Angular - Maatwerk

Peter Kassenaar
info@kassenaar.com

Forms & Validation

On Forms, Controls and Validation

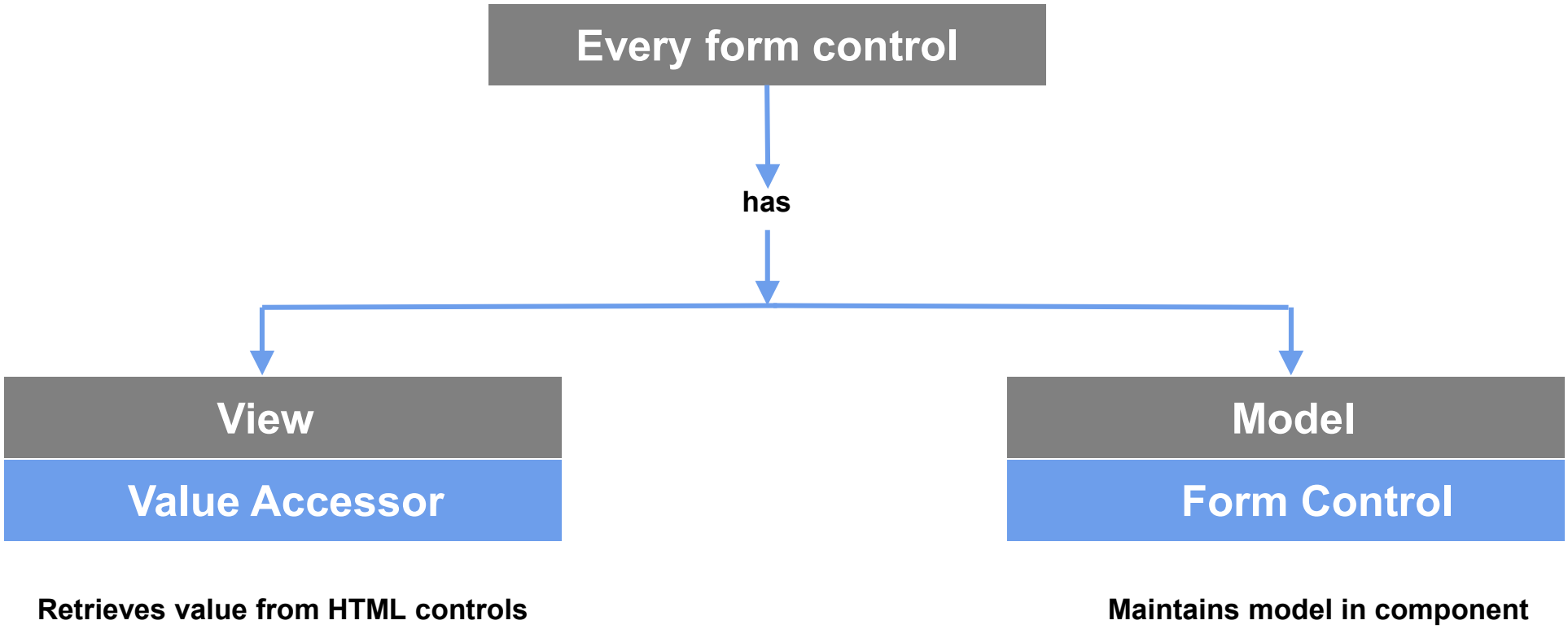
Forms contents

- Types of forms
- Form control Architecture
- Types of Form Controls
- State in Form Controls
- Form Arrays
  - Dynamic Forms

