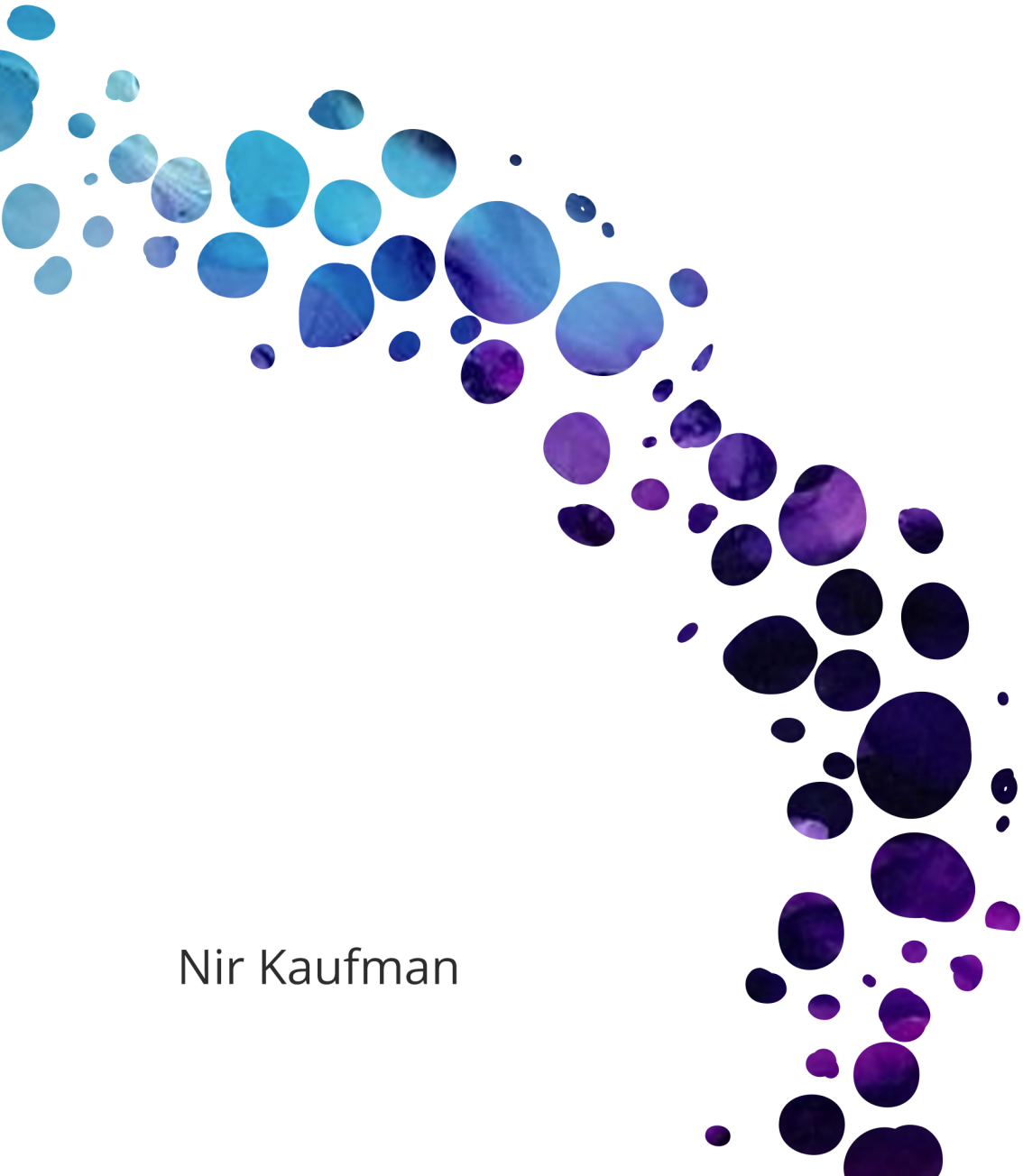
Angular Reactive Forms

A comprehensive guide for building forms with Angular



Nir Kaufman

Angular Reactive Forms

A comprehensive guide for building forms with Angular

Nir Kaufman

This book is for sale at http://leanpub.com/angular-forms

This version was published on 2017-05-13

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.

Contents

What this book is all about

This is a straightforward, comprehensive guide to Angular's Reactive Forms module. Each method is explained and demonstrated with code. It aims to be a reference book, a professional guide, and a cookbook of recipes for real-world usage.

How to use this book

The book is divided to three parts:

1. **API Guide**: Exploring the API in detail
2. **Forms Cookbook**: Practical recipes for real-world challenges
3. **Advanced Forms**: Implementation details and advanced usage

While I recommend that you read the opening section first, you can skip straight to the examples, use the API guide for reference, or dive right into the advanced stuff. It's really up to you.

Prior knowledge and experience

This book is aimed at developers who are familiar with web technologies (JavaScript, HTML, CSS) and have some experience with web development. You should be familiar with Angular (>2.x). You don't need to be an expert, but this book doesn't cover the basics. If you are new to Angular, you can start by reading the official documentation and walking through Angular's official tutorial. Visit the Angular website for more details: https://angular.io/.

Setting up a development environment

This book will be most effective if you code along with the examples. The recommended way to build an Angular application from scratch is using `angular-cli`. You can install this tool with npm: `npm install -g @angular/cli`. For detailed instructions, visit: https://cli.angular.io/.

You will need a way to edit your code. I've been using the WebStorm IDE for my projects for several years, and I highly recommend you give it a try. Visit the JetBrains website for more information: https://www.jetbrains.com/webstorm/.

An open source, free alternative is Microsoft's Visual Studio Code. It's a cross-platform text editor with some IDE features and a large collection of community plugins for extra capabilities. Visit the website for more information: https://code.visualstudio.com/.

Part 1. API Guide

The Building Blocks

A form is built from `FormControls` that we can group together in a map (`FormGroup`) or an array (`FormArray`). All three of them extends the `AbstractControl` class. In this chapter, we will explore those building blocks. We will focus on the creation and manipulation of `FormControl` objects and values.

ReactiveFormModule

Before we start, we need to import the `ReactiveFormModule` to our application:

Importing the ReactiveFormModule.

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

FormControl

Each instance of the `FormControl` class represents an individual form control element. A form control can be standalone, or included as part of a group/array with other form controls. The `FormControl`'s main role is to track, set, and retrieve the status and value of a native form element, such as a text field, checkbox, radio button, etc.

Setting up a standalone reactive `FormControl` is a two-step process:

1. Instantiate the `FormControl` class.
2. Bind the `FormControl` instance to a matching UI element.

To begin, create a fresh component, instantiate a `FormControl`, and assign it to a class member named `control`:

Control component class.

```
import {Component} from "@angular/core";
import {FormControl} from "@angular/forms";

@Component({
  selector   : 'app-control',
  templateUrl: './control.component.html'
})
export class ControlComponent {

  public control: FormControl;

  constructor() {
    this.control = new FormControl();
  }
}
```

The `FormControl` constructor can accept arguments, which are all optional. We will explore them in depth later.

FormControlDirective

To bind the `FormControl` instance to an input element in the template, we use the `FormControlDirective`. The `FormControlDirective`'s role is to sync a `FormControl` instance to a form control element in our template. We use the standard property binding syntax to bind our `control` property to the input element:

Control component template.

```
<input type="text"
       [formControl]="control"
       placeholder="type something...">
```

Getting the value

The `FormControl` exposes a `value` property. Let's implement a method that logs the value to the console. Add this method to the component class:

Logging the control value.

```
public logValue(){
  console.log(this.control.value);
}
```

And bind it to the `keydown` event of the input:

Binding to the keydown event.

```
<input type="text"
       [formControl]="control"
       (keydown.enter)="logValue()"
       placeholder="type something...">
```

Now, type something in the text field and hit Enter. You should see the value logged to the console.

You can also bind the `value` property directly to the template and see it update while you're typing. For this, add the following code to the template:

Binding the control's value to the template.

```
<span>{{ control.value }}</span>
```

Tracking control value changes

The `FormControl` also exposes an observable named `valueChanges`. By subscribing to it we can track the value as it changes. In the following example we subscribe to the control's `valueChanges` observable inside the `OnInit()` lifecycle hook:

Subscribing to the control's value.

```
ngOnInit(): void {
  this.control.valueChanges.subscribe(value => console.log(value));
}
```

Setting the control's value

We can set the control's value by passing a value to the `FormControl` constructor as the first argument:

Passing a default value.

```
constructor() {
  this.control = new FormControl('Initial Value');
}
```

We can also set the control's value by calling the method `setValue()`:

Setting the control's value.

```
public setControlValue(){
  this.control.setValue('New Value');
}
```

Now, add a button to the template to trigger this method:

Triggering the setValue() method.

```
<button (click)="setControlValue()">Set Value</button>
```

Note that the value will log to the console because we subscribed to the `valueChanges` observable.

The `setValue()` method accepts an optional configuration object with the following properties:

1. **`onlySelf`** : If `true`, the change will affect only this control and won't affect the parent. This is set to `false` by default.
2. **`emitEvent`** : If `true`, the change will emit a `valueChanges` event. This is set to `true` by default (and that's the reason that our `valueChanges` callback logs the value).
3. **`emitModelToViewChange`** : If `false`, the change won't be reflected by the view. In other words, you won't see the value inside the input element change. This is set to `true` by default.
4. **`emitViewToModelChange`** : If `true` (the default), an `ngModelChange` event will be fired. This is relevant when using the `ngModel` directive (which is out of scope for this book).

