Forms in Angular

Forms are Crucial, Forms are Complex

Forms are probably the most crucial aspect of your web application. While we often get events from clicking on links or moving the mouse, it's through *forms* where we get the majority of our rich data input from users.

On the surface, forms seem straightforward: you make an `input` tag, the user fills it out, and hits submit. How hard could it be?

It turns out, forms can be very complex. Here's a few reasons why:

- Form inputs are meant to modify data, both on the page and the server
- Changes often need to be reflected elsewhere on the page
- Users have a lot of leeway in what they enter, so you need to validate values
- The UI needs to clearly state expectations and errors, if any
- Dependent fields can have complex logic
- We want to be able to test our forms, without relying on DOM selectors

Thankfully, Angular has tools to help with all of these things.

- **`FormControl`s** encapsulate the inputs in our forms and give us objects to work with them
- **`Validator`s** give us the ability to validate inputs, any way we'd like
- **Observers** let us watch our form for changes and respond accordingly

In this chapter we're going to walk through building forms, step by step. We'll start with some simple forms and build up to more complicated logic.

`FormControl`s and `FormGroup`s

The two fundamental objects in Angular forms are `FormControl` and `FormGroup`.

`FormControl`

A `FormControl` represents a single input field - it is the smallest unit of an Angular form.

`FormControl`s encapsulate the field's value, and states such as being valid, dirty (changed), or has errors.

For instance, here's how we might use a `FormControl` in TypeScript:

```
// create a new FormControl with the value "Nate"
let nameControl = new FormControl("Nate");

let name = nameControl.value; // -> Nate

// now we can query this control for certain values:
nameControl.errors // -> StringMap<string, any> of errors
nameControl.dirty  // -> false
nameControl.valid  // -> true
// etc.
```

To build up forms we create `FormControls` (and groups of `FormControls`) and then attach metadata and logic to them.

Like many things in Angular, we have a class (`FormControl`, in this case) that we attach to the DOM with an attribute (`formControl`, in this case). For instance, we might have the following in our form:

```
<!-- part of some bigger form -->
<input type="text" [formControl]="name" />
```

This will create a new `FormControl` object within the context of our `form`. We'll talk more about how that works below.

FormGroup

Most forms have more than one field, so we need a way to manage multiple `FormControls`. If we wanted to check the validity of our form, it's cumbersome to iterate over an array of `FormControls` and check each `FormControl` for validity. `FormGroups` solve this issue by providing a wrapper interface around a collection of `FormControls`.

Here's how you create a `FormGroup`:

```
let personInfo = new FormGroup({
    firstName: new FormControl("Nate"),
    lastName: new FormControl("Murray"),
    zip: new FormControl("90210")
})
```

`FormGroup` and `FormControl` have a common ancestor (`AbstractControl`[44]). That means we can check the `status` or `value` of `personInfo` just as easily as a single `FormControl`:

[44] https://angular.io/docs/ts/latest/api/forms/index/AbstractControl-class.html

```
personInfo.value; // -> {
//   firstName: "Nate",
//   lastName: "Murray",
//   zip: "90210"
// }

// now we can query this control group for certain values, which have sensible
// values depending on the children FormControl's values:
personInfo.errors // -> StringMap<string, any> of errors
personInfo.dirty  // -> false
personInfo.valid  // -> true
// etc.
```

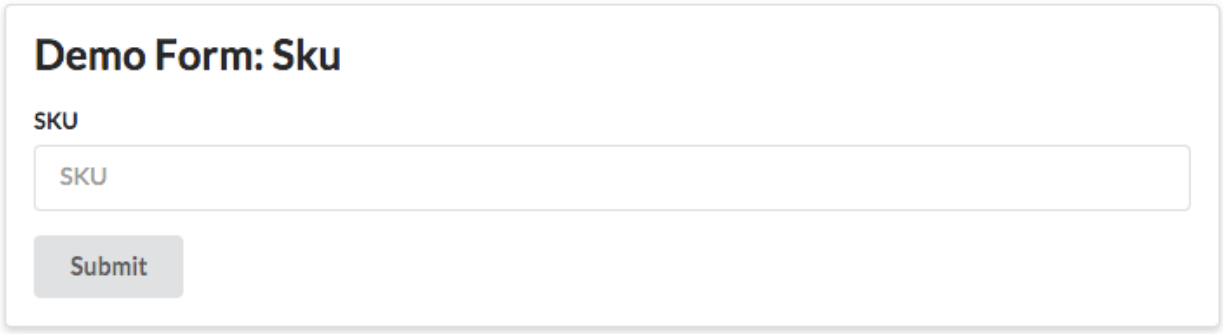
Notice that when we tried to get the `value` from the `FormGroup` we received an **object** with key-value pairs. This is a really handy way to get the full set of values from our form without having to iterate over each `FormControl` individually.

Our First Form

There are lots of moving pieces to create a form, and several important ones we haven't touched on. Let's jump in to a full example and I'll explain each piece as we go along.

You can find the full code listing for this section in the code download under `forms/`

Here's a screenshot of the very first form we're going to build:



Demo Form with Sku: Simple Version

In our imaginary application we're creating an e-commerce-type site where we're listing products for sale. In this app we need to store the product's SKU, so let's create a simple form that takes the SKU as the only input field.