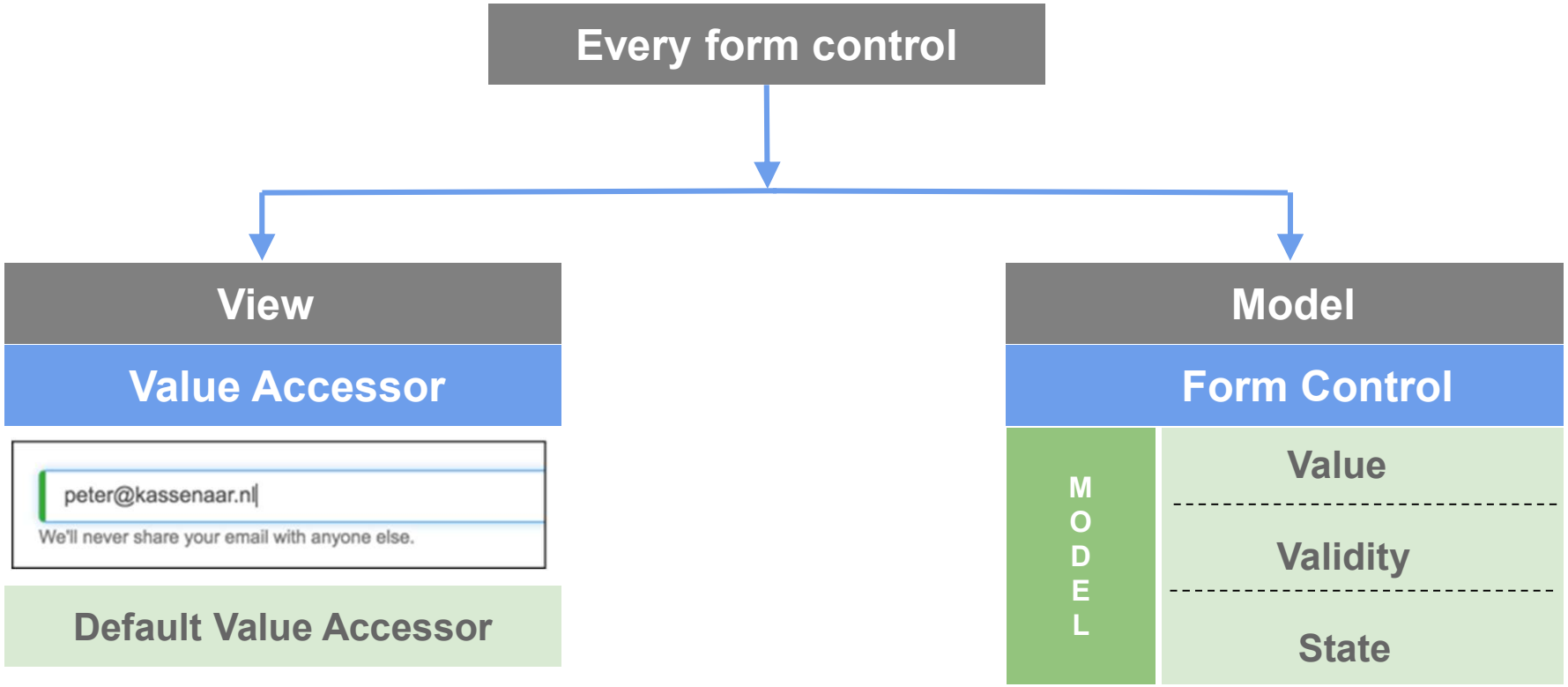
Angular 2 – Types of Forms

Template Driven Forms	Model Driven (Reactive Forms)
• Source of truth is the Template	• Source of truth is the component class / directive
• Define templates. Angular generates form model o/t fly	• Instantiate Form model and Control model yourself
• Less descriptive	• More Descriptive
• Quickly Build simple forms – Less control	• Code all the details. Takes more time, gives more control
• Less testable	• Very good testable