The following example demonstrates how to use this configuration object:

Configuring the setValue() method.

```
public setControlValue(){
  this.control.setValue('New Value',{
    onlySelf: false,
    emitEvent: false,
    emitModelToViewChange: false,
  });
}
```

Click the 'Set Value' button, and note that the value in the input field doesn't change. The `valueChanges` callback does not fire either. Press the Enter key, and note that the value that logs to console is the new value that we set.

If you want to be notified when a control's value is changed (from within the code, not because of user input), you can pass a listener to the `registerOnChange()` method:

Listening for a value changing.

```
this.control.registerOnChange( () => console.log('control value changed!') );
```

Resetting the Control's Value

Resetting a control's value is as simple as calling the method `reset()`:

Resetting a value.

```
this.control.reset()
```

By default, the value will become `null`. This method accepts optional configuration that we will explore in the validation chapter.

Creating a FormGroup

The `FormGroup` class represents a collection of form controls. The `FormGroup` aggregates the values, reducing the state of all of the child controls into one object. The `FormGroup` constructor accepts a map of controls as the first argument.

Let's create a simple registration form by creating a `FormGroup` and passing a map of three controls:

Simple registration form.

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

@Component({
  selector   : 'app-control',
  templateUrl: './control.component.html'
})
export class ControlComponent implements OnInit {

  public registrationForm: FormGroup;

  constructor() {
    this.registrationForm = new FormGroup({
      username: new FormControl(),
      email   : new FormControl(),
      password: new FormControl()
    });
  }

  ngOnInit(): void {
    this.registrationForm.valueChanges.subscribe(value => console.log(value));
  }

  public logValue() {
    console.log(this.registrationForm.value);
  }
}
```

We will use the keys of the map that we pass to the `FormGroup` constructor to register our UI form elements. The value type of the map is `AbstractControl`, which means that we can declare a nested `FormGroup` and `FormArray` (as we will see soon). The `ngOnInit()` method was refactored as well, and now we are subscribed to the `registrationForm`'s value instead of the value of an individual control. The `logValue()` method also now logs the `registrationForm`'s value.

Let's update our template according to those changes so we can see our form in action:

Registration form template.

```
<form [formGroup]="registrationForm"
      (ngSubmit)="logValue()">

  <input type="text"
         formControlName="username">
  <input type="text"
         formControlName="email">
  <input type="text"
         formControlName="password">

  <button type="submit">log value</button>

</form>

<div>
  <pre>{{ registrationForm.value | json}}</pre>
</div>
```

We use the `formGroup` directive to bind our `FormGroup` instance to the form element. The `formGroup` directive emits a custom event when the form is submitted, and we can use this event to trigger our `logValue()` method.

We use the `formControlName` directive to bind the input elements to individual controls on the form, passing it keys that match the `FormGroup` keys that we defined on our class. Finally, to display the `FormGroup`'s `value` object formatted, we use the `json` pipe.

Play with the form and see how the value changes. You can use `logValue()` to log the form values to the console.

You should know that there is another method to extract the `FormGroup` values: `getRawValue()`. This method will always return all values from the group, even if some controls are `disabled`. We will learn all about disabling controls in upcoming chapters.

Setting and resetting the FormGroup's value

Just like the `FormControl`, we can set the value of the `FormGroup` by calling the method `setValue()`. This time, we will need to pass an object with the exact structure we defined when we constructed the `FormGroup` (otherwise, we will get an error):

Setting the value for the group.

```
public setValue() {
  this.registrationForm.setValue({
    username: 'nirkaufman',
    email   : 'nirkaufman@gmail.com',
    password: 'admin'
  })
}
```

If you need to set just some of the fields, you can use the `patchValue()` method instead:

Setting part of the value for the group.

```
public patchValue() {
  this.registrationForm.patchValue({
    username: 'nirkaufman'
  })
}
```

The `patchValue()` method is inherited from the `AbstractControl` class, so it can be invoked within the `FormControl` context. In the case of a single control, `patchValue()` will behave exactly the same as the `setValue()` method.

We can pass an optional configuration object, just like we did with the `FormControl`. The configuration and its effect are identical in both cases.

Also, like with the `FormControl`, we can reset the `FormGroup`'s value to `null` by calling the `reset()` method:

Resetting the group's value.

```
this.registrationForm.reset()
```

Retrieving child controls

The `FormGroup` holds the map of controls in the `controls` property. You can get a reference to each child control simply by calling the `get()` method, passing a path:

Retrieving a control from a group.

```
const username = this.registrationForm.get('username');
const email    = this.registrationForm.get(['email']);
```

As you can see, there are actually two ways to pass the path: as a string or an array of strings. Our group is flat (doesn't include any subgroups), so our paths are simple. Let's refactor our code and introduce another `FormGroup` under the name `address` that will include three more controls:

Nested address group.

```
this.registrationForm = new FormGroup({
  username: new FormControl(),
  email   : new FormControl(),
  password: new FormControl(),
  address : new FormGroup({
    city  : new FormControl(),
    street: new FormControl(),
    zip   : new FormControl()
  })
});
```

We will need to update our template to reflect the changes. Add the following just under the password field:

Nested address group template.

```
<div formGroupName="address">
  <input type="text"
         formControlName="city">
  <input type="text"
         formControlName="street">
  <input type="number"
         formControlName="zip">
</div>
```

Note that we are using the `formGroupName` directive to register our new form group.

Now, if we want to retrieve the `city` or `street` control, the path that we pass to the `get()` method will look like this:

Retrieving a control from a group.

```
const city   = this.registrationForm.get('address.city');
const street = this.registrationForm.get(['address', 'street']);
```

Retrieving the root control

`AbstractControl` defines a `root` property that retrieves the top-level ancestor of a control. For example, the root control of address is the `loginForm`:

Retrieving the root control.

```
const address = this.registrationForm.get('address');
address.root; // loginForm
```

You can also set a new root for the control by calling the `setParent()` method and passing a `FormGroup` or `FormArray` as an argument. This method is useful when building forms dynamically and will be discussed in the "Advanced Forms" part of this book.

Mutating the control group

As we expect from any data structure, we can manage the `FormGroup` controls using a straightforward API. Let's start by inspecting the `FormGroup` class:

Part of the FormGroup class definition.

```
class FormGroup extends AbstractControl {
  ...
  controls: {[key: string]: AbstractControl}
  registerControl(name: string, control: AbstractControl): AbstractControl
  addControl(name: string, control: AbstractControl): void
  removeControl(name: string): void
  setControl(name: string, control: AbstractControl): void
  contains(controlName: string): boolean
  ...
}
```

As we can see from the preceding code, `controls` is nothing more than a map of `AbstractControls`. We can call the `contains()` method to check if a control exists in the group:

Checking if a control exists.

```
this.registrationForm.contains('username');
// returns true
```

We can replace an existing control by calling the `setControl()` method, passing the name of the control we want to replace as the first argument and a new control instance as the second argument. The following code replaces the `username` control with a new one:

Replacing a control within a group.

```
this.registrationForm.setControl('username', new FormControl('default value'));
```

If the named control doesn't exist, it will be created (you still need to create a UI for it).

You can also add or remove controls from the group by calling the `addControl()` or `removeControl()` method, respectively:

Adding and removing controls in a group.

```
// add a new control to the group
this.registrationForm.addControl('firstName', new FormControl());

// remove a control from the group
this.registrationForm.removeControl('firstName');
```

Adding and removing controls in our component class doesn't affect the template. We will see examples of dynamic templates in the cookbook section of this book.

The last method that we will explore is named `registerControl()`. This method performs a very similar action to `addControl()`, with one difference: it won't update the value or validity of the `FormGroup()`:

Registering a new control.

```
this.registrationForm.registerControl('firstName', new FormControl());
```

If you log `registrationForm.value`, you won't see the new control. If you log `registrationForm.controls`, you will find the `username` control in the map.

Creating a FormArray

A `FormArray` represents an array of controls. Similar to a `FormGroup`, it aggregates the values of each of its children into an array and calculates its status based on theirs. Let's add a `FormArray` control to our form to collect multiple phone numbers. Add the following code to the group map in the component class:

Adding a FormArray for phone numbers.

```
{
  ...
  phones: new FormArray( [ new FormControl() ] );
}
```

The `FormArray` constructor requires an array of controls as the first argument. You can also provide an empty array and add controls later. In this example, we created an array and passed an instance of `FormControl`.

To display a `FormArray` in our template, we need to iterate over the `FormArray` controls and register each control with the `formControlName` directive, using an index. Everything must be wrapped in an element that we bind to the `FormArray` instance using the `formArrayName` directive:

Displaying a form array in the template.

```
<div formArrayName="phones">
  <input type="text"
         *ngFor="let phone of registrationForm.get('phones').controls;
                 let i = index"
         [formControlName]="i">
</div>
```

A few things to be aware of:

1. We must create a wrapper element (`<div>` in this case) and use the `formArrayName` directive to bind it to the `FormArray` instance.
2. We loop through the `phones` controls using the `*ngFor` directive, assigning the index to a local variable (`let i`).
3. We bind to the `formControlName` directive (square brackets) and pass the index as the `name` (this is an array; there are no "keys" like with the `FormGroup`).

Working with the FormArray

Unlike with the `FormGroup`, which is a map, we use indexes instead of keys to mutate the `FormArray`. Let's start by examining the content of the array:

Checking the length and fetching the controls.

