as debouncing the search, so we don't overload our API with every key event. Also, performance optimizations will need to be made, such as canceling prior in flight search queries if the user starts typing again. Let's get started!

Here is a screenshot of the running search feature:

Reactive Search Component

Search Giphy



Example Reactive Search Feature

Our search API will use the Giphy API to search for Gifs. First, we will start with our template in the search component.


```

1 <form [formGroup]="searchForm">
2   <label for="search">Search Giphy</label>
3   <input formControlName="search" id="search" />
4 </form>
5
6 <a *ngFor="let result of results" [href]="result.url" target="_blank">
7   <img [src]="result.images.downsized.url" />
8 </a>

```

Looking at our template, we can see we are using the Reactive Forms API that we covered in previous chapters. We receive our results and loop over each result displaying the Gif image.

```

1 import { Component } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { FormBuilder, Validators, FormGroup } from '@angular/forms';
4 import { Observable } from 'rxjs';
5
6 export interface Result {
7   url: string;
8   images: {
9     downsized: {
10       url: string;
11     }
12   };
13 }
14
15 @Component({
16   selector: 'app-reactive-search-example',
17   templateUrl: './reactive-search-example.component.html',
18   styleUrls: ['./reactive-search-example.component.scss']
19 })
20 export class ReactiveSearchExampleComponent {
21   searchForm: FormGroup;
22   results: Result[];
23
24   constructor(private formBuilder: FormBuilder, private http: HttpClient) {
25     this.searchForm = this.formBuilder.group({
26       search: ['', Validators.required]
27     });
28
29     this.results = this.getResults();
30   }
31

```

```
32   getResults() {  
33     ...  
34   }  
35 }
```

To make our API request, we will use the `HttpClient` service. Just like injecting the `FormBuilder` we inject the `HttpClient` to use in our component.

ValueChanges Observable

To start making queries as the user types, we need to be notified whenever they begin typing. Angular Reactive Forms are called Reactive because they expose RxJS Observables as part of its API.

```
1  this.searchForm.controls.search.valueChanges.subscribe(query => {  
2    console.log(query);  
3  });
```

Each form control and the form itself provides an Observable called `valueChanges`. The `valueChanges` Observable will emit the current input value after each keystroke. For example, if we typed “cats”, we would get the following event values:

```
1  ['c', 'ca', 'cat', 'cats']
```

Now that we have our query values we can start making API requests with those values.

HttpClient

We will write the minimal code to make a successful API request and then optimize it. First, we need to construct the API request URL.

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit\  
2  =12`;
```

Our API request will be from the giphy API. You can make a free API key at <https://developers.giphy.com/>

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`;
2
3
4  this.searchForm.controls.search.valueChanges.subscribe(query => {
5    this.http.get<{data: Result[]}>(url).subscribe(data => {
6      this.results = data.results;
7    });
8  });
```

With the Angular, HttpClient service makes a get request and get our async response back via an Observable. When the response comes back, we receive that event in the subscribed value.

While what we have here worked, it has several problems. The first performance issue is we are now making an API request for every keystroke the user types which would quickly overload our API. Second, we have this awkward structure of an Observable nested in another Observable. This is similar to nesting promises. We can solve both of these issues with some RxJS magic.

SwitchMap Operator

To solve our nested Observable problem, we are going to use a particular operator called `switchMap`. The `switchMap` operator is similar to the `map` operator we covered in our previous chapter but has a couple of useful behaviors. Let's get started.

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`;
2
3
4  this.searchForm.controls.search.valueChanges.pipe(
5    map(query => this.http.get<{data: Result[]}>(url)),
6    map(res => res.data) // map our the list of search results from the API data object
7  ).subscribe(results => this.results = results);
```

If we used just a `map` operator, our code would look something like this. Every time we receive a new query make an HTTP request call and then return the value. This seems like it would work until we realize that the value we get back from the HTTP request is not the API data but an Observable that contains our API data. This is because the inner HTTP request is asynchronous and returns an Observable. So our `results` we get back in the `subscribe` would be incorrectly passing back the Observable instead of the data we want.

To fix this issue of nested subscribes, we will use the `switchMap` operator.

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`;
2
3
4  this.searchForm.controls.search.valueChanges.pipe(
5    switchMap(query => this.http.get<{data: Result[]}>(url)),
6    map(res => res.data)
7  ).subscribe(results => this.results = results);
```

Just like before we map over each value except this time we use the `switchMap` operator instead of the `map` operator. The `switchMap` is designed for receiving Observables as its value. When it gets an Observable as its value, it will subscribe automatically and return the “unwrapped” inner value to the outer Observable. Now the results in our subscribe are the actual API data we want to use.

