**Dynamic forms** is actually a **pattern** (not another API to build forms as some may think) in which we build a form based on **meta description**, and we use the **reactive form API** in order to achieve it.

The Reactive Forms API gives us some really useful tools to **build** the form, **access** its **data model** via accessing the form-group **controls** (directly via controls or using a getter of a certain control) etc and **define** its **validators** all via the code.

Dynamic forms are based on reactive forms. To give the application access reactive forms directives, the root module imports ReactiveFormsModule from the @angular/forms library.

create a /dynamic-form/ directory inside of /app.

\*\*/app/dynamic-form

Then create a file called dynamic-form.module.ts. To start, it will look like this:

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [CommonModule, ReactiveFormsModule],

})

export class DynamicFormModule {}

The final thing we need to do with the module (for now), is import it into our AppModule inside /app/app.module.ts:

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { DynamicFormModule } from './dynamic-form/dynamic-form.module';

@NgModule({

imports: [BrowserModule, DynamicFormModule],

bootstrap: [AppComponent],

declarations: [AppComponent],

})

export class AppModule {}

Now we need to create the container that will be used to make a dynamic form!

**The main container**

The point of entry for our dynamic form is the main container. This will be the only component that is exposed by our dynamic forms module, being responsible for accepting a form configuration and creating the form.

Create a directory inside of the /dynamic-form directory you’ve just made called /containers. Inside of that, create a directory called /dynamic-form.

\*\*/app/dynamic-form/containers/dynamic-form

Inside of that directory, create a component file called dynamic-form.component.ts.

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

import { FormGroup, FormBuilder } from '@angular/forms';

@Component({

selector: 'dynamic-form',

styleUrls: ['dynamic-form.component.scss'],

template: `

<form

class="dynamic-form"

[formGroup]="form">

</form>

`

})

export class DynamicFormComponent implements OnInit {

@Input()

config: any[] = [];

form: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.form = this.createGroup();

}

createGroup() {

const group = this.fb.group({});

this.config.forEach(control =&gt; group.addControl(control.name, this.fb.control()));

return group;

}

}

As our form is *dynamic*, we need to accept a configuration array in order to know what to create. To do this, we’re using an @Input() that accepts any array of objects.

We are also utilising the power of Angular’s reactive forms. This allows us to easily link all of our dynamically created fields into one form group, giving us access to the value object. We could also expand on our implementation to allow the validation to be configured, for example.

For each item in the configuration, we’re going to expect that the object contains *at least* two properties - type and name. This tells us what the type of the field is (input, select, button, etc) as well as what it’s called.

Inside createGroup, we loop through these items and create a new control for each one. We then add these dynamically created controls to the form group, ready for consumption by our dynamic fields.

Let’s declare and export this component inside of our DynamicFormModule:

// ...

import { DynamicFormComponent } from './containers/dynamic-form/dynamic-form.component';

@NgModule({

imports: [CommonModule, ReactiveFormsModule],

declarations: [DynamicFormComponent],

exports: [DynamicFormComponent],

})

export class DynamicFormModule {}

Now that we’ve created the form, let’s actually use it!

**Using the dynamic form**

Open up \*\*/app/app.component.ts. Inside of the <div>, we’re going to use ``, and pass in a configuration object:

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

styleUrls: ['app.component.scss'],

template: `

<div class="app">

<dynamic-form [config]="config"></dynamic-form>

</div>

`,

})

export class AppComponent {

config = [

{

type: 'input',

label: 'Full name',

name: 'name',

placeholder: 'Enter your name',

},

{

type: 'select',

label: 'Favourite food',

name: 'food',

options: ['Pizza', 'Hot Dogs', 'Knakworstje', 'Coffee'],

placeholder: 'Select an option',

},

{

label: 'Submit',

name: 'submit',

type: 'button',

},

];

}

You can see that we’re passing through the config array that we’ve defined in our AppComponent class through to the dynamic form component.