```
const phones = <FormArray>this.registrationForm.get('phones');

phones.controls; // returns [FormControl]
phones.length; // returns 1
phones.at(0); // returns FormControl
```

On the first line we are getting a reference to the array from the `FormGroup`. The `get()` method returns an `AbstractControl`, so we cast it to `FormArray` to keep our type right. The `controls` property contains the actual array of controls, while the `length` property returns the length of the `controls` array (`1` in this case). The `at()` method accepts an index and returns the control that is stored at this position.

FormArray values

Just like the `FormGroup`, the `FormArray` exposes methods for working with its values. We are already familiar with them:

1. **`phones.value()`** – Gets the values of the controls as an array.
2. **`phones.setValue()`** – Accepts an array of values and sets the controls' values.
3. **`phones.patch()`** – Enables us to set the values of part of the array.
4. **`phones.reset()`** – Resets the controls' values.

The `setValue()`, `patch()`, and `reset()` methods accept an optional configuration object. We explored this earlier, when we learned how set the value of a `FormControl` and a `FormGroup`.

Mutating the FormArray controls

To add a new control to the end of the `FormArray`, we can use the `push()` method. If we want to add the control at a specific position, we can use the `insert()` method instead, passing an index:

Adding new controls to the array.

```
const phones = <FormArray>this.registrationForm.get('phones');

phones.push(new FormControl()); // add to the end of the array
phones.insert(2, new FormControl()); // insert at index 2
```

To replace an existing control we use the `setControl()` method, passing the index and a new `FormControl`. To remove a control, we call the `removeAt()` method, passing an index:

Updating and removing controls.

```
const phones = <FormArray>this.registrationForm.get('phones');

phones.setControl(0 ,new FormControl()); // replace control at index 0
phones.removeAt(2); // remove control at index 2
```

Using FormBuilder to reduce Code

We can reduce the amount of code used for creating our form by using helper methods from a class named `FormBuilder`. Let's inspect the form that we built up throughout this chapter:

The complete registration form code.

```
export class ControlComponent {

  public registrationForm: FormGroup;

  constructor() {
    this.registrationForm = new FormGroup({
      username: new FormControl(),
      email   : new FormControl(),
      password: new FormControl(),
      address : new FormGroup({
        city  : new FormControl(),
        street: new FormControl(),
        zip   : new FormControl()
      }),
      phones: new FormArray([new FormControl()])
    });
  }
}
```

`FormBuilder` enables us to drop the `new` keyword, which makes the code cleaner:

Using FormBuilder.

```
export class ControlComponent {

  public registrationForm: FormGroup;

  constructor(builder:FormBuilder) {
    this.registrationForm = builder.group({
      username: null,
      email   : null,
      password: null,
      address : builder.group({
        city  : null,
        street: null,
        zip   : null
      }),
      phones: builder.array([builder.control(null)])
    });
  }
}
```

`FormBuilder` is just a helper for shortening the code, so using it is simply a matter of code style.

Summary

A form is a collection of one or more `FormControl` objects grouped in one or more `FormGroup` maps or `FormArrays`. The creation of a form is a two-step process:

1. Instantiate the controls in our component class.
2. Use form directives to connect those objects to our template.

`FormGroup` and `FormArray`, being data structures, expose methods for adding, updating, and removing controls.

Tracking Control State

In the previous chapter we learned how to build a form and get a value out of it. In this chapter we will learn how to track and react to the state and status of a control.

Control state properties

The state of a control can be one of the following:

1. **`touched`** – The user has triggered the `blur` DOM event.
2. **`untouched`** – The opposite of the `touched` state; the `blur` event has not triggered yet.
3. **`dirty`** – The user has changed the value (in the UI).
4. **`pristine`** – The opposite of `dirty`; the user has not changed the value yet.

Part of a control's state is the validation status, which can be one of these:

1. **`valid`** – The control has passed the validation checks.
2. **`invalid`** – At least one of the validation tests failed.
3. **`pending`** – Validation is in process (e.g., waiting for an asynchronous validator to complete).
4. **`disabled`** – The control is not being validated.

The next chapter is dedicated to validation; we will learn more about validation status there.

For now, let's build a simple form. Our form will contain a group of two controls, as follows:

A simple form.

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

@Component({
  selector   : 'app-state-form',
  templateUrl: './state-form.component.html',
  styleUrls  : ['state-form.component.css']
})
export class StateFormComponent {

  public stateForm: FormGroup;
```

```
12
13   constructor(fb: FormBuilder) {
14     this.stateForm = fb.group({
15       firstControl : null,
16       secondControl: null
17     });
18   };
19
20   get firstControl(): FormControl {
21     return <FormControl>this.stateForm.get('firstControl');
22   }
23 }
```

In the component template, let's start by binding the state properties of the `firstControl` control:

```
1  <form [formGroup]="stateForm">
2
3    <input type="text" formControlName="firstControl">
4    <input type="text" formControlName="secondControl">
5
6  </form>
7
8  <pre>
9
10   First Control State:
11
12   valid    : {{ firstControl.valid }}
13   invalid  : {{ firstControl.invalid }}
14   touched  : {{ firstControl.touched }}
15   untouched: {{ firstControl.untouched }}
16   dirty    : {{ firstControl.dirty }}
17   pristine : {{ firstControl.pristine }}
18
19 </pre>
```

You can interact with the form to see how the state changes.

Let's add a simple validation rule to see the validation status change as well. For now, we will use a built-in class imported from the `ReactiveFormsModule` named `Validators`, and call a static method named `required()`.

Let's refactor the component constructor:

Adding validation.

```
constructor(fb: FormBuilder) {
  this.stateForm = fb.group({
    firstControl : [null, Validators.required],
    secondControl: [null, Validators.required]
  });
};
```

Displaying a validation message

Now, both controls have a validation rule that forces a value to be entered. We can use the `invalid` property to conditionally display a message to the user if the `firstControl` control is empty. Add the following line under the `firstControl` input:

Adding a validation message.

```
<span *ngIf="firstControl.invalid">A value is required!</span>
```

You can enter a value in the `firstControl` input to see the message disappear.

Tracking the Group's state

The `FormGroup` aggregates the values of its child controls into one object, which means that a `FormGroup` is valid only if all of its child controls are valid. Let's bind the `FormControl` state properties to the template:

Refactoring the component template.

```
<pre>

   State Group State:

   valid    : {{ stateForm.valid }}
   invalid  : {{ stateForm.invalid }}
   touched  : {{ stateForm.touched }}
   untouched: {{ stateForm.untouched }}
   dirty    : {{ stateForm.dirty }}
   pristine : {{ stateForm.pristine }}

</pre>
```

You can interact with the controls and see the effect on the group. You can expect the exact same behavior from a `FormArray`.

Styling form controls

The state of a control is reflected in CSS classes that are added and removed dynamically by Angular:

1. `ng-valid/ng-invalid`
2. `ng-touched/ng-untouched`
3. `ng-dirty/ng-pristine`

To give visual feedback to the user about the state of a control, all we need to do is implement those classes:

Adding visual feedback.

```
input.ng-invalid {
  border-color: red;
}

input.ng-valid {
  border-color: green;
}
```

Setting the state manually

You can set the state of a control by calling one of the following methods:

1. `markAsTouched()`
2. `markAsUntouched()`
3. `markAsDirty()`
4. `markAsPristine()`
5. `markAsPending()`

Each method accepts an optional configuration object with one property, `onlySelf`, which is set to `false` by default. It determines whether or not the state change affects the control's parent or direct ancestors.

In the following example, we change the status of both controls to `touched`, but without affecting the status of the `formGroup`:

Setting status.

```
public setStatus(): void {
  this.firstControl.markAsTouched({onlySelf: true});
  this.secondControl.markAsTouched({onlySelf: true});
}
```

Resetting a control's state

We used the `reset()` method in the previous chapter to reset a control's value to `null`. The `reset()` method also affects the control's state, by setting it to `pristine` and `untouched`. If you invoke `reset()` on a `FormGroup` or `FormArray`, both the group (or array) and all descendants of the collection will be affected the same way.

Summary

A control's status is represented by a set of Booleans that we can use in our code or templates. Matching CSS classes will be added to the DOM element for us to implement in order to supply visual feedback to the user.

Validation

Form validation gives users a better UX by providing visual feedback and guiding them to enter valid values before sending the form to the server. The Reactive Forms module provides an easy way to perform validation tests.

Building a login form

In this chapter, we will build a simple login form that includes two text fields (`username` and `password`) and a submit button that will print the value of the form to the console:

Login form component class.

```
import {Component} from "@angular/core";
import {FormBuilder, FormControl} from "@angular/forms";

@Component({
  selector    : 'app-login',
  templateUrl: './login.component.html',
  styleUrls   : ['./login.component.css']
})
export class LoginComponent {

  public loginForm: FormGroup;

  constructor(builder: FormBuilder) {
    this.loginForm = builder.group({
      username: null,
      password: null,
    })
  }

  get username(): FormControl {
    return <FormControl>this.loginForm.get('username');
  }

  get password(): FormControl {
    return <FormControl>this.loginForm.get('password');
  }
```

```
27
28    public login() {
29      console.log(this.loginForm.value);
30    }
31  }
```

The component template should look like the following:

Login form template.

```
1  <form [formGroup]="loginForm"
2        (ngSubmit)="login()">
3
4      <input type="text"
5             formControlName="username"
6             placeholder="Enter username">
7
8      <input type="password"
9             formControlName="password"
10            placeholder="Enter password">
11
12   <button type="submit">Login</button>
13
14 </form>
```

Built-in validators

We can apply a validator function to a control in one of two ways:

1. Passing a validator to the `FormControl` constructor.
2. Setting a validator after the `FormControl` is instantiated by calling the `setValidators()` method.