If we look at a running example and inspect the network tab, we should see our API requests. Each time we type in a letter that triggers an event and calls our API. For the search query “cats” we get four events and four API calls.

Reactive Search Component

Search Giphy



Network Requests being Canceled

If we slow our network connection, we can see something interesting happening to those API requests. The prior three of the four requests are being canceled. This cancellation is possible because of Observables.

First, we are using the Angular `HttpClient` service. The `HttpClient` uses Observables. Observables can be canceled so if you unsubscribe from the Observable the `HttpClient` before the network event has come back, it will cancel the underlying network request. This is great for performance on mobile devices as it allows the device to prioritize requests.

The second part of how this cancellation is possible is because of our use of `switchMap`. The `switchMap` operator not only subscribes to our inner Observable but also unsubscribes from older

Observables. Each time we type a character, we get back an Observable with a pending API request. If we start typing before that request comes back, the `switchMap` will unsubscribe from the prior Observable and then subscribe to the latest. By unsubscribing, it triggers the cancellation of the HTTP request.

Now that we have our nested Observables cleaned up and we are canceling old requests let's continue to optimize this even further.

Debounce Time

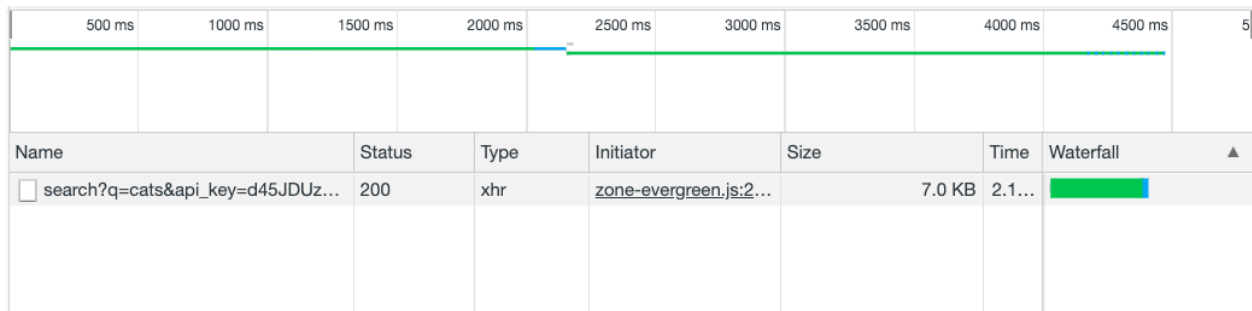
If we look at where we left off with our request Observable we have the following:

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`;
2
3
4  this.searchForm.controls.search.valueChanges.pipe(
5    switchMap(query => this.http.get<{data: Result[]}>(url)),
6    map(res => res.data)
7  ).subscribe(results => this.results = results);
```

We still need to optimize this further. Even though we cancel prior requests, the API is still being called to frequently. To solve this, we need to add a delay and wait until the user has paused typing before sending a request. To achieve this behavior, we can use the `debounceTime` operator.

```
1  const url = `https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`;
2
3
4  this.searchForm.controls.search.valueChanges.pipe(
5    debounceTime(500),
6    switchMap(query => this.http.get<{data: Result[]}>(url)),
7    map(res => res.data)
8  ).subscribe(results => this.results = results);
```

The `debounceTime` operator takes in a specified amount of time in milliseconds and will delay from allowing any events to continue in the stream until events have paused for the given time. In our example after 500 milliseconds of typing, we then will allow the query event to continue on to the `switchMap` and our HTTP request. If we look at the network requests in the browser tools, we see the following:



Debounced Requests

Now our search queries are much more efficient only making an API request after the user has stopped typing for a moment. We can further optimize our search. For example, we search “cats” and we get our results back, but then we backspace to “cat” and then quickly type “cats” again we see that a new API request is sent again for the term “cats” even though we already have the results of “cats” being displayed. We can use an operator called `distinctUntilChanged` to solve this. The `distinctUntilChanged` operator will not allow an event to continue if its new event value is the same as the old.