SKU is an abbreviation for "stockkeeping unit". It's a term for a unique id for a product that is going to be tracked in inventory. When we talk about a SKU, we're talking about a human-readable item ID.

Our form is super simple: we have a single input for `sku` (with a label) and a submit button.

Let's turn this form into a Component. If you recall, there are three parts to defining a component:

- Configure the `@Component()` decorator
- Create the template
- Implement custom functionality in the component definition class

Let's take these in turn:

Loading the `FormsModule`

In order to use the new forms library we need to first make sure we import the forms library in our `NgModule`.

There are two ways of using forms in Angular and we'll talk about them both in this chapter: using `FormsModule` or using `ReactiveFormsModule`. Since we'll use both, we'll import them both into our module. To do this we do the following in our `app.ts` where we bootstrap the app:

```
1  import {
2    FormsModule,
3    ReactiveFormsModule
4  } from '@angular/forms';
5
6  // farther down...
7
8  @NgModule({
9    declarations: [
10     FormsDemoApp,
11     DemoFormSkuComponent,
12     // ... our declarations here
13   ],
14   imports: [
15     BrowserModule,
16     FormsModule,          // <-- add this
17     ReactiveFormsModule   // <-- and this
18   ],
19   bootstrap: [ FormsDemoApp ]
20 })
21 class FormsDemoAppModule {}
```

This ensures that we're able to use the form directives in our views. At the risk of jumping ahead, the `FormsModule` gives us *template driven* directives such as:

- `ngModel` and
- `NgForm`

Whereas `ReactiveFormsModule` gives us directives like

- `formControl` and
- `ngFormGroup`

... and several more. We haven't talked about how to use these directives or what they do, but we will shortly. For now, just know that by importing `FormsModule` and `ReactiveFormsModule` into our `NgModule` means we can *use any of the directives in that list* in our view template or *inject any of their respective providers* into our components.

Simple SKU Form: @Component Decorator

Now we can start creating our component:

code/forms/src/app/demo-form-sku/demo-form-sku.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-demo-form-sku',
5    templateUrl: './demo-form-sku.component.html',
```

Here we define a `selector` of `app-demo-form-sku`. If you recall, `selector` tells Angular what elements this component will bind to. In this case we can use this component by having a `app-demo-form-sku` tag like so:

```
1  <app-demo-form-sku></app-demo-form-sku>
```

Simple SKU Form: `template`

Let's look at our template:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku</h2>
  <form #f="ngForm"
        (ngSubmit)="onSubmit(f.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             name="sku" ngModel>
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

`form` & `NgForm`

Now things get interesting: because we imported `FormsModule`, that makes `NgForm` available to our view. Remember that whenever we make directives available to our view, they will **get attached to any element that matches their `selector`**.

`NgForm` does something handy but **non-obvious**: it includes the `form` tag in its selector (instead of requiring you to explicitly add `ngForm` as an attribute). What this means is that if you import `FormsModule`, `NgForm` will get *automatically* attached to any `<form>` tags you have in your view. This is really useful but potentially confusing because it happens behind the scenes.

There are two important pieces of functionality that `NgForm` gives us:

1. A `FormGroup` named `ngForm`
2. A **`(ngSubmit)`** output

You can see that we use both of these in the `<form>` tag in our view:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
3   <form #f="ngForm"
4         (ngSubmit)="onSubmit(f.value)"
```

First we have `#f="ngForm"`. The `#v=thing` syntax says that we want to create a local variable for this view.

Here we're creating an alias to `ngForm`, for this view, bound to the variable `#f`. Where did `ngForm` come from in the first place? It came from the `NgForm` directive.

And what type of object is `ngForm`? It is a `FormGroup`. That means we can use `f` as a `FormGroup` in our view. And that's exactly what we do in the `(ngSubmit)` output.

Astute readers might notice that I just said above that `NgForm` is automatically attached to `<form>` tags (because of the default `NgForm` selector), which means we don't have to add an `ngForm` attribute to use `NgForm`. But here we're putting `ngForm` in an attribute (value) tag. Is this a typo?

No, it's not a typo. If `ngForm` were the *key* of the attribute then we would be telling Angular that we want to use `NgForm` on this attribute. In this case, we're using `ngForm` as the *attribute* when we're assigning a *reference*. That is, we're saying the value of the evaluated expression `ngForm` should be assigned to a local template variable `f`.

`ngForm` is already on this element and you can think of it as if we are "exporting" this `FormGroup` so that we can reference it elsewhere in our view.

We bind to the `ngSubmit` action of our form by using the syntax: `(ngSubmit)="onSubmit(f.value)"`.

- `(ngSubmit)` - comes from `NgForm`
- `onSubmit()` - will be implemented in our component definition class (below)
- `f.value` - `f` is the `FormGroup` that we specified above. And `.value` will return the key/value pairs of this `FormGroup`

Put it all together and that line says "when I submit the form, call `onSubmit` on my component instance, passing the value of the form as the arguments".