We will explore the second option later. For now, let's apply some built-in validators to our form:

Using built-in validators.

```
this.loginForm = builder.group({
  username: [null, Validators.required ],
  password: [null, [
    Validators.required,
    Validators.minLength(6)
  ]],
})
```

In this example, we used a class named `Validators` that comes from the `ReactiveFormsModule`. The `Validators` class includes few common and useful static validation methods. Let's take a peek at the class definition:

The Validators class declaration.

```
export declare class Validators {

  // Validator that compares the values of the given form controls
  static equalsTo(...fieldPaths: string[]): ValidatorFn;

  // Validator that requires controls to have a non-empty value
  static required(control: AbstractControl): {[key: string]: boolean};

  // Validator that requires a control's value to be true
  static requiredTrue(control: AbstractControl): {[key: string]: boolean};

  // Validator that performs email validation
  static email(control: AbstractControl): {[key: string]: boolean;};

  // Validator that requires controls to have a value of a minimum length
  static minLength(minLength: number): ValidatorFn;

  // Validator that requires controls to have a value of a maximum length
  static maxLength(maxLength: number): ValidatorFn;

  // Validator that requires a control to match a regex to its value
  static pattern(pattern: string | RegExp): ValidatorFn;

  // No-op validator
  static nullValidator(c: AbstractControl): {[key: string]: boolean;};

  /**
```

```
28     * You can compose multiple validators into a single function
29     * that returns the union of the individual error maps.
30     */
31    static compose(validators: ValidatorFn[]): ValidatorFn;
32    static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn;
33  }
```

If you pass an array of validators instead of one, the `Validators.compose()` method will compose them into one behind the scenes. The following code snippet was taken from the Angular source code:

Composing validators.

```
1  function coerceToValidator(validator: ValidatorFn | ValidatorFn[]): ValidatorFn {
2    return Array.isArray(validator) ? composeValidators(validator) : validator;
3  }
```

Tracking validation status

`AbstractControl` exposes `valid` and `invalid` boolean properties. Because both `FormControl` and `FormGroup` extend `AbstractControl`, we can use those properties to determine the control's state. A `FormControl` is valid if all the validation checks are passed. A `FormGroup` is valid only if all of its nested `FormControl`s are valid.

To see this in action, let's bind those properties to the template. Add the following to the component template, just under the `</form>` closing tag:

Binding validation properties to the template.

```
1  <pre>
2    <code>
3      loginForm valid  : {{ loginForm.valid }}
4      loginForm invalid: {{ loginForm.invalid }}
5      ------------------------------------------------------
6      username valid   : {{ username.valid }}
7      username invalid : {{ username.invalid }}
8      ------------------------------------------------------
9      password valid   : {{ password.valid }}
10     password invalid : {{ password.invalid }}
11   </code>
12 </pre>
```

Now you can interact with the form and see the effect on both the individual controls and the form group. We can use these properties in our code as well. For example, let's prevent the logging of the form's value if it's not valid:

Preventing login.

```
public login() {
  if (this.loginForm.valid) {
    console.log(this.loginForm.value);
  }
}
```

Custom validators

As we have seen, a validator is nothing more than a static method that processes an `AbstractControl` and returns an object that describes the errors (if any occur). If the validation has passed, the validator should return `null`. For example, let's assume that the username cannot contain spaces and build a validator for that:

Custom validator.

```
import {AbstractControl} from "@angular/forms";

export class CustomValidators {

  static username(control: AbstractControl) {
    const value = control.value;

    if (value) {
      return /\s/.test(value) ? { invalidUserName: true } : null;
    }
  }
}
```

We treat our custom validator just like the built-in ones:

Applying the custom validator.

```
constructor(builder: FormBuilder) {
  this.loginForm = builder.group({
    username: [null, [
      Validators.required,
      CustomValidators.username
    ]],
    password: [null, [
      Validators.required,
```

```
      Validators.minLength(6)
    ]]
  });
}
```

You can interact with the form and see that it becomes invalid if the username contains spaces.

Working with the errors Map

The `errors` object that is a property of `AbstractControl` is the sum of all of our custom validators' `errors` objects. That's the reason we need to return an object from our validators. If all the validators return `null`, that means that all validators passed (the `errors` object is `null`). Let's bind the `errors` object to our template to see it in action. Add the following code to the component template after the `</pre>` closing tag:

Binding the errors object.

```
<div>{{ username.errors | json }}</div>
```

We can also use this object on our component class. Let's update the `login()` method:

login.component.ts.

```
public login() {

  if (this.username.errors !== null) {
    console.group('username errors');
    console.log(this.username.errors);
    console.groupEnd();
  }

  if (this.loginForm.valid) {
    console.log(this.loginForm.value);
  }
}
```

Managing the errors object manually

We can set errors from within the code. This becomes useful when validation is run manually (triggered by code, rather than the UI). For a demonstration, add the following method to the component class:

login.component.ts.

```
public setUserNameErrors() {
  this.username.setErrors({ badCharacters: true });
}
```

To see it in action, create another button in your template to trigger this method:

login.component.html.

```
<button type="button"
        (click)="setUserNameErrors()">
        Set Errors
</button>
```

The `setErrors()` method accepts an object that will override the current one. You can also provide a second configuration object which has an `emitValue` Boolean property (this defaults to `true`). This event is relevant if we are subscribed to the validity status observable, which we will cover later in this chapter.

Checking for an Error

We can check if an error exists in a control by calling the `hasError()` method, passing the error code we are looking for. We can also retrieve the error data by calling the `getError()` method:

Checking for and retrieving errors.

```
this.username.hasError(`badCharacters`); // returns true (the error exists)
this.username.getError(`badCharacters`); // returns true (value of badCharacters)
```

The `hasError()` and `getError()` methods accept an optional path. The previous example is identical to the following one, but this time we call `hasError()` and `getError()` on the `FormGroup`:

Passing a path.

```
this.loginForm.hasError('badChars', ['username']));
this.loginForm.getError('badChars', ['username']));
```

Async validators

An async validator is a validation method that returns an observable or a promise, which needs to resolve to an `errors` object or `null`. For this example, let's create another validation method on our `CustomValidators` class that simulates an API call to check if the provided username already exists:

Async validator.

```
import {AbstractControl} from "@angular/forms";

export class CustomValidators {

  static unique(control: AbstractControl) {

    return new Promise( resolve => {
     setTimeout( () => resolve(null), 3000 );
    })
  }
}
```

Our code uses the `setTimeout()` method to suspend the resolve for 3 seconds. You can check the template and see that both the `valid` and `invalid` properties of the `username` control are set to `false` until the async validator resolves. The same goes for the `FormGroup` that wraps the control.

We pass our async validators as a third argument to the `FormControl` constructor. Let's add our `unique` validator on the `username` control:

app.component.ts.

```
constructor(builder: FormBuilder) {

  this.loginForm = builder.group({
    username: [null, [
      Validators.required,
      CustomValidators.username
    ], CustomValidators.unique],

    password: [null, [
      Validators.required,
      Validators.minLength(6)
    ]]
  });
}
```

We can also pass an array of async validators, just like with sync validators.

Passing arguments to a custom validator

If you need to pass arguments to your validator method, you can wrap it with a method that returns the actual validator. If you look at the definition of the `minLength()` validator, you will see that it accepts an argument and returns a validator function:

minLength validator.

```
static minLength(minLength: number): ValidatorFn;
```

The following custom validator checks if the numeric value of the control is within a given range:

Validating that a numeric value is within a given range.

```
static between(min: number, max: number): ValidatorFn {
  return (control: FormControl) => {
    if (control.value > min && control.value < max) {
      return null;
    }
    return {inRange: false}
  }
}
```

Tracking validation status

A control's validation status can be one of the following:

1. **`VALID`** - The control is valid (has passed all validation tests).
2. **`INVALID`** - At least one validation check has failed.
3. **`PENDING`** - Validation is in process (like in the case of an async validator).
4. **`DISABLED`** - Validation has been disabled for this control.

Note that uppercase strings are used for the validation status.

We can bind the status to the template to see it in action. Add the following to the `<code>` block in the component template:

Binding the group's status in the template.

```
<pre>
  <code>
    loginForm Status : {{ loginForm.status }}
    ---------------------------------------------------------
  </code>
</pre>
```

Each time you enter a value in the `username` field, the control runs its validators (including our async validator) and the status becomes `PENDING`. After 3 seconds, the status will become `VALID` or `INVALID`, depending on the results from the async validator and all the other validators.

Each one of the statuses is exposed separately as a Boolean as well. Let's bind them to our template. The template should look like the following:

Binding the controls' statuses.

```
<form [formGroup]="loginForm"
      (ngSubmit)="login()">

  <input type="text"
         formControlName="username"
         placeholder="Enter username">

  <input type="password"
         formControlName="password"
         placeholder="Enter password">

  <button type="submit">Login</button>

  <button type="button" (click)="setUserNameErrors()">Set Errors</button>

</form>

<pre>
  <code>
    loginForm Status : {{ loginForm.status }}
    ---------------------------------------------------------
    loginForm valid   : {{ loginForm.valid }}
    loginForm invalid : {{ loginForm.invalid }}
    loginForm pending : {{ loginForm.pending }}
    loginForm disabled: {{ loginForm.disabled }}
```

```
    ------------------------------------------------------
    loginForm valid  : {{ loginForm.valid }}
    loginForm invalid: {{ loginForm.invalid }}
    ------------------------------------------------------
    username valid   : {{ username.valid }}
    username invalid : {{ username.invalid }}
    ------------------------------------------------------
    password valid   : {{ password.valid }}
    password invalid : {{ password.invalid }}
  </code>

  <div>{{ username.errors | json }}</div>

</pre>
```

We will see how to set the `DISABLED` status later in this chapter.