```
1 this.searchForm.controls.search.valueChanges.pipe(
2   debounceTime(500),
3   distinctUntilChanged()
4   switchMap(query => this.http.get<{data: Result[]}>(url)),
5   map(res => res.data)
6 ).subscribe(results => this.results = results);
```

So now if we quickly change our search query back to the previous query, it won’t trigger a new request since the prior event was “cats” and the new event is “cats”. The `distinctUntilChanged` is really useful in RxJS when you don’t want to be notified of an event unless the event value has changed.

We could take our optimizations even one small step further by using the `filter` operator. Using the `filter` operator, we could filter out queries that are no longer than two characters as they won’t provide enough text for a quality result.

```
1 this.searchForm.controls.search.valueChanges.pipe(
2   filter(value => value.length > 2),
3   debounceTime(500),
4   distinctUntilChanged()
5   switchMap(query => this.http.get<{data: Result[]}>(url)),
6   map(res => res.data)
7 ).subscribe(results => this.results = results);
```

So now we have optimized our search feature we can clean up the code even more. Remember, in the previous chapter, we discussed how it is essential to unsubscribe from Observables. We can let

Angular handle some of this work for us. First, we remove the subscribe statement and assign the Observable as a property of our component.

```

1  import { Component } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3  import { FormBuilder, Validators, FormGroup } from '@angular/forms';
4  import { Observable } from 'rxjs';
5  import { map, filter, switchMap, debounceTime, distinctUntilChanged } from 'rxjs/operators';
6
7
8  // create your own API key here https://developers.giphy.com/
9  const API_KEY = '';
10
11 export interface Result {
12   url: string;
13   images: {
14     downsized: {
15       url: string;
16     }
17   };
18 }
19
20 @Component({
21   selector: 'app-reactive-search-example',
22   templateUrl: './reactive-search-example.component.html',
23   styleUrls: ['./reactive-search-example.component.scss']
24 })
25 export class ReactiveSearchExampleComponent {
26   searchForm: FormGroup;
27   results: Observable<Result[]>;
28
29   constructor(private formBuilder: FormBuilder, private http: HttpClient) {
30     this.searchForm = this.formBuilder.group({
31       search: ['', Validators.required]
32     });
33
34     this.results = this.searchForm.controls.search.valueChanges.pipe(
35       filter(value => value.length > 2),
36       debounceTime(500),
37       distinctUntilChanged(),
38       switchMap(query => this.http.get<{data: Result[]}>(`https://api.giphy.com/v1/gifs/search?q=${query}&api_key=${API_KEY}&limit=12`)),
39       map(res => res.data)
40   }

```



```
41     );  
42   }  
43 }
```

Now instead of us subscribing and unsubscribing we let Angular handle this for us by using the `async` pipe in the component template.