This is an array of objects that contain information about the fields that we want in our form. Different types of fields have different properties:

* “name” is a basic input, that has a placeholder and a label above it
* “food” is a select dropdown, which has an array of options for the user to select as well as a placeholder and label
* “submit” is a button so we can submit the form

Let’s get the ball rolling and create components for each type of field we have (input, select and button).

**Input field**

For our components we need a /components directory. This will be at the same level as our /containers directory, inside /dynamic-form.

\*\*/app/dynamic-form/components

Next, create three folders - /form-input, /form-select and /form-button.

\*\*/app/dynamic-form/components/form-input/

\*\*/app/dynamic-form/components/form-select/

\*\*/app/dynamic-form/components/form-button/

We’ll start with the input field. Inside of the /form-input directory, create a component file named form-input.component.ts.

Each of our fields will need to receive two inputs - the configuration for that field (such as placeholder, label, etc) and the FormGroup from our DynamicFormComponent.

To start off with, our FormInputComponent will look like this:

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

import { FormGroup } from '@angular/forms';

@Component({

selector: 'form-input',

styleUrls: ['form-input.component.scss'],

template: `

<div

class="dynamic-field form-input"

[formGroup]="group">

<label>{{ config.label }}</label>

<input

type="text"

[attr.placeholder]="config.placeholder"

[formControlName]="config.name" />

</div>

`,

})

export class FormInputComponent {

config;

group: FormGroup;

}

You can see we’ve set two properties on the class for the config and group. However, these aren’t using @Input() like you would expect, because we aren’t going to be using this component in the traditional way.

Our select and button components are very similar…

**Select field**

For select, you guessed it - we need to create a component file called form-select.component.ts inside /form-select. This will look like this:

import { Component } from '@angular/core';

import { FormGroup } from '@angular/forms';

@Component({

selector: 'form-select',

styleUrls: ['form-select.component.scss'],

template: `

<div

class="dynamic-field form-select"

[formGroup]="group">

<label>{{ config.label }}</label>

<select [formControlName]="config.name">

<option value="">{{ config.placeholder }}</option>

<option \*ngFor="let option of config.options">

{{ option }}

</option>

</select>

</div>

`,

})

export class FormSelectComponent implements Field {

config;

group: FormGroup;

}

The main difference here is that we’re looping over the options property that we defined in the configuration above. This displays all the options to the user, and we’re also adding an empty property above all of these with our placeholder property, indicating to the user that they need to select an option.

**Button**

The button is very simple - create a component file called form-button.component.ts inside /form-button, and fill it with this:

import { Component } from '@angular/core';

import { FormGroup } from '@angular/forms';

@Component({

selector: 'form-button',

styleUrls: ['form-button.component.scss'],

template: `

<div

class="dynamic-field form-button"

[formGroup]="group">

<button type="submit">

{{ config.label }}

</button>

</div>

`,

})

export class FormButtonComponent implements Field {

config;

group: FormGroup;

}

This is just displaying a simple button, with the config.label as the button’s text.

As with all components, we need to declare these inside of the module we created earlier. Open up dynamic-form.module.ts and add these as declarations:

// ...

import { FormButtonComponent } from './components/form-button/form-button.component';

import { FormInputComponent } from './components/form-input/form-input.component';

import { FormSelectComponent } from './components/form-select/form-select.component';

@NgModule({

// ...

declarations: [

DynamicFormComponent,

FormButtonComponent,

FormInputComponent,

FormSelectComponent,

],

exports: [DynamicFormComponent],

})

export class DynamicFormModule {}

**DynamicField**

We’ve got our three components so far that we want to dynamically create - FormInputComponent, FormSelectComponent and FormButtonComponent.

To create these, we’re going to use a directive. This is going to act pretty similarly to the router-outlet directive. There’s no need for a view (hence we’re using a directive), and we’re going to create the components *next* to our directive in the DOM.

Inside of the /components directory, create a directory named /dynamic-field.