You can mark a control as `PENDING` at any time by calling the method `markAsPending()`:

Marking a control as pending.

```
public setUsernamePending() {
  this.username.markAsPending();
}
```

You can add a button to the template to trigger this method and watch the status change. Add the following line just under the Set Errors button:

Triggering a status change.

```
<button type="button" (click)="setUsernamePending()">Set Pending</button>
```

You can pass an optional configuration object to determine whether the status change will affect the parent control as well. Let's refactor the `setUsernamePending()` method to affect only the `username` control, and not the entire group:

Configuring the setUsernamePending() method.

```
public setUsernamePending() {
  this.username.markAsPending({ onlySelf: true });
}
```

Click the Set Pending button and watch how only the `username` control is affected.

Reactive status changes

We can subscribe to a control's `statusChanges` observable in the component class. The `subscribe()` method returns a `Subscription` object that we can use later to unsubscribe. Let's implement the `OnInit()` and `OnDestroy()` lifecycle hooks for `subscribe()` and `unsubscribe()`:

Subscribing to status changes.

```
export class LoginComponent implements OnInit, OnDestroy {

  public loginForm: FormGroup;
  private statusSubscription: Subscription;

  public ngOnInit(): void {
    this.statusSubscription =
        this.loginForm.statusChanges.subscribe(status => console.log(status));
  }

  public ngOnDestroy(): void {
    this.statusSubscription.unsubscribe();
  }
}
```

You can now interact with the form and watch the console for status logs.

Earlier, when we set the `errors` object manually, I mentioned that we can pass a configuration object as a second argument to the `setErrors()` method. This object contains a single property named `emitEvent`, which by default is set to `true`. If we set it to `false`, we won't be able to receive it in our `subscribe()` callback.

Currently, when you click the Set Errors button, you can see that the status logs as `INVALID`. Let's refactor our code and set the `emitEvent` property to `false`:

login.component.ts.

```
private setUserNameErrors() {
  this.username.setErrors({badCharacters: true}, {emitEvent: false});
}
```

Click the Set Errors button to trigger this method. Note that the `INVALID` state doesn't log, because no event was emitted.

Using native validation

You can use native validation attributes in the template. Angular integrates with native form validations seamlessly. Let's add the `minlength` attribute to the `username` control:

Native validations.

```
<input type="text"
       formControlName="username"
       minlength="10"
       placeholder="Enter username">
```

Disabling and enabling validation of a control

The fourth possible status of a control is `DISABLED`. When a control is disabled, it's currently exempt from validation tests. We can set a control to be disabled in two ways:

1. On instantiation, by passing the state to the `FormControl` constructor.
2. Dynamically, by calling the `disable()` method. (To reenable it, we call the `enable()` method.)

Let's explore the first option by setting the `username` control of our form to be disabled on creation:

Disabling the username control.

```
this.loginForm = builder.group({
  username: [
    { value: null, disabled: true },
    [ Validators.required, CustomValidators.username ],
    CustomValidators.unique
  ],

  password: [null, [
    Validators.required,
    Validators.minLength(6)
  ]]
```

Instated of passing a single argument (`null` in our case) as a default value for the control, we are passing an object that contains two properties: `value` and `state`. This is true also if you use the `FormControl` constructor:

Using the FormControl constructor.

```
new FormControl({ value: null, disabled: true })
```

The second option is to call the `disable()` method. In the following example, I created a toggle method that toggles the disabled status of the `username` control:

Toggling the disabled status.

```
public toggleStatus(){
  this.password.disabled ? password.enable() : password.disable();
}
```

Add a button to your template to trigger the toggle method:

Adding the toggle state button.

```
<button type="button" (click)="toggleStatus()">enable / disable</button>
```

Now, enable the `username` control (set `disabled` to `false`) and type a value to make it valid. The `FormGroup` is still invalid because the password is required. Click the "enable / disable" button to disable the `password` control and watch the form group become valid.

Controlling the side effects

Both the `enable()` and `disable()` methods accept an optional configuration object with two properties:

1. **`onlySelf`** – If `true`, the status change will not affect the control's parent (default is `false`).
2. **`emitEvent`** – If `false`, the status change won't emit an event (default is `true`).

Let's refactor the `toggleStatus()` method to set both the `username` and `password` controls:

Refactoring the toggle method.

```
public toggleStatus() {
  this.username.disabled ? this.username.enable() : this.username.disable();
  this.password.disabled ? this.password.enable() : this.password.disable();
}
```

Now, the status of our `loginForm` changes between `DISABLED` or `ENABLED` as a result of pressing our status toggle button. Let's create a configuration object that sets the `onlySelf` property to `true` and pass it to the methods:

Adding the config object.

```
public toggleStatus() {
  const config   = { onlySelf: true };

  this.username.disabled ? this.username.enable(config)
                         : this.username.disable(config);
  this.password.disabled ? this.password.enable(config)
                         : this.password.disable(config);
}
```

Now the `loginForm`'s status won't be affected by the status change of the `password` control.

Preventing the status change event

Let's subscribe to the `password` control's `statusChanges` observable in our `ngOnInit()` method:

Subscribing to password status changes.

```
public ngOnInit(): void {
  this.password.statusChanges.subscribe( status => console.log(`password: ${stat\
us}`))
}
```

Then let's set the `emitEvent` property to `false`:

Preventing emitting the status change event.

```
public toggleStatus() {
  const config   = { onlySelf: true, emitEvent: false };

  this.username.disabled ? this.username.enable(config)
                         : this.username.disable(config);
  this.password.disabled ? this.password.enable(config)
                         : this.password.disable(config);
}
```

Now when you click the "enable / disable" button, you won't see the `password` status logged in the console.

If you'd like to grab all the values from all the controls, whether they're disabled or not, you can call the `getRawValue()` method:

Getting the raw value.

```
this.loginForm.getRawValue();
```

Reacting to disabled events

You can pass a listener to the `registerOnDisabledChange()` method of a `FormControl` that will be called when that control becomes disabled or enabled. Add the following code to the `ngOnInit()` method to see it in action:

Receiving notifications on disabled events.

```
public ngOnInit(): void {
  this.username.registerOnDisabledChange( isDisabled => console.log(`username di\
sabled? ${isDisabled}`) );
}
```

Managing validators manually

Thus far, we've defined our validation when we defined our controls. However, we can set and clear the validation methods at any time by calling one of these methods:

1. **`setValidators()` / `setAsyncValidators()`** – Used to set validators.
2. **`clearValidators()` / `clearAsyncValidators()`** – Used to remove validators.

For example, let's create a new validator for the `password` control and apply it on demand. For now, we'll make the validator always return an error. Add the following method to the `CustomValidators` class:

Adding a validator.

```
static passwordStrength(control: AbstractControl) {
  return { invalidPassword: true }
}
```

Now, let's create a method on the component class to add this validator to the `password` control:

Adding the validator method.

```
public setPasswordValidator() {
  this.password.setValidators(CustomValidators.passwordStrength);
}
```

To trigger this method, add a button to the template:

Adding the validator button.

```
<button type="button"
        (click)="setPasswordValidator()"> Set Password Validator </button>
```

Now, make the form valid by entering a valid username and password. Click the Set Password Validator button to set the `passwordStrength()` validator on the `password` control.

The validator was set, but it doesn't run. As you can see, the status of the `loginForm` remains `VALID`.

If you enter another character in one of the inputs, the validation methods will run and the form will become invalid because of the new validator that we set. To rerun validators from the code, we need to call the `updateValueAndValidity()` method. Let's refactor our `setValidation()` method:

Running validations manually.

```
public setValidation () {
  this.password.setValidators(CustomValidators.passwordStrength);
  this.password.updateValueAndValidity();
}
```

The `updateValueAndValidity()` method accepts an optional configuration object with the same two properties we saw earlier, `onlySelf` and `emitEvent`.

To set async validators, use the `setAsyncValidators()` method.

Finally, to clear a control from its validators, use one of these methods:

1. **`clearValidators()`**
2. **`clearAsyncValidators()`**

Summary

Validators are static methods that process an `AbstractControl` and can be synchronous or asynchronous. We can exclude a control from being validated by disabling it. The validation state can be tracked and used to build a dynamic template, and validators can be set, cleared, checked, retrieved, and run manually from our code.

Part 2. Forms Cookbook

Nested forms

You can use the `FormArray` to create nested forms. The following example shows how to create a profile form that enable the user to add his working experience.