```
1 <form [formGroup]="searchForm">  
2   <label for="search">Search Giphy</label>  
3   <input formControlName="search" id="search" />  
4 </form>  
5  
6 <a *ngFor="let result of results | async" [href]="result.url" target="_blank">  
7   <img [src]="result.images.downsized.url" />  
8 </a>
```

The `async` pipe tells Angular that our `results` data is an Observable which allows Angular to automatically subscribe and unsubscribe when the component is created and destroyed.

Now we have built an efficient search feature with just a couple of lines of RxJS code. Angular uses Observables for all async events because it makes it easy to compose the events together as Observables. We can see several APIs like forms and HTTP requests working seamlessly together because of the standard Observable API.

With the Angular Reactive Forms API and Observables, you can build fantastic user experiences.

Chapter 15 - Reusable Form Components

When building out large Angular applications likely, you will come across the use case of creating reusable Forms as well as nesting form components. In this chapter, we will cover how to build a reusable Angular form using the `ControlValueAccessor` API that we learned in the earlier chapter.

For our use case, we will have three forms we need to build. The first form will be a create password form that will have two fields, `password` and `confirmPassword`. The second form will be a user profile form with three fields, `firstName`, `lastName`, and `email`. Our last a final form will combine both the user profile form and the create password form into a single user sign up form.

We split the sign up form into the two smaller forms to allow the forms to be reusable in other parts of our application. For example, the password form can be used in the sign up form as well as a password reset form.

Reusable Forms

First Name

Last Name

Email

Password

Confirm Password

Example of Reusable Forms in Angular

In the image above, we see all three forms. Each form is highlighted with a different color. The blue form is the user profile form. The green form is the create password form, and the red form is the combined user sign up form. Let's start with the parent user sign up form.

```

1  import { Component } from '@angular/core';
2  import { FormGroup, FormBuilder } from '@angular/forms';
3
4  @Component({
5    selector: 'app-reusable-forms-example',
6    templateUrl: './reusable-forms-example.component.html',
7    styleUrls: ['./reusable-forms-example.component.scss']
8  })
9  export class ReusableFormsExampleComponent {
10    signupForm: FormGroup;
11
12    constructor(private formBuilder: FormBuilder) {
13      this.signupForm = this.formBuilder.group({
14        password: [],
15        profile: []
16      });
17    }
18
19    submit() {
20      console.log(this.signupForm.value);
21    }
22  }

```

Our parent form is a standard reactive form using the `FormBuilder` service. Notice how we have only two control names, `password` and `profile`. Only two control names are needed as they represent a single subform control. Each of our form controls are nested reactive forms. Even though they are nested forms, the parent form sees them as just another form control.

```

1  <form [formGroup]="signupForm" (ngSubmit)="submit()">
2    <app-profile-form formControlName="profile"></app-profile-form>
3    <app-password-form formControlName="password"></app-password-form>
4    <button>Sign Up</button>
5  </form>

```

By having our subforms implement the `ControlValueAccessor` API, each subform is a stand-alone reusable form control. By making each subform a standalone control, it makes it easy to reuse, validate, and nest our custom forms in Angular.

Sub Forms with Control Value Accessor

Let's take a look at the profile form to see how to implement it using the `ControlValueAccessor` API to make it reusable. Our profile form has three inputs, `firstName`, `lastName`, and `email`. In this form, the `email` input is required. Let's first start with the template.

```

1 <div [formGroup]="form">
2   <label for="first-name">First Name</label>
3   <input formControlName="firstName" id="first-name" />
4
5   <label for="last-name">Last Name</label>
6   <input formControlName="lastName" id="last-name" />
7
8   <label for="email">Email</label>
9   <input formControlName="email" type="email" id="email" />
10  <div *ngIf="emailControl.touched && emailControl.hasError('required')" class="error"
11  r">
12    email is required
13  </div>
14 </div>

```

This form is a standard reactive form but notice that we don't use a form tag and instead use a div. We don't use a form tag because when we make this a custom control that we can embed into other forms and we cannot nest a form element in another form. In the TypeScript, we will create a reactive form using the FormBuilder as well as the FormControl API we learned in our previous chapter.

```

1 import { Component, forwardRef, OnDestroy } from '@angular/core';
2 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3
4 @Component({
5   selector: 'app-profile-form',
6   templateUrl: './profile-form.component.html',
7   styleUrls: ['./profile-form.component.scss']
8 })
9 export class ProfileFormComponent {
10   form: FormGroup;
11
12   constructor(private formBuilder: FormBuilder) {
13     this.form = this.formBuilder.group({
14       firstName: [],
15       lastName: [],
16       email: ['', Validators.required]
17     });
18   }
19
20   get email() {
21     return this.form.controls.email;

```

```

22     }
23 }

```

The profile form we create with the FormBuilder service. To make the form reusable we will use the ControlValueAccessor to map the form to the parent form and relay updates such as value changes and validations updates.