\*\*/app/dynamic-form/components/dynamic-field

Inside of this, create a directive file named dynamic-field.directive.ts. Let’s build this directive piece by piece. To start, we’ll lay the foundations:

import { Directive, Input } from '@angular/core';

import { FormGroup } from '@angular/forms';

@Directive({

selector: '[dynamicField]',

})

export class DynamicFieldDirective {

@Input() config;

@Input() group: FormGroup;

}

We’ve set the selector to [dynamicField] as we’re going to use it as an attribute rather than an element.

The advantage of this is that we can actually use this on a built-in Angular directive called ng-container. The ng-container will render out to be invisible in the DOM, therefore when we dynamically create our components, we’ll only see them in DOM rather than a load of <dynamic-field></dynamic-field> elements too.

We’ve added two @Input() bindings to our directive. These are the config and group that we’re going to pass down to our dynamic field components.

Let’s start dynamically rendering components!

There are two providers that we need to dynamically render components - ComponentFactoryResolver and ViewContainerRef. We covered ViewContainerRef above, and you can probably guess what ComponentFactoryResolver does - resolves the component factories that Angular has created for each component.

Let’s add these to our constructor:

import { ComponentFactoryResolver, Directive, Input, OnInit, ViewContainerRef } from '@angular/core';

import { FormGroup } from '@angular/forms';

@Directive({

selector: '[dynamicField]'

})

export class DynamicFieldDirective implements OnInit {

@Input()

config;

@Input()

group: FormGroup;

constructor(

private resolver: ComponentFactoryResolver,

private container: ViewContainerRef

) {}

ngOnInit() {

}

}

I’ve also added the ngOnInit lifecycle hook, ready for us to start creating our dynamic components.

To resolve a component factory, we need to pass through the component class that the component factory was generated from, e.g. FormInputComponent.

As we are allowing the type of the field to be dictated by a string ('input', 'select', etc), we need to create a lookup object to map the strings over to their relevant component class:

// ...

import { FormButtonComponent } from '../form-button/form-button.component';

import { FormInputComponent } from '../form-input/form-input.component';

import { FormSelectComponent } from '../form-select/form-select.component';

const components = {

button: FormButtonComponent,

input: FormInputComponent,

select: FormSelectComponent

};

@Directive(...)

export class DynamicFieldDirective implements OnInit {

// ...

}

For example, this will allow us to access components['button'] and receive FormButtonComponent back, which we can then pass to the component factory resolver to get the component factory:

// ...

const components = {

button: FormButtonComponent,

input: FormInputComponent,

select: FormSelectComponent

};

@Directive(...)

export class DynamicFieldDirective implements OnInit {

// ...

ngOnInit() {

const component = components[this.config.type];

const factory = this.resolver.resolveComponentFactory<any>(component);

}

// ...

}

That’s all it takes! We’ve now referenced the component for the given type defined in the configuration, and passed that through to the resolveComponentFactory method that ComponentFactoryRsolver provides.

You might have noticed that we’re using <any> next to resolveComponentFactory. This is to tell TypeScript the type of our component so we can get information about the properties we can set later on (these will be config and group).

As we’re creating multiple different components, we’re just going to set this to any. We could use an interface here instead, and have the dynamic components implement that for peace of mind.

Now that we’ve got the component factory, we can simply tell our ViewContainerRef to create this component for us:

// ...

@Directive(...)

export class DynamicFieldDirective implements OnInit {

// ...

component;

ngOnInit() {

const component = components[this.config.type];

const factory = this.resolver.resolveComponentFactory<any>(component);

this.component = this.container.createComponent(factory);

}

// ...

}

We’re setting this to a property on the class called component - this is so we can access the component in other methods if needed. For instance, we could add ngOnChanges to keep the dynamic component in-sync with the config and group passed down to DynamicFieldDirective.

We can now pass the config and group into our dynamically created component. These are just properties on the component class, and we can access the initialised component class via this.component.instance:

// ...