On the following examples we will use twitter bootstrap (v4) css framework. You can find more documentation and installing instructions here: https://v4-alpha.getbootstrap.com/

Step 1: create an experience form component:

experience.component.ts:

```
import {Component, Input} from "@angular/core";
import {FormGroup} from "@angular/forms";

@Component({
  selector   : 'app-experience',
  templateUrl: './experience.component.html',
})
export class ExperienceComponent {

  @Input()
  public group: FormGroup;

}
```

The experience component accept a `FormGroup` as an `input()`. The template includes a validation message on the `title` field:

The template will look the following:

experience.component.html:

```
<div class="card">
  <div class="card-block">

    <h4 class="card-title">Experience</h4>

    <div [formGroup]="group">

      <div class="form-group">
        <label>Title</label>
        <input type="text"
               class="form-control"
               formControlName="title"
               placeholder="Enter Title">
        <small *ngIf="group.controls.title.invalid"
               class="form-text text-danger">Title is invalid
        </small>
      </div>

      <div class="form-group">
        <label>Company</label>
        <input type="text"
               class="form-control"
               formControlName="company"
               placeholder="Company">
      </div>

      <div class="form-group">
        <label>Description</label>
        <textarea type="text"
                  class="form-control"
                  formControlName="description"
                  placeholder="description"></textarea>
      </div>

    </div>
  </div>
</div>
```

Step 2: create the profile form component:

Now, let's build the profile form:

profile.component.ts:

```
import {Component, OnInit} from "@angular/core";
import {FormBuilder, FormArray, FormGroup, Validators} from "@angular/forms";

@Component({
  selector   : 'app-profile',
  templateUrl: './profile.component.html',
})
export class ProfileComponent implements OnInit {

  public profileForm: FormGroup;
  private builder: FormBuilder;

  constructor(builder: FormBuilder) {
    this.builder = builder;
  }

  ngOnInit() {
    this.profileForm = this.builder.group({
      name      : null,
      email     : null,
      experience: this.builder.array([])
    })
  }

  get experience(): FormArray {
    return <FormArray>this.profileForm.get('experience');
  }

  private createExperienceGroup(): FormGroup {
    return this.builder.group({
      title      : [null, Validators.required],
      company    : null,
      description: null
    })
  }

  public addExperience(): void {
    this.experience.push(this.createExperienceGroup());
```

```
39    }
40  }
```

When we create the `experience` group, we set the `required` validator to the title. It will cause the validation message to display on the experience form component.

In the template, we loop over the experience controls passing the current group to the `experience` component.

profile.component.html:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Basic Details</h4>
    <form [formGroup]="profileForm">

      <div class="form-group">
        <label>Name</label>
        <input type="text"
               class="form-control"
               formControlName="name"
               placeholder="Enter name">
      </div>

      <div class="form-group">
        <label>Email address</label>
        <input type="email"
               class="form-control"
               formControlName="email"
               placeholder="Enter email">
      </div>

      <div formArrayName="experience">
        <app-experience *ngFor="let group of experience.controls; let i = index;"
                        [group]="experience.controls[i]"></app-experience>
      </div>

      <button (click)="addExperience()"
              class="btn btn-primary">+
      </button>

    </form>
```

```
32
33    </div>
34  </div>
```

You can click the + button to add an `experience` group.

Part 3. Advanced Forms

Custom Form Controls

You can build your own custom form controls that work seamlessly with the Reactive Forms API. The core idea behind form controls is the ability to access a control's value. This is done with a set of directives that implement the `ControlValueAccessor` interface.

The ControlValueAccessor Interface

`ControlValueAccessor` is an interface for communication between a `FormControl` and the native element. It abstracts the operations of writing a value and listening for changes in the DOM element representing an input control. The following snippet was taken from the Angular source code, along with the original comments:

The ControlValueAccessor interface.

```
interface ControlValueAccessor {
    /**
     * Write a new value to the element.
     */
    writeValue(obj: any): void;
    /**
     * Set the function to be called when the control receives a change event.
     */
    registerOnChange(fn: any): void;
    /**
     * Set the function to be called when the control receives a touch event.
     */
    registerOnTouched(fn: any): void;
    /**
     * This function is called when the control status changes to or from "DISAB\
LED".
     * Depending on the value, it will enable or disable the appropriate DOM ele\
ment.
     *
     * @param isDisabled
     */
    setDisabledState?(isDisabled: boolean): void;
}
```

ControlValueAccessor Directives

Each time you use the `formControl` or `formControlName` directive on a native `<input>` element, one of the following directives is instantiated, depending on the type of the input:

1. **`DefaultValueAccessor`** – Deals with all input types, excluding checkboxes, radio buttons, and select elements.
2. **`CheckboxControlValueAccessor`** – Deals with checkbox input elements.
3. **`RadioControlValueAccessor`** – Deals with radio control elements [RH: Or just "radio buttons" or "radio button inputs"?].
4. **`SelectControlValueAccessor`** – Deals with a single select element.
5. **`SelectMultipleControlValueAccessor`** – Deals with multiple select elements.

Let's peek under the hood of the `CheckboxControlValueAccessor` directive to see how it implements the `ControlValueAccessor` interface. The following snippet was taken from the Angular docs:

checkbox_value_accessor.ts.

```
import {Directive, ElementRef, Renderer, forwardRef} from '@angular/core';
import {ControlValueAccessor, NG_VALUE_ACCESSOR} from './control_value_accessor';

export const CHECKBOX_VALUE_ACCESSOR: any = {
  provide: NG_VALUE_ACCESSOR,
  useExisting: forwardRef(() => CheckboxControlValueAccessor),
  multi: true,
};

@Directive({
  selector : `input[type=checkbox][formControlName],
              input[type=checkbox][formControl],
              input[type=checkbox][ngModel]`,
  host : {
    '(change)': 'onChange($event.target.checked)',
    '(blur)'  : 'onTouched()'
  }
  ,
  providers: [CHECKBOX_VALUE_ACCESSOR]
})

export class CheckboxControlValueAccessor implements ControlValueAccessor {
  onChange  = (_: any) => {};
  onTouched = () => {};
```

```
25
26    constructor(private _renderer: Renderer, private _elementRef: ElementRef) {}
27
28    writeValue(value: any): void {
29      this._renderer.setElementProperty(this._elementRef.nativeElement, 'checked',\
30   value);
31    }
32
33    registerOnChange(fn: (_: any) => {}): void {
34      this.onChange = fn;
35    }
36
37    registerOnTouched(fn: () => {}): void {
38      this.onTouched = fn;
39    }
40
41    setDisabledState(isDisabled: boolean): void {
42      this._renderer.setElementProperty(this._elementRef.nativeElement, 'disabled'\
43  , isDisabled);
44    }
45  }
```

Let's explain what's going on:

1. This directive is instantiated when an input of type `checkbox` is declared with the `formControl`, `formControlName`, or `ngModel` directives.
2. The directive listens to `change` and `blur` events in the `host`.
3. This directive will change both the `checked` and `disabled` properties of the element, so the `ElementRef` and `Renderer` [RH: ElementRef and Renderer what? Classes?] are injected.
4. The `writeValue()` implementation is straightforward: it sets the `checked` property of the native element. Similarly, `setDisabledState()` sets the `disabled` property.
5. The function being passed to the `registerOnChange()` method is responsible for updating the outside world about changes to the value. It is called in response to a `change` event with the input value.
6. The function being passed to the `registerOnTouched()` method is triggered by the `blur` event.
7. Finally, the `CheckboxControlValueAccessor` directive is registered as a provider.

Sample Custom Form Control: Button Group

Let's build a custom `FormControl` based on the Twitter Bootstrap button group component (https://v4-alpha.getbootstrap.com/components/button-group/#basic-example).

We will start with a simple component:

custom-control.component.ts.

```
import {Component} from "@angular/core";

@Component({
  selector   : 'rf-custom-control',
  templateUrl: 'custom-control.component.html',
})
export class CustomControlComponent {

  private level: string;
  private disabled: boolean;

  constructor(){
    this.disabled = false;
  }

  public isActive(value: string): boolean {
    return value === this.level;
  }

  public setLevel(value: string): void {
    this.level = value;
  }
}
```

Here is the template:

custom-control.component.html.

```
<div class="btn-group btn-group-lg">

  <button type="button"
          class="btn btn-secondary"
          [class.active]="isActive('low')"
          [disabled]="disabled"
          (click)="setLevel('low')">low</button>

  <button type="button"
          class="btn btn-secondary"
          [class.active]="isActive('medium')"
          [disabled]="disabled"
          (click)="setLevel('medium')">medium</button>
```

```
14
15    <button type="button"
16            class="btn btn-secondary"
17            [class.active]="isActive('high')"
18            [disabled]="disabled"
19            (click)="setLevel('high')">high</button>
20  </div>
```

Next, let's implement the `ControlValueAccessor` interface:

custom-control.ts component class.

```
1   export class CustomControlComponent implements ControlValueAccessor {
2
3     private level: string;
4     private disabled: boolean;
5     private onChange: Function;
6     private onTouched: Function;
7
8     constructor() {
9       this.onChange  = (_: any) => {};
10      this.onTouched = () => {};
11      this.disabled  = false;
12    }
13
14    public isActive(value: string): boolean {
15      return value === this.level;
16    }
17
18    public setLevel(value: string): void {
19      this.level = value;
20      this.onChange(this.level);
21      this.onTouched();
22    }
23
24    writeValue(obj: any): void {
25      this.level = obj;
26    }
27
28    registerOnChange(fn: any): void {
29      this.onChange = fn;
30    }
31
```

```
32    registerOnTouched(fn: any): void {
33      this.onTouched = fn;
34    }
35
36    setDisabledState(isDisabled: boolean): void {
37      this.disabled = isDisabled;
38    }
39  }
```

The last step is to register our custom control component under the `NG_VALUE_ACCESSOR` token. `NG_VALUE_ACCESSOR` is an `OpaqueToken` used to register multiple `ControlValue` providers. (If you are not familiar with `OpaqueToken`, the `multi` property, and the `forwardRef()` function, read the official dependency injection guide on the Angular website: https://angular.io/docs/ts/latest/guide/dependency-injection.html).

Here's how we register the `CustomControlComponent` as a provider:

Registering the control as a provider.