Angular Form Control Architecture

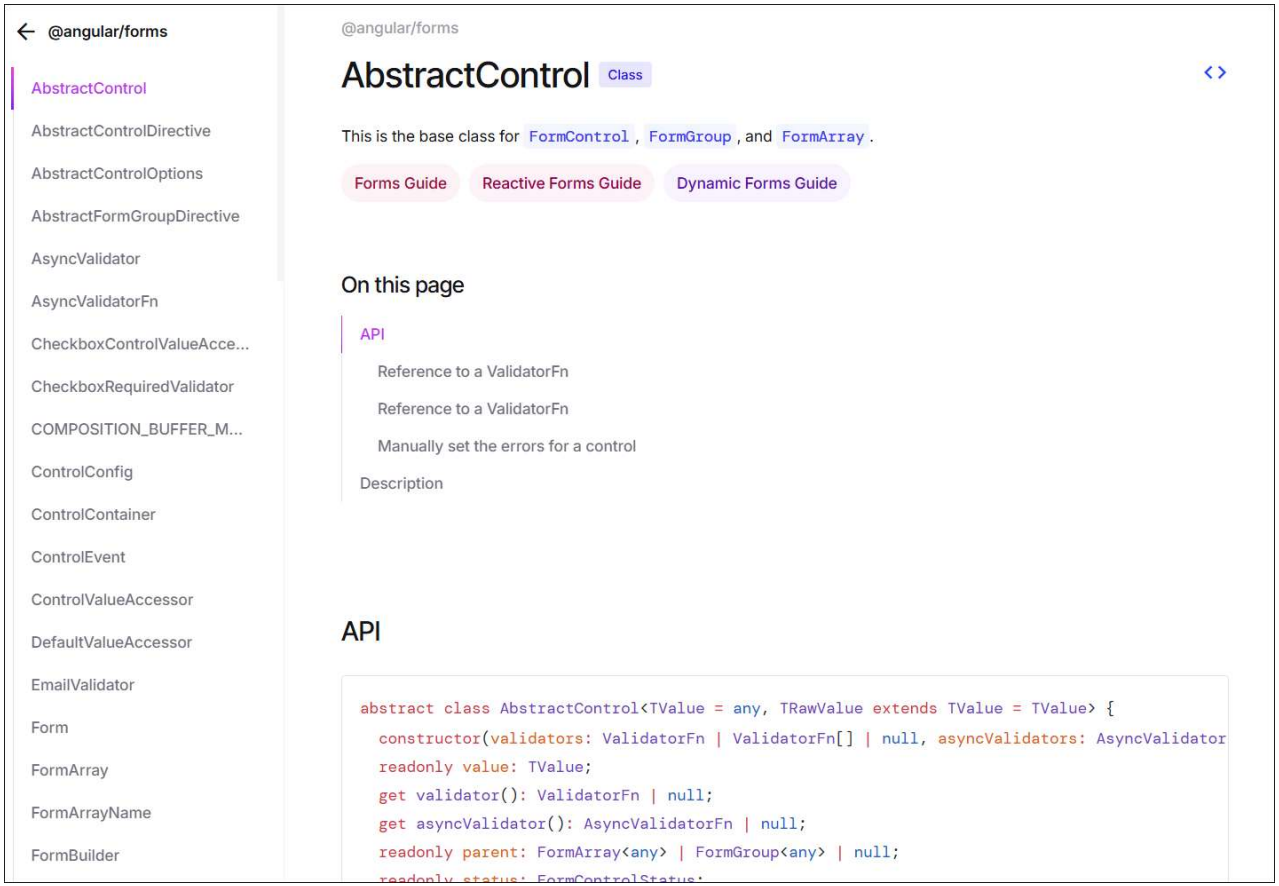


Form Controls in Detail

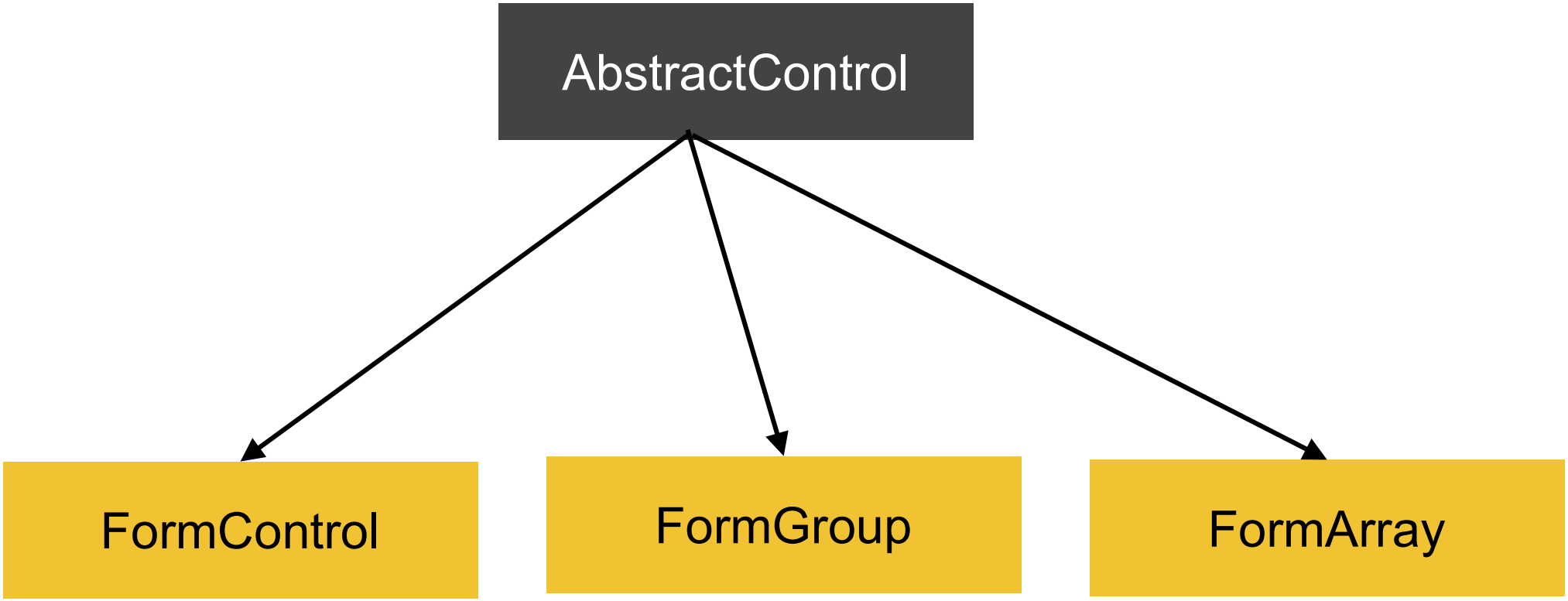


Angular Forms – Base class

- Every control extends from an `AbstractControl`
- https://angular.dev/api/forms/AbstractControl



So, every control is an extension



Summary – so far…

1

Template Driven Forms

Less to code

2

Model Driven Forms

More to code

3

Model

Value/Validity/State

Summary – Types of form

- The Top-5 main differences between Template Driven Forms and Model Driven/Reactive Forms

1. Form Creation
   - Template-driven:

     Forms are defined in the HTML template using Angular directives like `ngModel`, `required`, `ngForm`, etc.
   - Reactive:

     Forms are created in TypeScript using the `FormGroup`, `FormControl`, and `FormBuilder` APIs.

   So, NO [(ngModel)] in Reactive Forms

2. Control Flow (Imperative vs. Declerative)

- Template-driven:
  - Declarative.
  - You define validation and binding via HTML attributes; Angular handles the wiring behind the scenes.
- Reactive:
  - Imperative.
  - You explicitly define form structure, validation, and updates in the component class. You handle all logic in TypeScript.

3. Validation Handling

- Template-driven:
  - Uses HTML5 attributes (`required`, `minlength`, etc.) and Angular validators declared in the template.
- Reactive:
  - Uses explicit `Validators API`, providing more control, composability, and reusability.
  - When all `Validators` return `true`, a control is considered `valid`.

4. Scalability & Testability

- Template-driven:
  - Easier for simple forms, but harder to unit test and less maintainable for complex logic.
  - However, suitable for example for search forms (with one search form), sign-up and login forms, etc.
- Reactive:
  - Designed for complex forms. More testable, easier to debug and extend due to its programmatic nature.
  - Suitable for example for wizards, multi step forms, dynamic forms.

5. Two-way data binding

- Template-driven:
  - Uses [(ngModel)] which provides automatic two-way data binding between form controls and the model.
- Reactive:
  - Data flows explicitly; no `ngModel` needed or possible. You `.subscribe()` to `valuechanges` or call `setValue()` or `patchValue()` to update.