```

1  import { Component, forwardRef, OnDestroy } from '@angular/core';
2  import { ControlValueAccessor, NG_VALUE_ACCESSOR, FormBuilder, FormGroup, Validators\
3  , FormControl, NG_VALIDATORS } from '@angular/forms';
4  import { Subscription } from 'rxjs';
5
6  // describes what the return value of the form control will look like
7  export interface ProfileFormValues {
8      firstName: string;
9      lastName: string;
10     email: number;
11 }
12
13 @Component({
14     selector: 'app-profile-form',
15     templateUrl: './profile-form.component.html',
16     styleUrls: ['./profile-form.component.scss'],
17     providers: [
18         {
19             provide: NG_VALUE_ACCESSOR,
20             useExisting: forwardRef(() => ProfileFormComponent),
21             multi: true
22         },
23         {
24             provide: NG_VALIDATORS,
25             useExisting: forwardRef(() => ProfileFormComponent),
26             multi: true,
27         }
28     ]
29 })
30 export class ProfileFormComponent implements ControlValueAccessor, OnDestroy {
31     form: FormGroup;
32     subscriptions: Subscription[] = [];
33
34     get value(): ProfileFormValues {
35         return this.form.value;
36     }

```

```
37
38   set value(value: ProfileFormValues) {
39     this.form.setValue(value);
40     this.onChange(value);
41     this.onTouched();
42   }
43
44   get emailControl() {
45     return this.form.controls.email;
46   }
47
48   constructor(private formBuilder: FormBuilder) {
49     // create the inner form
50     this.form = this.formBuilder.group({
51       firstName: [],
52       lastName: [],
53       email: ['', Validators.required]
54     });
55
56     this.subscriptions.push(
57       // any time the inner form changes update the parent of any change
58       this.form.valueChanges.subscribe(value => {
59         this.onChange(value);
60         this.onTouched();
61       })
62     );
63   }
64
65   ngOnDestroy() {
66     this.subscriptions.forEach(s => s.unsubscribe());
67   }
68
69   onChange: any = () => {};
70   onTouched: any = () => {};
71
72   registerOnChange(fn) {
73     this.onChange = fn;
74   }
75
76   writeValue(value) {
77     if (value) {
78       this.value = value;
79     }

```

```

80
81     if (value === null) {
82         this.form.reset();
83     }
84 }
85
86 registerOnTouched(fn) {
87     this.onTouched = fn;
88 }
89
90 // communicate the inner form validation to the parent form
91 validate(_: FormControl) {
92     return this.form.valid ? null : { profile: { valid: false, }, };
93 }
94 }

```

In our decorator, we register the component using `NG_VALUE_ACCESSOR` as well as using `NG_VALIDATORS` to have angular acknowledge that this form will self validate. By self-validating, we can have the form validate its inputs and then communicate the validation state to the parent form.

In the constructor, we listen for our inner form values and then trigger the control to update that the form value has changed.

```

1  this.subscriptions.push(
2    this.form.valueChanges.subscribe(value => {
3      this.onChange(value);
4      this.onTouched();
5    })
6  );

```

We also want the parent form to be able to know if the profile form is valid or not. To do this, we implement a `validate()` method.

```

1  validate(_: FormControl) {
2    return this.form.valid ? null : { profile: { valid: false, } };
3  }

```

If the inner form is invalid, then we communicate back to the parent form that the inner form is in an invalid state which will allow us to handle validation at the parent level. Next, let's take a look at the Password form.

Reusable Password Creation Form

The password form uses the same technique as our profile form. We will use the FormBuilder service as well as the ControlValueAccessor API.

```

1  <div [formGroup]="form">
2    <label for="password">Password</label>
3    <input formControlName="password" type="password" id="password" />
4    <div *ngIf="passwordControl.touched && passwordControl.hasError('required')" class\
5  ="error">
6      password is required
7    </div>
8
9    <label for="confirm-password">Confirm Password</label>
10   <input formControlName="confirmPassword" type="password" id="confirm-password" />
11   <div *ngIf="confirmPasswordControl.touched && confirmPasswordControl.hasError('req\
12 uired')" class="error">
13       password is required
14   </div>
15
16   <div *ngIf="passwordControl.touched && confirmPasswordControl.touched && form.hasE\
17 rror('mismatch')" class="error">
18       passwords do not match
19   </div>
20 </div>

```

The Password form has two inputs, the password as well as the confirm password. The form will also use group validation to make sure that both inputs match correctly.