```
1  const CUSTOM_VALUE_ACCESSOR: any = {
2    provide    : NG_VALUE_ACCESSOR,
3    useExisting: forwardRef(() => CustomControlComponent),
4    multi      : true,
5  };
6
7  @Component({
8    selector   : 'app-custom-control',
9    providers  : [CUSTOM_VALUE_ACCESSOR],
10   templateUrl: 'custom-control.component.html',
11 })
```

Our custom control is ready. Let's try it out:

app.component.ts.

```
1  import {Component, OnInit} from "@angular/core";
2  import {FormControl} from "@angular/forms";
3
4  @Component({
5    selector: 'rf-root',
6    template: `
7      <div class="container">
8        <h1 class="h1">REACTIVE FORMS</h1>
```

```
9
10        <rf-custom-control [formControl]="buttonGroup"></rf-custom-control>
11
12        <pre>
13          <code>
14            Control dirty:   {{buttonGroup.dirty}}
15            Control touched: {{buttonGroup.touched}}
16          </code>
17        </pre>
18
19      </div>
20    `,
21  })
22  export class AppComponent implements OnInit {
23
24    public buttonGroup: FormControl;
25
26    constructor() {
27      this.buttonGroup = new FormControl('medium');
28    }
29
30    ngOnInit(): void {
31      this.buttonGroup.valueChanges.subscribe(value => console.log(value));
32    }
33  }
```

Custom Form Control Validation

A custom control can include validation checks as as a built-in method. To achieve this, we need to implement an interface and register our class as a provider under a specific token.

The Slider Custom Control

Let's start by building a custom form control. Our control will wrap a native range input. We'll keep it simple so we can focus on implementing the validator:

Custom form control.

```
import {Component, forwardRef} from "@angular/core";
import {ControlValueAccessor, NG_VALUE_ACCESSOR} from "@angular/forms";

const SLIDER_CONTROL_ACCESSOR = {
  provide    : NG_VALUE_ACCESSOR,
  useExisting: forwardRef(() => SliderComponent),
  multi      : true
};

@Component({
  selector   : 'app-slider',
  templateUrl: './slider.component.html',
  providers  : [SLIDER_CONTROL_ACCESSOR]

})
export class SliderComponent implements ControlValueAccessor {

  public value: number;
  public disabled: boolean;
  private onChange: Function;
  private onTouched: Function;

  constructor() {
    this.onChange  = (_: number) => {};
    this.onTouched = () => {};
  }

```

```
28    public updateValue(value: number): void {
29      this.value = value;
30      this.onChange(this.value);
31      this.onTouched();
32    }
33
34    writeValue(obj: any): void {
35      this.value = obj;
36    }
37
38    registerOnChange(fn: any): void {
39      this.onChange = fn;
40    }
41
42    registerOnTouched(fn: any): void {
43      this.onTouched = fn;
44    }
45
46    setDisabledState(isDisabled: boolean): void {
47      this.disabled = isDisabled;
48    }
49  }
```

The template will look like the following:

Custom form control template.

```
1  <input type="range"
2         [value]="value"
3         [disabled]="disabled"
4         (input)="updateValue($event.target.value)">
5  <label>{{ value }}</label>
```

Adding Custom Validation

We can add a custom validator to our custom control by implementing the `Validator` interface and registering it under the `NG_VALIDATORS` token.

Let's start by implementing a simple check for our slider component:

```
import {NG_VALIDATORS} from "@angular/forms";

export class SliderComponent implements ControlValueAccessor, Validator {
  ...
  validate(c: AbstractControl): {[p: string]: any} {
    const error = {
      inRange: false,
      value  : c.value,
      min    : 10,
      max    : 50
    };

    return (c.value >= 10 && c.value <= 50) ? null : error;
  }
  ...
```

Next, we need to register our `SliderComponent` under the `NG_VALIDATORS` token. `NG_VALIDATORS` is an `OpaqueToken` used to register validators:

Registering the control as a validator.

```
const SLIDER_CONTROL_VALIDATOR = {
  provide    : NG_VALIDATORS,
  useExisting: forwardRef(() => SliderComponent),
  multi      : true
};

@Component({
  selector   : 'app-slider',
  templateUrl: './slider.component.html',
  providers  : [SLIDER_CONTROL_ACCESSOR, SLIDER_CONTROL_VALIDATOR]
})
```

If you are not familiar with `OpaqueToken`, the `multi` property, and the `forwardRef()` function, read the official dependency injection guide on the Angular website: https://angular.io/docs/ts/latest/guide/dependency-injection.html.

Let's test our custom slider's built-in validation. Create another component, use the slider component in the template and bind the value and the validity status as well:

Testing the slider validator.

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

@Component({
  selector: 'app-chapter-6',
  template: `
    <app-slider [formControl]="slider"></app-slider>

    <pre>
      slider valid? : {{ slider.valid }}
      errors        : {{ slider.errors | json }}
    </pre>
  `,
})
export class Chapter6Component {

  public slider: FormControl;

  constructor() {
    this.slider = new FormControl(0);
  }
}
```

You can interact with the slider and see how the validation status and the `errors` object update.

Passing Arguments to a Built-in Validator

Our validator works, but the range of minimum and maximum values is hard-coded. To configure it, we can define `@Input()` properties and use those values in the validation method:

Configuring the built-in validator.

```
export class SliderComponent implements ControlValueAccessor, Validator {
  ...
  @Input() min: string;
  @Input() max: string;

  validate(c: AbstractControl): {[p: string]: any} {
    const minValue = parseInt(this.min);
    const maxValue = parseInt(this.max);

    const error = {
      inRange: false,
      value  : c.value,
      min    : minValue,
      max    : maxValue
    };

    return (c.value >= minValue && c.value <= maxValue) ? null : error;
  }
  ...
}
```

Now, we can use the `min` and `max` attributes on the slider component:

Configuring the built-in validator.

```
<app-slider [formControl]="slider" min="12" max="24"></app-slider>
```

Extracting the Validator Function

We can make our built-in validator reusable by extracting the validation function out of the component class. Doing this enables us to use it in other components just like any other validator.

We need to wrap our validator with a function that accepts arguments and returns the validator function:

Extracting the validator function.

```
export function rangeValidator(minValue: string, maxValue: string) {
  return function (c: AbstractControl): {[p: string]: any} {
    const minValue = parseInt(this.min);
    const maxValue = parseInt(this.max);

    const error = {
      inRange: false,
      value  : c.value,
      min    : minValue,
      max    : maxValue
    };

    return (c.value >= minValue && c.value <= maxValue) ? null : error;
  }
}
```

On the component class, we can set and use the validation function like this:

Using the rangeValidator function.

```
export class SliderComponent implements ControlValueAccessor, Validator, OnChang\
es {
  ...
  private rangeValidatorFn: Function;

  @Input() min: string;
  @Input() max: string;

  ngOnChanges(): void {
    this.rangeValidatorFn = rangeValidator(this.min, this.max);
  }

  validate(c: AbstractControl): {[p: string]: any} {
    return this.rangeValidatorFn(c);
  }
  ...
```

We implemented the `OnChanges` hook so we can recreate the validator if the `@Input()`s change.

Custom Validator Directive

Building a custom validation directive enables us to easily reuse our validator, and apply it to both custom and native form elements.

Because a directive is a class, we can easily inject dependencies if needed. For the next example we will convert our range validator to a directive:

Building a sliderRange directive.

```
import {Directive, Input, forwardRef} from '@angular/core';
import {Validator, AbstractControl, NG_VALIDATORS} from "@angular/forms";

@Directive({
  selector : '[sliderRange]',
  providers: [
    {
      provide    : NG_VALIDATORS,
      useExisting: forwardRef(() => SliderRangeDirective),
      multi      : true
    }
  ]
})
export class SliderRangeDirective implements Validator {

  @Input() sliderRange: number[];

  validate(c: AbstractControl): {[p: string]: any} {
    const minValue = this.sliderRange[0];
    const maxValue = this.sliderRange[1];

    const error = {
      inRange: false,
      value  : c.value,
      min    : minValue,
      max    : maxValue
    };

    return (c.value >= minValue && c.value <= maxValue) ? null : error;
  }
}
```

Now we can apply our custom validator to our slider component, or a native range element:

Using the sliderRange directive.

```
// applying the validator to our custom control
<app-slider [formControl]="slider" [sliderRange]="[5,10]"></app-slider>

// applying the validator to a native range element
<input type="range" [formControl]="slider" [sliderRange]="[5,10]">
```

Dynamic Forms

By leveraging Angular's dynamic component rendering abilities, we can build a dynamic, configuration-driven form component. In this chapter, we will build a simple dynamic form that can be easily extended.

Dynamic Control Component

Let's start by building a simple dynamic control component that will wrap a native input element.

First, we define an interface for our control's configuration:

Control config interface.

```
export interface controlConfig {
  type: string;
  name: string;
  label: string;
  placeholder: string;
}
```

Next, we build a component that uses this configuration to initialize an input element:

Dynamic control component.

```
import {controlConfig} from "./control-config";
import {Component} from '@angular/core';
import {FormGroup} from "@angular/forms";

@Component({
  selector: 'app-dynamic-control',
  template: `
    <div [formGroup]="formGroup">

      <label>{{ controlConfig.label }}</label>

      <input [type]="controlConfig.type"
             [placeholder]="controlConfig.placeholder"
             [formControlName]="controlConfig.name">
```

```
    </div>
  `,
})
export class DynamicControlComponent {
  public formGroup: FormGroup;
  public controlConfig: controlConfig;
}
```

Our component declared a `FormGroup` that we used in the template to register the control. Note that we didn't use the `@Input()` decorator. We will initialize the `FormGroup` and the `controlConfig` when we instantiate the component.

Don't forget to register the `DynamicControlComponent` under the `entryComponents` array on the module:

Registering the dynamic control component.