So, in a Reactive Form…

- No more `ngForm` → use `[formGroup]`
- No more `ngModel` → use `formControlName`
- Import the `ReactiveFormsModule`
- Form state lives in the Component, *not* in the Template
- Possible validations are in the Component, not in the Template

- The view is *not* generated for you.
- You need to write the HTML yourself

Reactive Forms

Mostly used for complex (longer) Forms and dynamic Forms

Form Controls + Validators

- In a Reactive Form, all controls are TypeScript variables. Let's build one.
- Reactive forms are based on *reactive programming* we already know
  - Events, Event Emitters
  - Observables
- Every form control is an observable!

```
export abstract class AbstractControl {

      ...
      private _valueChanges: EventEmitter<any>;
      …
      get valueChanges(): Observable<any> {
            return this._valueChanges;
      }
      ...
}
```

Form Controls are observables

- Import & instantiate in the Component
- Build your model in `constructor` or `ngOnInit`.
- Listen to changes `(.subscribe())` and act accordingly:

```
export class AppComponent1 implements OnInit {

   myReactiveForm: FormGroup;

   constructor(private formBuilder: FormBuilder) {
   }

   ngOnInit() {
      this.myReactiveForm = this.formBuilder.group({
         email    : ``,
         password: ``
      })
   }
}
```

Subscribe to those observables

```
// 1. complete form
this.myReactiveForm.valueChanges.subscribe((value)=>{
   console.log(value);
});

// 2. watch just one control
this.myReactiveForm.get('email').valueChanges.subscribe((value)=>{
   console.log(value);
});
```

Don’t forget to import `ReactiveFormsModule`

- In `app.module.ts`, or in your `bootstrapApplication()` if it is modal.

```
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {FormsModule, ReactiveFormsModule} from '@angular/forms';
import …
@NgModule({
   imports     : [
      BrowserModule,
      FormsModule,
      ReactiveFormsModule,
      …
   ],
   …
})
export class AppModule {
}
```

use [formGroup] and formControlName

```
<form novalidate [formGroup]="myReactiveForm">
   <div class="form-group">
      <label for="inputEmail">Email address</label>
      <input type="email" class="form-control" id="inputEmail"
            placeholder="Enter email" name="email"
            formControlName="email">
   </div>
   …
   // all other controls
</form>
```

Build the form in your component

```
export class AppComponent1 implements OnInit {
   myReactiveForm: FormGroup;
   constructor(private formBuilder: FormBuilder) {
   }
   ngOnInit() {
      // 1. Define the model of Reactive Form.
      // Notice the nested formBuilder.group() for group Customer
      this.myReactiveForm = this.formBuilder.group({
         email   : ``,
         password: ``,
         customer: this.formBuilder.group({
            prefix: ``,
            firstName: ``,
            lastName: ``
         })
      })
   }
}
```

Subscribe to changes

```
ngOnInit() {
    …

   // 2. Subscribe to changes at form level or...
   this.myReactiveForm.valueChanges.subscribe((value)=>{
      console.log('Changes at form level: ', value);
   });

   // 3. Subscribe to changes at control level.
   this.myReactiveForm.get('email').valueChanges.subscribe((value)=>{
      console.log('Changes at control level: ', value);
   });
}
```

Submitting a reactive form

- Can be based on `.valueChanges()` (though not very likely) for any given form control or complete form
- Use just `.click()` event handler for submit button

```
<button type="submit" class="btn btn-primary"
      (click)="onSubmit()"
      [disabled]="!myReactiveForm.valid">
   Submit
</button>
```

```
onSubmit() {
   console.log('Form submitted: ', this.myReactiveForm.value);
   // TODO: do something useful with form
}
```

Validating a Reactive Form

- Adding `Validators` to class definition
  - `email  : ['', Validators.required],`
- Multiple validations? Add an array of `Validators`, using `[validator1, validator2, validatorN,…]`
- Previously: using `Validators.compose(…)`, which is now outdated!

```
this.myReactiveForm = this.formBuilder.group({
   email   : ['', Validators.required],
   password: ['', [Validators.required, Validators.minLength(6)]],
   confirm: ['', [Validators.required, Validators.minLength(6)]],
   …
});
```