```

1  import { Component, forwardRef, OnDestroy } from '@angular/core';
2  import { NG_VALUE_ACCESSOR, FormGroup, FormBuilder, ControlValueAccessor, Validators\
3  , NG_VALIDATORS, FormControl } from '@angular/forms';
4  import { Subscription } from 'rxjs';
5
6  import { matchingInputsValidator } from './validators';
7
8  export interface PasswordFormValues {
9      password: string;
10     confirmPassword: string;
11 }
12
13 @Component({

```

```
14 selector: 'app-password-form',
15 templateUrl: './password-form.component.html',
16 styleUrls: ['./password-form.component.scss'],
17 providers: [
18   {
19     provide: NG_VALUE_ACCESSOR,
20     useExisting: forwardRef(() => PasswordFormComponent),
21     multi: true
22   },
23   {
24     provide: NG_VALIDATORS,
25     useExisting: forwardRef(() => PasswordFormComponent),
26     multi: true,
27   }
28 ]
29 })
30 export class PasswordFormComponent implements ControlValueAccessor, OnDestroy {
31   form: FormGroup;
32   subscriptions: Subscription[] = [];
33
34   get value(): PasswordFormValues {
35     return this.form.value;
36   }
37
38   set value(value: PasswordFormValues) {
39     this.form.setValue(value);
40     this.onChange(value);
41     this.onTouched();
42   }
43
44   get passwordControl() {
45     return this.form.controls.password;
46   }
47
48   get confirmPasswordControl() {
49     return this.form.controls.confirmPassword;
50   }
51
52   constructor(private formBuilder: FormBuilder) {
53     this.form = this.formBuilder.group({
54       password: ['', Validators.required],
55       confirmPassword: ['', Validators.required]
56     }, { validator: matchingInputsValidator('password', 'confirmPassword') });
```

```
57
58     this.subscriptions.push(
59         this.form.valueChanges.subscribe(value => {
60             this.onChange(value);
61             this.onTouched();
62         })
63     );
64 }
65
66 ngOnDestroy() {
67     this.subscriptions.forEach(s => s.unsubscribe());
68 }
69
70 onChange: any = () => {};
71 onTouched: any = () => {};
72
73 registerOnChange(fn) {
74     this.onChange = fn;
75 }
76
77 writeValue(value) {
78     if (value) {
79         this.value = value;
80     }
81
82     if (value === null) {
83         this.form.reset();
84     }
85 }
86
87 registerOnTouched(fn) {
88     this.onTouched = fn;
89 }
90
91 validate(_: FormControl) {
92     return this.form.valid ? null : { passwords: { valid: false, }, };
93 }
94 }
```

We define our form with the `FormBuilder` and then relay that information back to the parent.

```

1  constructor(private formBuilder: FormBuilder) {
2    this.form = this.formBuilder.group({
3      password: ['', Validators.required],
4      confirmPassword: ['', Validators.required]
5    }, { validator: matchingInputsValidator('password', 'confirmPassword') });
6
7    this.subscriptions.push(
8      this.form.valueChanges.subscribe(value => {
9        this.onChange(value);
10       this.onTouched();
11     })
12   );
13 }

```

We also create the `validate` method to tell the parent form when the password control form is invalid or valid.

```

1  validate(_: FormControl) {
2    return this.form.valid ? null : { passwords: { valid: false, }, };
3  }

```

Now that we have both subforms created and defined, we can easily reuse them and compose them into other forms in our Angular application. Going back to our sign up form, we can see both subforms used as independent controls.

```

1  <form [formGroup]="signupForm" (ngSubmit)="submit()">
2    <app-profile-form formControlName="profile"></app-profile-form>
3    <app-password-form formControlName="password"></app-password-form>
4    <button>Sign Up</button>
5  </form>
6
7  <p>
8    Form is {{signupForm.valid ? 'Valid' : 'Invalid'}}
9  </p>
10
11 <pre>
12 {{signupForm.value | json}}
13 </pre>

```

When we submit our form, we get the following form values.