@Directive(...)

export class DynamicFieldDirective implements OnInit {

// ...

component;

ngOnInit() {

const component = components[this.config.type];

const factory = this.resolver.resolveComponentFactory<any>(component);

this.component = this.container.createComponent(factory);

this.component.instance.config = this.config;

this.component.instance.group = this.group;

}

// ...

}

Let’s go ahead and declare this in our module too:

// ...

import { DynamicFieldDirective } from './components/dynamic-field/dynamic-field.directive';

@NgModule({

// ...

declarations: [

DynamicFieldDirective,

DynamicFormComponent,

FormButtonComponent,

FormInputComponent,

FormSelectComponent,

],

exports: [DynamicFormComponent],

})

export class DynamicFormModule {}

We’re nearly there, however if you ran this in your browser now, you’d get an error.

When we want a component to be able to be created dynamically, we need to let Angular know so it can expose the component factories for us. To do this, we can utilise a property inside our @NgModule() configuration - entryComponents. This is an array of components that Angular will expose to us.

// ...

@NgModule({

// ...

declarations: [

DynamicFieldDirective,

DynamicFormComponent,

FormButtonComponent,

FormInputComponent,

FormSelectComponent,

],

exports: [DynamicFormComponent],

entryComponents: [

FormButtonComponent,

FormInputComponent,

FormSelectComponent,

],

})

export class DynamicFormModule {}

**Looping through the fields**

We’ve got our dynamic field components sorted, as well as the directive to render them. Now all we need to do is hook this up to our DynamicFormComponent:

// ...

@Component({

selector: 'dynamic-form',

styleUrls: ['dynamic-form.component.scss'],

template: `

<form

class="dynamic-form"

[formGroup]="form">

<ng-container

\*ngFor="let field of config;"

dynamicField

[config]="field"

[group]="form">

</ng-container>

</form>

`,

})

export class DynamicFormComponent implements OnInit {

// ...

}

As we mentioned earlier, we’re using ng-container as the element to repeat for our dynamic fields. This is invisible when our component is rendered, meaning that we will only see our dynamic field elements in the DOM.

We’re adding an ngFor to this container, repeating it for each configuration item.

Next is where we hook up our dynamic field directive. We set the attribute dynamicField on our container, which tells Angular to go ahead and run our directive for this element - binding the correct context of this.config and this.group to each directive instance.

The context is bound when we pass in the two @Input bindings that our directive needs - the configuration object for that field, and the form group for the form.

**Submitting the form**

The next thing we need to do is implement the submit functionality.

All we need to do is add a handler for the (ngSubmit) event on our <form> component, and add an @Output() to our DynamicFormComponent so we can notify the component that uses it.

Open up /app/dynamic-form/containers/dynamic-form.component.ts:

import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

// ...

@Component({

selector: 'dynamic-form',

styleUrls: ['dynamic-form.component.scss'],

template: `

<form

class="dynamic-form"

[formGroup]="form"

(ngSubmit)="submitted.emit(form.value)">

<ng-container

\*ngFor="let field of config;"

dynamicField

[config]="field"

[group]="form">

</ng-container>

</form>

`

})

export class DynamicFormComponent implements OnInit {

// ...

@Output()

submitted: EventEmitter = new EventEmitter();

}

Here we’ve added an @Output() named submitted so we can notify the parent component when the form is submitted.

We’re emitting to this output directly inside the (ngSubmit) event, passing through form.value so the parent component will receive the value of all the fields.

Let’s add this to our app.component.ts file:

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

styleUrls: ['app.component.scss'],

template: `

<div class="app">

<dynamic-form

[config]="config"

(submitted)="formSubmitted($event)">

</dynamic-form>

</div>

`,

})

export class AppComponent {

// ...

formSubmitted(value) {

console.log(value);

}

}

Now when we submit our form, formSubmitted will be fired with an object containing the value of the fields in our form!