`input` & `NgModel`

Our `input` tag has a few things we should touch on before we talk about `NgModel`:

code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
3   <form #f="ngForm"
4         (ngSubmit)="onSubmit(f.value)"
5         class="ui form">
6
7     <div class="field">
8       <label for="skuInput">SKU</label>
9       <input type="text"
10            id="skuInput"
11            placeholder="SKU"
12            name="sku" ngModel>
13    </div>
```

- `class="ui form"` and `class="field"` - these classes are totally optional. They come from the CSS framework Semantic UI[45]. I've added them in some of our examples just to give them a nice coat of CSS but they're not part of Angular.
- The `label` "`for`" attribute and the `input` "`id`" attribute are to match, as per W3C standard[46]
- We set a `placeholder` of "SKU", which is just a hint to the user for what this `input` should say when it is blank

The `NgModel` directive specifies a `selector` of `ngModel`. This means we can attach it to our `input` tag by adding this sort of attribute: `ngModel="whatever"`. In this case, we specify `ngModel` with no attribute value.

There are a couple of different ways to specify `ngModel` in your templates and this is the first. When we use `ngModel` with no attribute value we are specifying:

1. a *one-way* data binding
2. we want to create a `FormControl` on this form with the name `sku` (because of the `name` attribute on the `input` tag)

`NgModel` **creates a new `FormControl`** that is **automatically added** to the parent `FormGroup` (in this case, on the form) and then binds a DOM element to that new `FormControl`. That is, it sets up an association between the `input` tag in our view and the `FormControl` and the association is matched by a name, in this case `"sku"`.

[45] http://semantic-ui.com/

[46] http://www.w3.org/TR/WCAG20-TECHS/H44.html

`NgModel` vs. `ngModel`: what's the difference? Generally, when we use PascalCase, like `NgModel`, we're specifying the *class* and referring to the object as it's defined in code. The lower case (CamelCase), as in `ngModel`, comes from the `selector` of the directive and it's only used in the DOM / template.

It's also worth pointing out that `NgModel` and `FormControl` are separate objects. `NgModel` is the *directive* that you use in your view, whereas `FormControl` is the object used for representing the data and validations in your form.

Sometimes we want to do *two-way* binding with `ngModel` like we used to do in Angular 1. We'll look at how to do that towards the end of this chapter.

Simple SKU Form: Component Definition Class

Now let's look at our class definition:

code/forms/src/app/demo-form-sku/demo-form-sku.component.ts

```
8  export class DemoFormSkuComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15   onSubmit(form: any): void {
16     console.log('you submitted value:', form);
17   }
18
19 }
```

Here our class defines one function: `onSubmit`. This is the function that is called when the form is submitted. For now, we'll just `console.log` out the value that is passed in.

Try it out!

Putting it all together, here's what our code listing looks like:

code/forms/src/app/demo-form-sku/demo-form-sku.component.ts

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

@Component({
  selector: 'app-demo-form-sku',
  templateUrl: './demo-form-sku.component.html',
  styles: []
})
export class DemoFormSkuComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }

}
```

and the template:

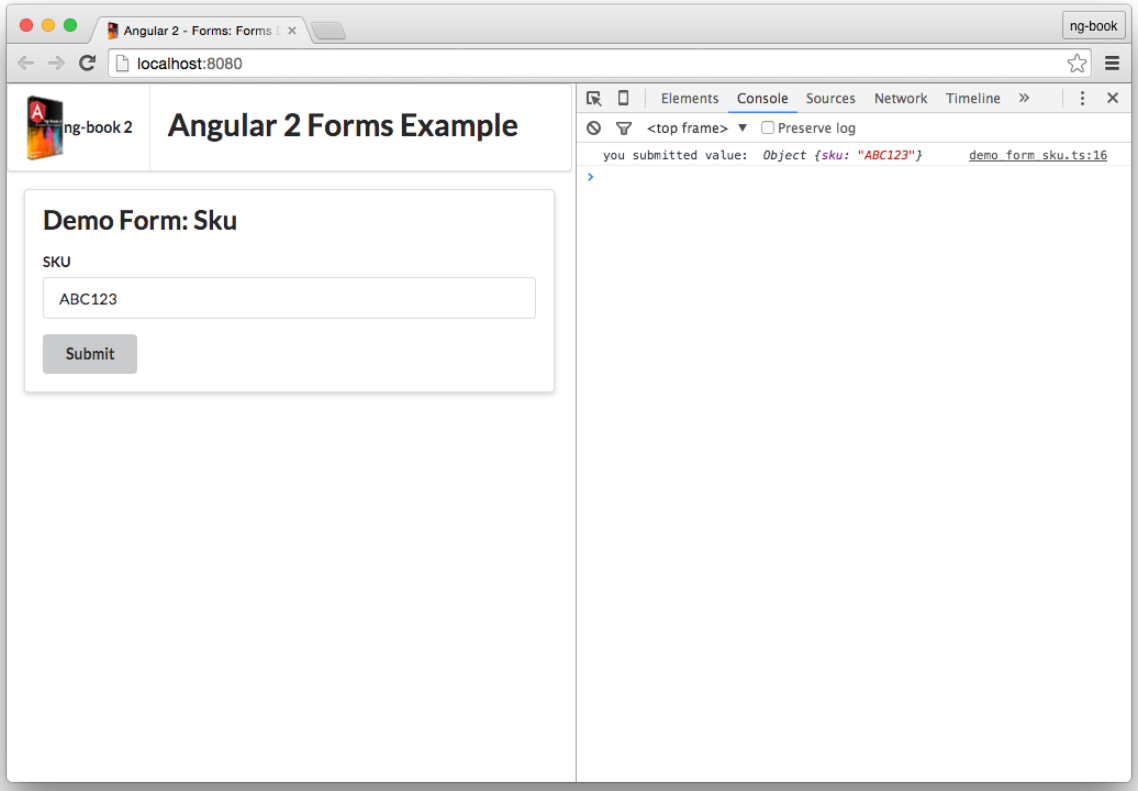
code/forms/src/app/demo-form-sku/demo-form-sku.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku</h2>
  <form #f="ngForm"
        (ngSubmit)="onSubmit(f.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             name="sku" ngModel>
    </div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