```
1  {  
2    "password": {  
3      "password": "123456",  
4      "confirmPassword": "123456"  
5    },  
6    "profile": {  
7      "firstName": "John",  
8      "lastName": "Doe",  
9      "email": "example@example.com"  
10   }  
11 }
```

Angular passes back the form value the way we structured our subforms making it easy to gather multiple values from composed and nested forms. By using the `ControlValueAccessor` API we can make our forms reusable and composable for our Angular applications.

Chapter 16 - Validation Management

In our previous chapter, we learned how to create reusable Angular Forms. In this chapter, we will learn with a little Component magic how to drastically simplify our form validation logic.

In our example, we are going to build a form with three inputs, user name, email and profile description.

Name

Email

Profile Description

Submit

Demo Form for Validation Management

We will start by looking at our `app.module.ts` file.

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule, ReactiveFormsModule } from '@angular/forms';
4  import { RouterModule } from '@angular/router';
5
6  import { AppComponent } from './app.component';
7  import { ValidationMessageComponent } from './validation-message.component';
8
9  @NgModule({
10   imports: [
11     BrowserModule,
12     ReactiveFormsModule
13   ],
14   declarations: [
15     ValidationMessageComponent,
16     AppComponent
17   ],
18   bootstrap: [ AppComponent ]
19 })
20 export class AppModule {
21 }
```

We will come back to the ValidationMessageComponent in a moment. First, let's take a look at the example component.

```
1  import { Component } from '@angular/core';
2  import { FormBuilder, Validators } from '@angular/forms';
3
4  import { ValidationService } from './validation.service';
5
6  @Component({
7   selector: 'app-advanced-validation-management-example',
8   templateUrl: './advanced-validation-management-example.component.html'
9 })
10 export class AdvancedValidationManagementExampleComponent {
11   userForm: any;
12
13   constructor(private formBuilder: FormBuilder) {
14
15     this.userForm = this.formBuilder.group({
16       name: ['', Validators.required],
17       email: ['', [Validators.required, ValidationService.emailValidator]],
18       profile: ['', [Validators.required, Validators.minLength(10)]]
19     });
20   }
21 }
```

```

19     });
20
21     console.log(this.userForm);
22 }
23
24 saveUser() {
25     if (this.userForm.dirty && this.userForm.valid) {
26         alert(`Name: ${this.userForm.value.name} Email: ${this.userForm.value.email}`);
27     }
28 }
29 }

```

First, we import the FormBuilder class. We inject it through our App component constructor. In our constructor is the following:

```

1  this.userForm = this.formBuilder.group({
2      name: ['', Validators.required],
3      email: ['', [Validators.required, ValidationService.emailValidator]],
4      profile: ['', Validators.required]
5  });

```

Now that we have our userForm created, let's take a look at our form template.

```

1  <form [formGroup]="userForm" (submit)="saveUser()">
2      <label for="name">Name</label>
3      <input formControlName="name" id="name" #name="ngControl" />
4      <div *ngIf="name.touched && name.hasError('required')">Required</div>
5
6      <label for="email">Email</label>
7      <input formControlName="email" id="email" #email="ngControl" />
8      <div *ngIf="email.touched && email.hasError('email')">Invalid</div>
9
10     <label for="profile">Profile Description</label>
11     <input formControlName="email" id="profile" #profile="ngControl" />
12     <div *ngIf="profile.touched && profile.hasError('required')">Invalid</div>
13
14     <button type="submit" [disabled]="!userForm.valid">Submit</button>
15 </form>

```

We could do something like this example where we show and hide based on input properties. We can create template variables with the # syntax. This would work fine, but what if our form grows with more controls? Alternatively, what if we have multiple validators on our form controls like our email example? Our template will continue to grow and become more complex.