Adding Custom Validators

- Creating a Password-confirm validator
- Steps:
  1. Create a validation function, taking `AbstractControl` as a parameter
  2. Write your logic
  3. Don’t forget: pass the function in as a configuration parameter for the group or form you are validating!

```
function passwordMatcher(control: AbstractControl) {
   return control.get('password').value === control.get('confirm').value
      ? null : {'nomatch': true};
   // we *could*  return just true/false here, but by returning an object
   // we're more flexible in composing our validators.
}
```

```
this.myReactiveForm = this.formBuilder.group({
   email    : ['', Validators.required],
   password: ['', Validators.compose([Validators.required, Validators.minLength(6)])],
   confirm : ['', Validators.compose([Validators.required, Validators.minLength(6)])],
 },
 {validator: passwordMatcher} // pass in the validator function
);
```

Or, different notation for Modern Angular:

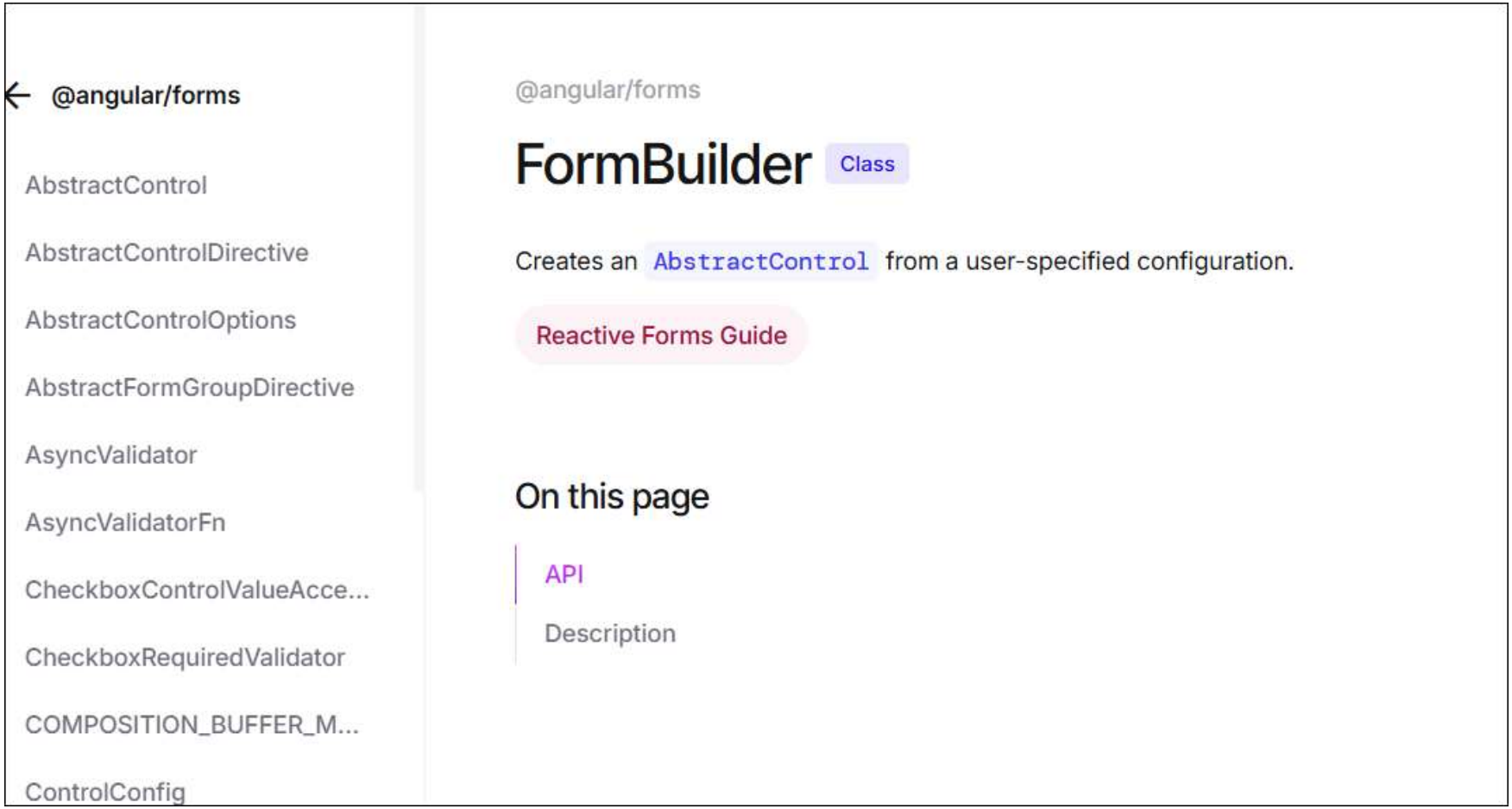
```
this.myReactiveForm = new FormGroup({
  email: new FormControl('', {
   nonNullable: true,
   validators: Validators.required }
   ),
  password: new FormControl('', {
    nonNullable: true,
    validators: [Validators.required, Validators.minLength(6)]
  }),
  confirm: new FormControl('', {
    nonNullable: true,
    validators: [Validators.required, Validators.minLength(6)]
  }),
  customer: new FormGroup({
    prefix: new FormControl('', { nonNullable: true }),
    firstName: new FormControl('', { nonNullable: true }),
    lastName: new FormControl('', { nonNullable: true })
  })
}, { validators: passwordMatcher });
```