If we try this out in our browser, here's what it looks like:



Demo Form with Sku: Simple Version, Submitted

Using `FormBuilder`

Building our `FormControls` and `FormGroups` implicitly using `ngForm` and `ngControl` is convenient, but doesn't give us a lot of customization options. A more flexible and common way to configure forms is to use a `FormBuilder`.

`FormBuilder` is an aptly-named helper class that helps us build forms. As you recall, forms are made up of `FormControls` and `FormGroups` and the `FormBuilder` helps us make them (you can think of it as a "factory" object).

Let's add a `FormBuilder` to our previous example. Let's look at:

- how to use the `FormBuilder` in our component definition class
- how to use our custom `FormGroup` on a `form` in the view

Reactive Forms with `FormBuilder`

For this component we're going to be using the `formGroup` and `formControl` directives which means we need to import the appropriate classes. We start by importing them like so:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

Using `FormBuilder`

We inject `FormBuilder` by creating an argument in the `constructor` of our component class:

What does `inject` mean? We haven't talked much about dependency injection (DI) or how DI relates to the hierarchy tree, so that last sentence may not make a lot of sense. We talk a lot more about dependency injection in the Dependency Injection chapter, so go there if you'd like to learn more about it in depth.

At a high level, Dependency Injection is a way to tell Angular what dependencies this component needs to function properly.

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
```

```
17        'sku': ['ABC123']
18      });
19    }
20
21    ngOnInit() {
22    }
23
24    onSubmit(value: string): void {
25      console.log('you submitted value: ', value);
26    }
27
28  }
```

During injection an instance of `FormBuilder` will be created and we assign it to the `fb` variable (in the constructor).

There are two main functions we'll use on `FormBuilder`:

- `control` - creates a new `FormControl`
- `group` - creates a new `FormGroup`

Notice that we've setup a new *instance variable* called `myForm` on this class. (We could have just as easily called it `form`, but I want to differentiate between our `FormGroup` and the `form` we had before.)

`myForm` is typed to be a `FormGroup`. We create a `FormGroup` by calling `fb.group()`. `.group` takes an object of key-value pairs that specify the `FormControls` in this group.

In this case, we're setting up one control `sku`, and the value is `["ABC123"]` - this says that the default value of this control is `"ABC123"`. (You'll notice that is an array. That's because we'll be adding more configuration options there later.)

Now that we have `myForm` we need to use that in the view (i.e. we need to *bind* it to our `form` element).

Using `myForm` in the view

We want to change our `<form>` to use `myForm`. If you recall, in the last section we said that `ngForm` is applied for us automatically when we use `FormsModule`. We also mentioned that `ngForm` creates its own `FormGroup`. Well, in this case, we **don't** want to use an outside `FormGroup`. Instead we want to use our instance variable `myForm`, which we created with our `FormBuilder`. How can we do that?

Angular provides another directive that we use **when we have an existing `FormGroup`**: it's called `formGroup` and we use it like this:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
2   <h2 class="ui header">Demo Form: Sku with Builder</h2>
3   <form [formGroup]="myForm"
```

Here we're telling Angular that we want to use `myForm` as the `FormGroup` for this form.

Remember how earlier we said that when using `FormsModule` that `NgForm` will be automatically applied to a `<form>` element? There is an exception: `NgForm` won't be applied to a `<form>` that has `formGroup`.

If you're curious, the `selector` for `NgForm` is:

```
1   form:not([ngNoForm]):not([formGroup]),ngForm,[ngForm]
```

This means you *could* have a form that doesn't get `NgForm` applied by using the `ngNoForm` attribute.

We also need to change `onSubmit` to use `myForm` instead of `f`, because now it is `myForm` that has our configuration and values.

There's one last thing we need to do to make this work: bind our `FormControl` to the `input` tag. Remember that **`ngControl` creates a new `FormControl` object**, and attaches it to the parent `FormGroup`. But in this case, we used `FormBuilder` to create our own `FormControls`.