So let's look at building a custom component that helps abstract our validation logic out of our forms. Here is the same form but with our new component.

```
1  <form [formGroup]="userForm" (submit)="saveUser()">
2    <label for="name">Name</label>
3    <input formControlName="name" id="name" />
4    <app-validation-messages [control]="userForm.get('name')"></app-validation-message\
5  s>
6
7    <label for="email">Email</label>
8    <input formControlName="email" id="email" />
9    <app-validation-messages [control]="userForm.get('email')"></app-validation-messag\
10 es>
11
12    <label for="profile">Profile Description</label>
13    <textarea formControlName="profile" id="profile"></textarea>
14    <app-validation-messages [control]="userForm.get('profile')"></app-validation-mess\
15 ages>
16
17    <button type="submit" [disabled]="!userForm.valid">Submit</button>
18 </form>
```

Here our app-validation-messages component takes in a reference of the control input to check its validity. This is what the rendered form looks like with our validation.

Name

Required

Email

Invalid email address

Profile Description

Required

Validation Messages Triggered

Here is the example code for ourapp-validation-messages component.

```

1  import { Component, Input } from '@angular/core';
2  import { FormControl } from '@angular/forms';
3  import { ValidationService } from '../validation.service';
4
5  @Component({
6    selector: 'app-validation-message',
7    template: `<div *ngIf="errorMessage !== null" class="error">{{errorMessage}}</div>`
8  })
9  export class ValidationMessageComponent {
10   @Input() control: FormControl;
11   constructor() { }
12

```

```

13  get errorMessage() {
14      for (const propertyName in this.control.errors) {
15          if (this.control.errors.hasOwnProperty(propertyName) && this.control.touched) {
16              return ValidationService.getValidatorErrorMessage(propertyName, this.control\
17 .errors[propertyName]);
18          }
19      }
20
21      return null;
22  }
23  }

```

Our `app-validation-messages` takes in an input property named `control`, which passes us a reference to a form control. If an error does exist on the form control it looks for that error in our validation service. We store what messages we would like to show in a central location in the validation service, so all validation messages are consistent application-wide. Our validation service takes in the name of the error and an optional value parameter for more complex error messages.

Here is an example of our validation service:

```

1  export class ValidationService {
2      static getValidatorErrorMessage(validatorName: string, validatorValue?: any) {
3          let config = {
4              'required': 'Required',
5              'invalidCreditCard': 'Is invalid credit card number',
6              'invalidEmailAddress': 'Invalid email address',
7              'invalidPassword': 'Invalid password. Password must be at least 6 characters l\
8 ong, and contain a number.',
9              'minlength': `Minimum length ${validatorValue.requiredLength}`
10         };
11
12         return config[validatorName];
13     }
14
15     static creditCardValidator(control) {
16         // Visa, MasterCard, American Express, Diners Club, Discover, JCB
17         if (control.value.match(/^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0\
18 -9][0-9])[0-9]{12}|3[47][0-9]{13}|3(?:0[0-5]|68)[0-9]{11}|(?:2131|1800|35\d{3}\
19 })\d{11})$/)) {
20             return null;
21         } else {
22             return { 'invalidCreditCard': true };
23         }

```

```

24     }
25
26     static emailValidator(control) {
27         // RFC 2822 compliant regex
28         if (control.value.match(/[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*/+=?^_`{|}~
29 |}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?/)) {
30             return null;
31         } else {
32             return { 'invalidEmailAddress': true };
33         }
34     }
35
36     static passwordValidator(control) {
37         // {6,100}           - Assert password is between 6 and 100 characters
38         // (?=.*[0-9])       - Assert a string has at least one number
39         if (control.value.match(/^(?=.*[0-9])[a-zA-Z0-9!@#%^&*]{6,100}$/)) {
40             return null;
41         } else {
42             return { 'invalidPassword': true };
43         }
44     }
45 }

```

In our service, we have our custom validators and a list of error messages with corresponding text that should be shown in a given use case.

Our app-validation-messages can now be used across our application and help prevent us from writing extra markup and template logic for validation messages for each individual control.

Chapter 17 - Code Examples

Thank you for purchasing Angular Form Essentials! Your download should have included a zip file with all code examples. If not, visit <https://angularforms.dev/purchased/code-examples.zip> to download the latest available examples.