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {FormRendererComponent} from './form-renderer.component';
import {ReactiveFormsModule} from "@angular/forms";
import {DynamicControlComponent} from './dynamic-control.component';

@NgModule({
  imports        : [CommonModule, ReactiveFormsModule],
  declarations   : [FormRendererComponent, DynamicControlComponent],
  entryComponents: [DynamicControlComponent],
  exports        : [FormRendererComponent]
})
export class FormEngineModule {}
```

The dynamic control can be easily extended by adding more options to the configuration interface and binding it to the component template.

Dynamic Form Renderer Component

Let's build the `FormRendererComponent` to render our dynamic controls:

Form renderer component.

```
import {
  Component, ComponentFactoryResolver, EventEmitter,
  Input, OnChanges, Output, ViewChild, ViewContainerRef
} from "@angular/core";

import {FormControl, FormGroup} from '@angular/forms';
import {controlConfig} from "./control-config";
import {DynamicControlComponent} from "./dynamic-control.component";

@Component({
  selector: 'app-form-renderer',
  template: `
    <form [formGroup]="form"
          (ngSubmit)="submitForm()">

      <ng-container #container></ng-container>

      <button type="submit"
              class="btn btn-outline-primary">submit
      </button>
    </form>
  `
})
export class FormRendererComponent implements OnChanges {

  @Input()
  private formConfig: controlConfig[];

  @Output()
  public formSubmit: EventEmitter<any>;

  @ViewChild('container', {read: ViewContainerRef})
  private formContainer: ViewContainerRef;

  private cfr: ComponentFactoryResolver;
  public form: FormGroup;

```

```
38    constructor(cfr: ComponentFactoryResolver) {
39      this.cfr        = cfr;
40      this.form       = new FormGroup({});
41      this.formSubmit = new EventEmitter<any>();
42    }
43
44    public submitForm() {
45      this.formSubmit.emit(this.form.value);
46
47    }
48
49    ngOnChanges() {
50      if (this.formConfig) {
51        this.formContainer.clear();
52
53        this.formConfig.forEach(controlConfig => {
54          this.form.addControl(controlConfig.name, new FormControl());
55          this.buildControl(controlConfig);
56        });
57      }
58    }
59
60    private buildControl(controlConfig) {
61      const factory = this.cfr.resolveComponentFactory(DynamicControlComponent);
62      const control = this.formContainer.createComponent(factory);
63
64      control.instance.controlConfig = controlConfig;
65      control.instance.formGroup     = this.form;
66    }
67  }
```

Here's what's going on here:

1. We get an array of `controlConfig` types through `@Input()` binding.
2. We query the `<ng-container>` component as a `ViewContainerRef`. We will use this to attach our dynamically created control components to the template.
3. Inside the constructor we create an empty `FormGroup` instance.
4. We implement `ngOnChanges()` with a simple loop that goes through the `controlConfig` array, adding controls to the `FormGroup`, and call the `buildControl()` method.
5. The `buildControl()` method is responsible for instantiating the `DynamicControlComponent` and initializing its properties with a `controlConfig` object and the `FormGroup` instance.

6. We catch the `ngSubmit` event and emit a custom event instead.

`ng-container` is an Angular component that will not render to the DOM. This is helpful when all you need is a `viewContainer` to attach components to (like in our case).

Using the Dynamic Form Renderer

Let's use our `FormRendererComponent` to see it in action. Create another component and use the `<app-form-renderer>` as follows:

```
import {Component} from '@angular/core';
import {controlConfig} from "./form-renderer/control-config";

@Component({
  selector: 'app-root',
  template: `
    <h1>Dynamic Form Renderer</h1>

    <app-form-renderer [formConfig]="formConfig"
                       (formSubmit)="submitForm($event)">
    </app-form-renderer>

    <pre>{{ formValue | json }}</pre>
  `,
})
export class AppComponent {

  public formConfig: controlConfig[];
  public formValue: any;

  constructor() {
    this.formConfig = [
      {
        type       : 'text',
        name       : 'username',
        label      : 'User name:',
        placeholder: 'Type username...',
      }
    ]
```

```
30    }
31
32    public submitForm(value: any) {
33      this.formValue = value;
34    }
35  }
```

Summary

By leveraging Angular's capability to render components dynamically, we can build a configuration-driven form easily. This example can be extended by passing other properties to the dynamic field (like styling, error messages, validators, etc.) and creating more dynamic controls for different kind of data types.

State Driven Forms With Redux

Redux is a modern state management pattern for client side applications. In its core, redux manage a single immutable object that represents a global state. This state can be re-calculate by a pure function, as a reaction to an event. If you're not familiar with Redux, refer to redux documentation for further reading: http://redux.js.org/

Through this chapter, we will learn the basic setup for a state-driven form with Redux. We will keep it simple so we can focus on the setup process rather than the complexity of the form values and validations.

On later chapters, we will extend this example further more.

Redux And Angular

The `@angular-redux/store` library helps you integrate your redux store into your Angular applications. For this chapter, we will use this solution to demonstrate the concept of state-driven forms. You can learn more about this library by visiting: https://github.com/angular-redux/store

You can use other implantation of Redux or build your solution based on this example.

Building a State Driven Form

Use npm to install the `@angular-redux/store` package by running: `npm install --save redux @angular-redux/store`. This command will install both the redux library and the angular-redux store library.

To keep us focused, we will define our application state with a single form. Create an file named: `app.state.ts` with the following code:

App state interface.

```
export interface AppState {
  supportForm: SupportForm;
}

export interface SupportForm {
  name: string,
  email: string,
  issue: string,
  description: string
}
```

For this example we defined both the `AppState` and the `SupportForm` interface within the same file. In a real world project, it will be a good idea to separate them into different files.

Currently, we will have a single event on our application: `UPDATE_FORM` that will update the state with our `SupportForm` values. Let's built a reducer function for it:

form.reducer.ts

```
export function formReducer(state, action) {
  switch (action.type) {
    case 'UPDATE_FORM':
      return { ...state, supportForm: action.payload };
    default:
      return state;
  }
}
```

This function is effective: If a `UPDATE_FORM` action was dispatched, It will return a new state with an update `supportForm` value. Nothing fancy.

It's a good idea to use a constant instead of hard coded string for the `action.type`.

Let's configure our store in our `AppModule`:

app.module.ts

```
import {NgReduxModule, NgRedux} from "@angular-redux/store";
import {NgModule} from "@angular/core";
import {ReactiveFormsModule} from "@angular/forms";
import {BrowserModule} from "@angular/platform-browser";
import {AppComponent} from "./app.component";
import {AppState} from "./app.state";
import {formReducer} from "./form.reducer";


@NgModule({
  declarations: [AppComponent],
  imports     : [
    BrowserModule,
    ReactiveFormsModule,
    NgReduxModule,
  ],
  bootstrap   : [AppComponent]
})

export class AppModule {

  constructor(ngRedux: NgRedux<AppState>) {

    const initialState: AppState = {
      supportForm: {
        name       : null,
        email      : null,
        issue      : null,
        description: null,
      }
    };

    ngRedux.configureStore(formReducer, initialState)
  }
}
```

Let's explain what's going on:

1. First, we import the `NgReduxModule` to our `AppModule`.
2. Next, we need to inject the `NgRedux` service to the `AppModule` class.
3. We define an initial state (resetting our supportForm values to null).
4. Then, we configure our Store providing our formReducer and the initial state.

Connecting a Form Component

The last thing we need to do is to build the actual form component and connect it to our store.

This sample application contains only an `AppComponent` so we will define our form there:

app.component.ts

```
import {Component} from "@angular/core";
import {FormGroup, FormBuilder} from "@angular/forms";

@Component({
  selector: 'app-root',
  template: `
      <h1>Support Form</h1>
      <form [formGroup]="supportForm">
        <input type="text"
               formControlName="name">
        <input type="text"
               formControlName="email">
        <input type="text"
               formControlName="issue">
        <textarea formControlName="description"></textarea>
      </form>
  `,
})

export class AppComponent {

  public supportForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.supportForm = fb.group({
      name       : null,
      email      : null,
```

```
      issue      : null,
      description: null
    })
  }
}
```

Up until now, just a standard form. Nothing fancy. Note that we dropped the submit button. The next thing to do is to sync the form with our state. Let's implement it on the `ngInit` lifeCycle hook. Let's see the complete code for this component class:

app.component.ts

```
export class AppComponent implements OnInit {

  public supportForm: FormGroup;
  private store: NgRedux<AppState>;

  constructor(fb: FormBuilder, store: NgRedux<AppState>) {
    this.store       = store;
    this.supportForm = fb.group({
      name       : null,
      email      : null,
      issue      : null,
      description: null
    })
  }

  ngOnInit(): void {
    // dispatch an action when the for values changes
    this.supportForm.valueChanges
        .subscribe(values => this.store.dispatch({
          type   : 'UPDATE_FORM',
          payload: values
        }));

    // updates the form values from the state
    this.store.select('supportForm')
        .debounceTime(100)
        .subscribe(values => this.supportForm.patchValue(values));
  }
}
```

The concept is straightforward: whenever the state changed, we update the form. Whenever the form values updates from the component side, we dispatch an action to update the state.

Note that we used the `debounceTime()` function from the `ngRx` library to prevent an infinity loop.

Summary

On this chapter, we learned the basic setup for a state-driven form with angular and redux. You can implement a similar process by yourself, or use the `@angular-redux/forms` package https://github.com/angular-redux/form which is a part of the '@angular-redux' package suite.