When we want to bind an **existing `FormControl`** to an `input` we use `formControl`:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
8       <label for="skuInput">SKU</label>
9       <input type="text"
10             id="skuInput"
11             placeholder="SKU"
12             [formControl]="myForm.controls['sku']">
```

Here we are instructing the `formControl` directive to look at `myForm.controls` and use the existing `sku` `FormControl` for this `input`.

Try it out!

Here's what it looks like all together:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.ts

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

@Component({
  selector: 'app-demo-form-sku-with-builder',
  templateUrl: './demo-form-sku-with-builder.component.html',
  styles: []
})
export class DemoFormSkuWithBuilderComponent implements OnInit {
  myForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  ngOnInit() {
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }

}
```

and the template:

code/forms/src/app/demo-form-sku-with-builder/demo-form-sku-with-builder.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: Sku with Builder</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form">

    <div class="field">
      <label for="skuInput">SKU</label>
```

```
9         <input type="text"
10                id="skuInput"
11                placeholder="SKU"
12                [formControl]="myForm.controls['sku']">
13      </div>
14
15    <button type="submit" class="ui button">Submit</button>
16    </form>
17  </div>
```

Remember:

To create a new `FormGroup` and `FormControls` implicitly use:

- `ngForm` and
- `ngModel`

But to bind to an existing `FormGroup` and `FormControls` use:

- `formGroup` and
- `formControl`

Adding Validations

Our users aren't always going to enter data in exactly the right format. If someone enters data in the wrong format, we want to give them feedback and not allow the form to be submitted. For this we use *validators*.

Validators are provided by the `Validators` module and the simplest validator is `Validators.required` which simply says that the designated field is required or else the `FormControl` will be considered invalid.

To use validators we need to do two things:

1. Assign a validator to the `FormControl` object
2. Check the status of the validator in the view and take action accordingly

To assign a validator to a `FormControl` object we simply pass it as the second argument to our `FormControl` constructor:

```
let control = new FormControl('sku', Validators.required);
```

Or in our case, because we're using `FormBuilder` we will use the following syntax:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

```
  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }
```

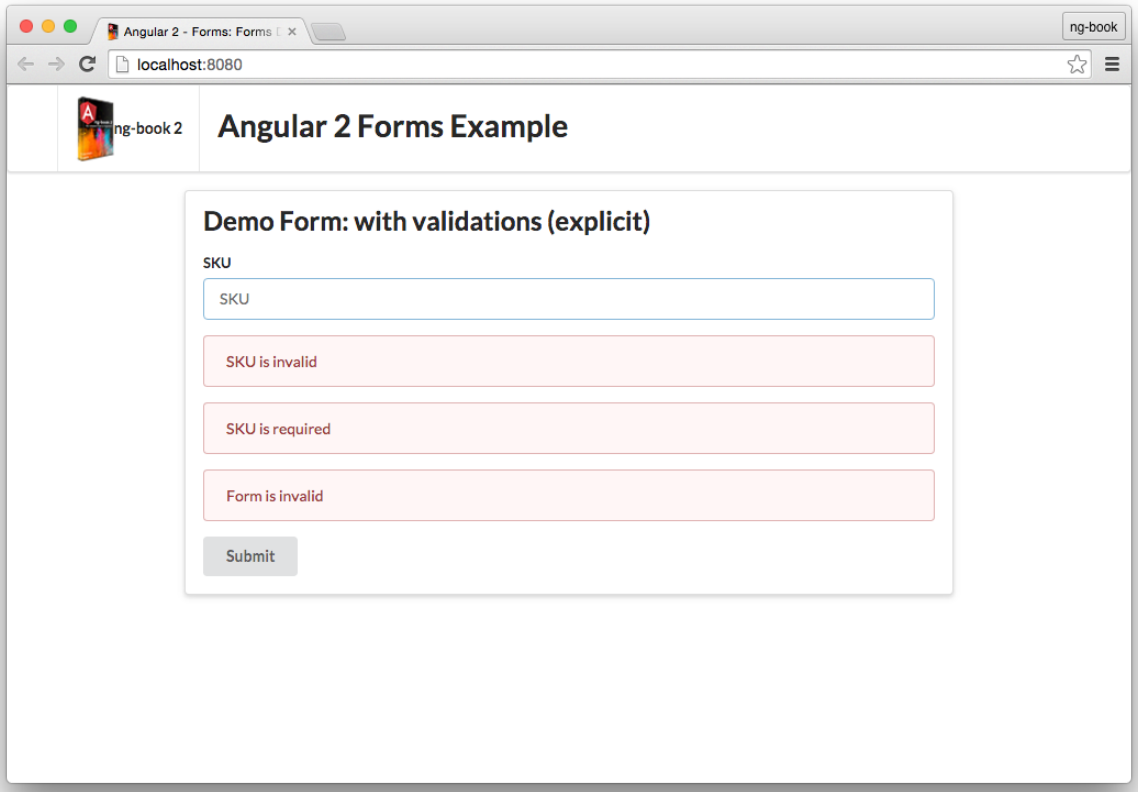
Now we need to use our validation in the view. There are two ways we can access the validation value in the view:

1. We can explicitly assign the `FormControl` `sku` to an instance variable of the class - which is more verbose, but gives us easy access to the `FormControl` in the view.
2. We can lookup the `FormControl` `sku` from `myForm` in the view. This requires less work in the component definition class, but is slightly more verbose in the view.

To make this difference clearer, let's look at this example both ways:

Explicitly setting the `sku` `FormControl` as an instance variable

Here's a screenshot of what our form is going to look like with validations:



Demo Form with Validations

The most flexible way to deal with individual `FormControls` in your view is to set each `FormControl` up as an instance variable in your component definition class. Here's how we could setup `sku` in our class:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