Key differences:

- Replaced `this.formBuilder.group()` with `new FormGroup()`
- Replaced the array syntax `['']` with `new FormControl()`
- Added `nonNullable: true` option which is recommended in modern Angular for better type safety
- Nested group is also created using `new FormGroup()` instead of `this.formBuilder.group()`

More on FormBuilder class

- https://angular.dev/api/forms/FormBuilder Information on using and configuring FormBuilder

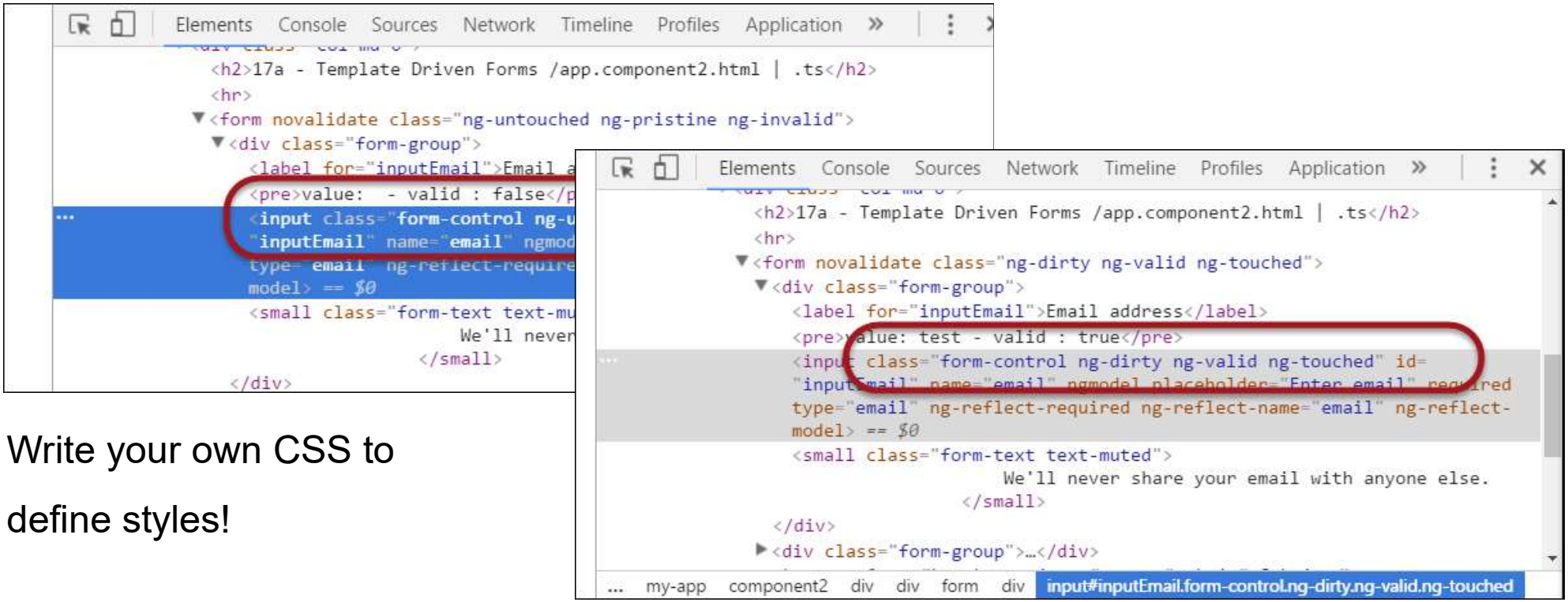


Form Control States

Create your own CSS to react to State changes – automatically controlled by Angular

Angular classes and checks

- Angular adds classes to the rendered HTML to indicate state
  - `ng-untouched / ng-touched,`
  - `ng-pristine / ng-dirty`
  - `ng-invalid / ng-valid`



Write your own CSS to define styles!

17a - Template Driven Forms /app.component2.html | .ts

Email address

```
value:  - valid : false
```

Enter email

We'll never share your email with anyone else.

Password

Validity

Invalid

Results

{ "email": "", "password": "" }

17a - Template Driven Forms /app.component2.html | .ts

Email address

```
value: test - valid : true
```

test

We'll never share your email with anyone else.

Validity

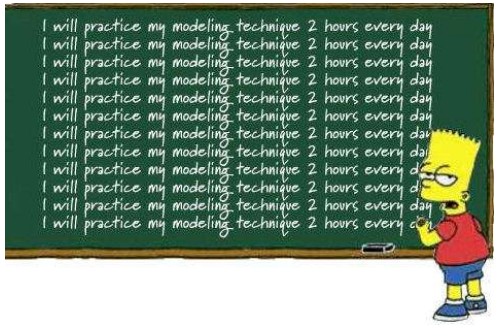
Valid

Results

Workshop

- Create your own Reactive Form for a sample application
  - Examples: a Signup-form, an order form, address form, etc.
- Add at least 3 reactive form fields
- Add validators for the form fields
- Write CSS that reacts to form field changes.

`../examples/150-reactive-forms`



Dynamic Form Arrays

Creating a form where you DO NOT KNOW the number of Form Controls in advance