```
14 export class DemoFormWithValidationsExplicitComponent {
15   myForm: FormGroup;
16   sku: AbstractControl;
17
18   constructor(fb: FormBuilder) {
19     this.myForm = fb.group({
20       'sku':  ['', Validators.required]
21     });
22
23     this.sku = this.myForm.controls['sku'];
24   }
25
```

```
26    onSubmit(value: string): void {
27      console.log('you submitted value: ', value);
28    }
29
30  }
```

Notice that:

1. We setup `sku: AbstractControl` at the top of the class and
2. We assign `this.sku` after we've created `myForm` with the `FormBuilder`

This is great because it means we can reference `sku` anywhere in our component view. The downside is that by doing it this way, we'd have to setup an instance variable **for every field in our form**. For large forms, this can get pretty verbose.

Now that we have our `sku` being validated, I want to look at four different ways we can use it in our view:

1. Checking the validity of our whole form and displaying a message
2. Checking the validity of our individual field and displaying a message
3. Checking the validity of our individual field and coloring the field red if it's invalid
4. Checking the validity of our individual field on a particular requirement and displaying a message

Form message

We can check the validity of our whole form by looking at `myForm.valid`:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
20      <div *ngIf="!myForm.valid"
```

Remember, `myForm` is a `FormGroup` and a `FormGroup` is valid if all of the children `FormControls` are also valid.

Field message

We can also display a message for the specific field if that field's `FormControl` is invalid:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
14              [formControl]="sku">
15        <div *ngIf="!sku.valid"
16          class="ui error message">SKU is invalid</div>
17        <div *ngIf="sku.hasError('required')"
```

Field coloring

I'm using the Semantic UI CSS Framework's CSS class `.error`, which means if I add the class `error` to the `<div class= "field">` it will show the input tag with a red border.

To do this, we can use the property syntax to set conditional classes:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
7     <div class="field"
8         [class.error]="!sku.valid && sku.touched">
```

Notice here that we have two conditions for setting the `.error` class: We're checking for `!sku.valid` and `sku.touched`. The idea here is that we only want to show the error state if the user has tried editing the form ("touched" it) and it's now invalid.

To try this out, enter some data into the `input` tag and then delete the contents of the field.

Specific validation

A form field can be invalid for many reasons. We often want to show a different message depending on the reason for a failed validation.

To look up a specific validation failure we use the `hasError` method:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
17        <div *ngIf="sku.hasError('required')"
18          class="ui error message">SKU is required</div>
```

Note that `hasError` is defined on both `FormControl` and `FormGroup`. This means you can pass a second argument of `path` to lookup a specific field from `FormGroup`. For example, we could have written the previous example as:

```
      <div *ngIf="myForm.hasError('required', 'sku')"
        class="error">SKU is required</div>
```

Putting it together

Here's the full code listing of our form with validations with the `FormControl` set as an instance variable:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.ts

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

@Component({
  selector: 'app-demo-form-with-validations-explicit',
  templateUrl: './demo-form-with-validations-explicit.component.html',
  styles: []
})
export class DemoFormWithValidationsExplicitComponent {
  myForm: FormGroup;
  sku: AbstractControl;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku':  ['', Validators.required]
    });

    this.sku = this.myForm.controls['sku'];
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }

}
```

And the template:

code/forms/src/app/demo-form-with-validations-explicit/demo-form-with-validations-explicit.component.html

```
<div class="ui raised segment">
  <h2 class="ui header">Demo Form: with validations (explicit)</h2>
  <form [formGroup]="myForm"
        (ngSubmit)="onSubmit(myForm.value)"
        class="ui form"
        [class.error]="!myForm.valid && myForm.touched">

    <div class="field"
        [class.error]="!sku.valid && sku.touched">
      <label for="skuInput">SKU</label>
      <input type="text"
             id="skuInput"
             placeholder="SKU"
             [formControl]="sku">
       <div *ngIf="!sku.valid"
         class="ui error message">SKU is invalid</div>
       <div *ngIf="sku.hasError('required')"
         class="ui error message">SKU is required</div>
    </div>

    <div *ngIf="!myForm.valid"
      class="ui error message">Form is invalid</div>

    <button type="submit" class="ui button">Submit</button>
  </form>
</div>
```

Removing the `sku` instance variable

In the example above we set `sku: AbstractControl` as an instance variable. We often won't want to create an instance variable for each `AbstractControl`, so how would we reference this `FormControl` in our view without an instance variable?

Instead we can use the `myForm.controls` property as in:

code/forms/src/app/demo-form-with-validations-shorthand/demo-form-with-validations-shorthand.component.html

```
10        <label for="skuInput">SKU</label>
11        <input type="text"
12               id="skuInput"
13               placeholder="SKU"
14               [formControl]="myForm.controls['sku']">
15         <div *ngIf="!myForm.controls['sku'].valid"
16           class="ui error message">SKU is invalid</div>
17         <div *ngIf="myForm.controls['sku'].hasError('required')"
```

In this way we can access the `sku` control without being forced to explicitly add it as an instance variable on the component class.

We used bracket-notation, e.g. `myForm.controls['sku']`. We could also use the dot-notation, e.g `myForm.controls.sku`. In general, be aware that TypeScript may give a warning if you use the dot-notation and the object is not properly typed (but that is not a problem here).

Custom Validations

We often are going to want to write our own custom validations. Let's take a look at how to do that.

To see how validators are implemented, let's look at `Validators.required` from the Angular core source:

```
export class Validators {
  static required(c: FormControl): StringMap<string, boolean> {
    return isBlank(c.value) || c.value == "" ? {"required": true} : null;
  }
```

A validator: - Takes a `FormControl` as its input and - Returns a `StringMap<string, boolean>` where the key is "error code" and the value is `true` if it fails

Writing the Validator

Let's say we have specific requirements for our `sku`. For example, say our `sku` needs to begin with `123`. We could write a validator like so:

code/forms/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.ts

```
18 function skuValidator(control: FormControl): { [s: string]: boolean } {
19   if (!control.value.match(/^123/)) {
20     return {invalidSku: true};
21   }
22 }
```

This validator will return an error code `invalidSku` if the input (the `control.value`) does not begin with `123`.

Assigning the Validator to the `FormControl`

Now we need to add the validator to our `FormControl`. However, there's one small problem: we already have a validator on `sku`. How can we add multiple validators to a single field?

For that, we use `Validators.compose`:

code/forms/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.ts

```
33   constructor(fb: FormBuilder) {
34     this.myForm = fb.group({
35       'sku':  ['', Validators.compose([
36         Validators.required, skuValidator])]
37     });
```

`Validators.compose` wraps our two validators and lets us assign them both to the `FormControl`. The `FormControl` is not `valid` unless both validations are valid.

Now we can use our new validator in the view:

code/forms/src/app/demo-form-with-custom-validation/demo-form-with-custom-validation.component.html

```
19       <div *ngIf="sku.hasError('invalidSku')"
20         class="ui error message">SKU must begin with <span>123</span></div>
```

Note that in this section, I'm using "explicit" notation of adding an instance variable for each `FormControl`. That means that in the view in this section, `sku` refers to a `FormControl`.

If you run the sample code, one neat thing you'll notice is that if you type something in to the field, the `required` validation will be fulfilled, but the `invalidSku` validation may not. This is great - it means we can partially-validate our fields and show the appropriate messages.

Watching For Changes

So far we've only extracted the value from our form by calling `onSubmit` when the form is submitted. But often we want to watch for any value changes on a control.

Both `FormGroup` and `FormControl` have an `EventEmitter` that we can use to observe changes.

`EventEmitter` is an *Observable*, which means it conforms to a defined specification for watching for changes. If you're interested in the Observable spec, you can find it here[47]

To watch for changes on a control we:

1. get access to the `EventEmitter` by calling `control.valueChanges`. Then we
2. add an *observer* using the `.subscribe` method

Here's an example:

code/forms/src/app/demo-form-with-events/demo-form-with-events.component.ts

```
21  constructor(fb: FormBuilder) {
22    this.myForm = fb.group({
23      'sku':  ['', Validators.required]
24    });
25
26    this.sku = this.myForm.controls['sku'];
27
28    this.sku.valueChanges.subscribe(
29      (value: string) => {
30        console.log('sku changed to:', value);
31      }
32    );
33
34    this.myForm.valueChanges.subscribe(
35      (form: any) => {
36        console.log('form changed to:', form);
37      }
38    );
39
40  }
```

[47] https://github.com/jhusain/observable-spec

Here we're observing two separate events: changes on the sku field and changes on the form as a whole.

The observable that we pass in is an object with a single key: `next` (there are other keys you can pass in, but we're not going to worry about those now). `next` is the function we want to call with the new value whenever the value changes.

If we type '`kj`' into the text box we will see in our console:

```
sku changed to:  k
form changed to:  Object {sku: "k"}
sku changed to:  kj
form changed to:  Object {sku: "kj"}
```

As you can see each keystroke causes the control to change, so our observable is triggered. When we observe the individual `FormControl` we receive a value (e.g. `kj`), but when we observe the whole form, we get an object of key-value pairs (e.g. `{sku: "kj"}`).

ngModel

`NgModel` is a special directive: it binds a model to a form. `ngModel` is special in that it **implements two-way data binding**. Two-way data binding is almost always more complicated and difficult to reason about vs. one-way data binding. Angular is built to generally have data flow one-way: top-down. However, when it comes to forms, there are times where it is easier to opt-in to a two-way bind.

Just because you've used `ng-model` in Angular 1 in the past, don't rush to use `ngModel` right away. There are good reasons to avoid two-way data binding[48]. Of course, `ngModel` can be really handy, but know that we don't necessarily rely on two-way data binding as much as we did in Angular 1.

Let's change our form a little bit and say we want to input `productName`. We're going to use `ngModel` to keep the component instance variable in sync with the view.

First, here's our component definition class:

[48] https://www.quora.com/Why-is-the-two-way-data-binding-being-dropped-in-Angular-2

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.ts

```
12 export class DemoFormNgModelComponent {
13   myForm: FormGroup;
14   productName: string;
15
16   constructor(fb: FormBuilder) {
17     this.myForm = fb.group({
18       'productName':  ['', Validators.required]
19     });
20   }
21
22   onSubmit(value: string): void {
23     console.log('you submitted value: ', value);
24   }
25 }
```

Notice that we're simply storing `productName: string` as an instance variable.

Next, let's use `ngModel` on our `input` tag:

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.html

```
13       <label for="productNameInput">Product Name</label>
14       <input type="text"
15              id="productNameInput"
16              placeholder="Product Name"
17              [formControl]="myForm.get('productName')"
18              [(ngModel)]="productName">
```

Now notice something - the syntax for `ngModel` is funny: we are using both brackets and parentheses around the `ngModel` attribute! The idea this is intended to invoke is that we're using both the *input* `[]` brackets and the *output* `()` parentheses. It's an indication of the two-way bind.

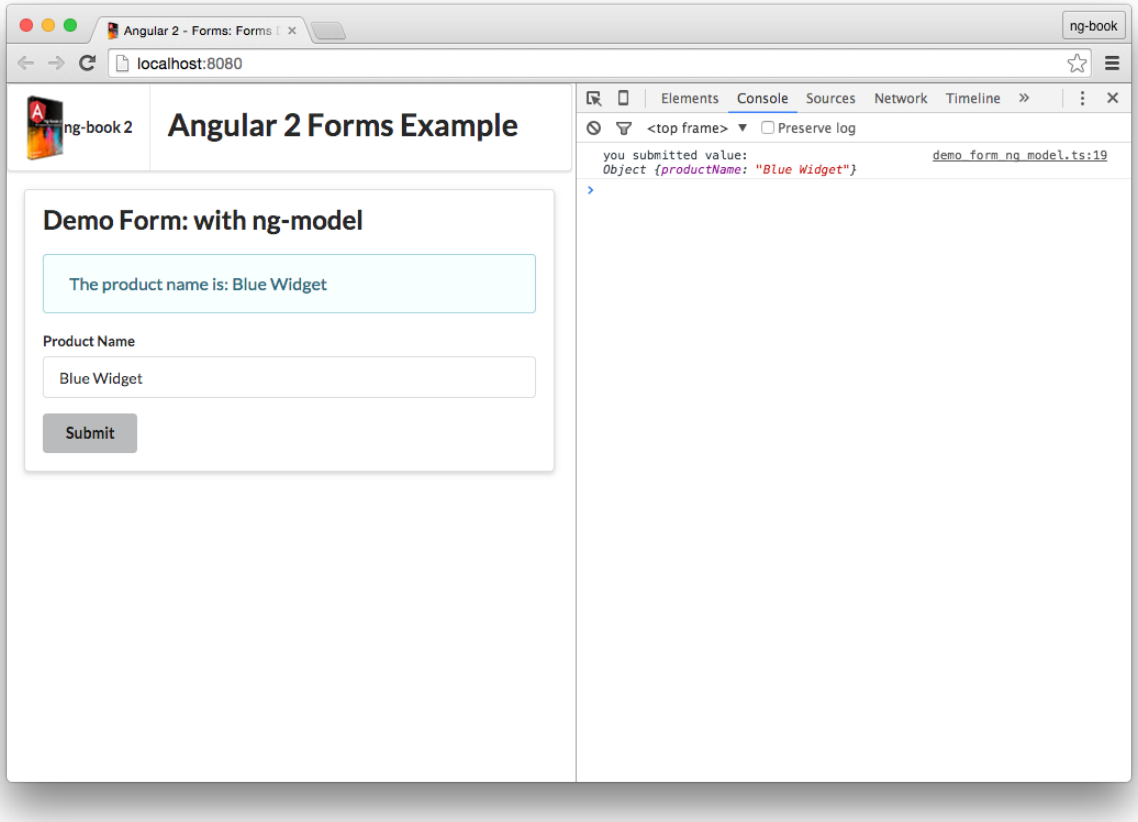
Notice something else here: we're still using `formControl` to specify that this `input` should be bound to the `FormControl` on our form. We do this because `ngModel` is only binding the `input` to the instance variable - the `FormControl` is completely separate. But because we still want to validate this value and submit it as part of the form, we keep the `formControl` directive.

Last, let's display our `productName` value in the view:

code/forms/src/app/demo-form-ng-model/demo-form-ng-model.component.html

```
4   <div class="ui info message">
5     The product name is: {{productName}}
6   </div>
```

Here's what it looks like:



Demo Form with ngModel

Easy!

Wrapping Up

Forms have a lot of moving pieces, but Angular makes it fairly straightforward. Once you get a handle on how to use `FormGroups`, `FormControls`, and `Validations`, it's pretty easy going from there!