Dynamic forms

- Sometimes, you don't know the number of Form Controls of a Form in advance/ at compile time
  - Use a `FormArray` for that
  - You can add and delete form controls at runtime to the array
  - Submit the complete array when you're done
- Example (on Angular 14, but concepts have not changed since):
  - `../examples/160-dynamic-formarray`

Example

Playlist Editor

This is a form with a Dynamic Form Array example, so we can indefinitely expand the form. See code for comments.

Playlist Component

Title * Liked Songs

Date * 17-05-2025

Description * My Description

Bruce Springsteen	Dancing in the dark	4:00
Madonna	Like a Virgin	03:30

+ Add Song

Save Playlist

TODO: Post to database, playlist:


```
{
  "title": "Liked Songs",
  "date": "2025-05-17",
  "description": "My Description",
  "songs": [
    {
      "artist": "Bruce Springsteen",
      "title": "Dancing in the dark",
```


Example 'Playlist

Dynamic FormArray

Code breakdown

- Creating a `playlist` variable, containing a `FormBuilder.group()`
  - Note: we could also have opted for `New FormGroup()`.
  - This is personal preference
- Note the `songs` variable. It contains an array of controls!

```
playlist = this.fb.group({
  title: ['', Validators.required],
  date: [new Date(), Validators.required],
  description: [''],
  songs: this.fb.array([])
})
```

Getting the songs in the form

- To get just the `songs` in the form, write a getter
  - This is not mandatory. It is just shorthand to retrieve all controls in the array

```
get songs(): FormArray {
  return this.playlist.controls['songs'] as FormArray
}
```

Adding and deleting songs

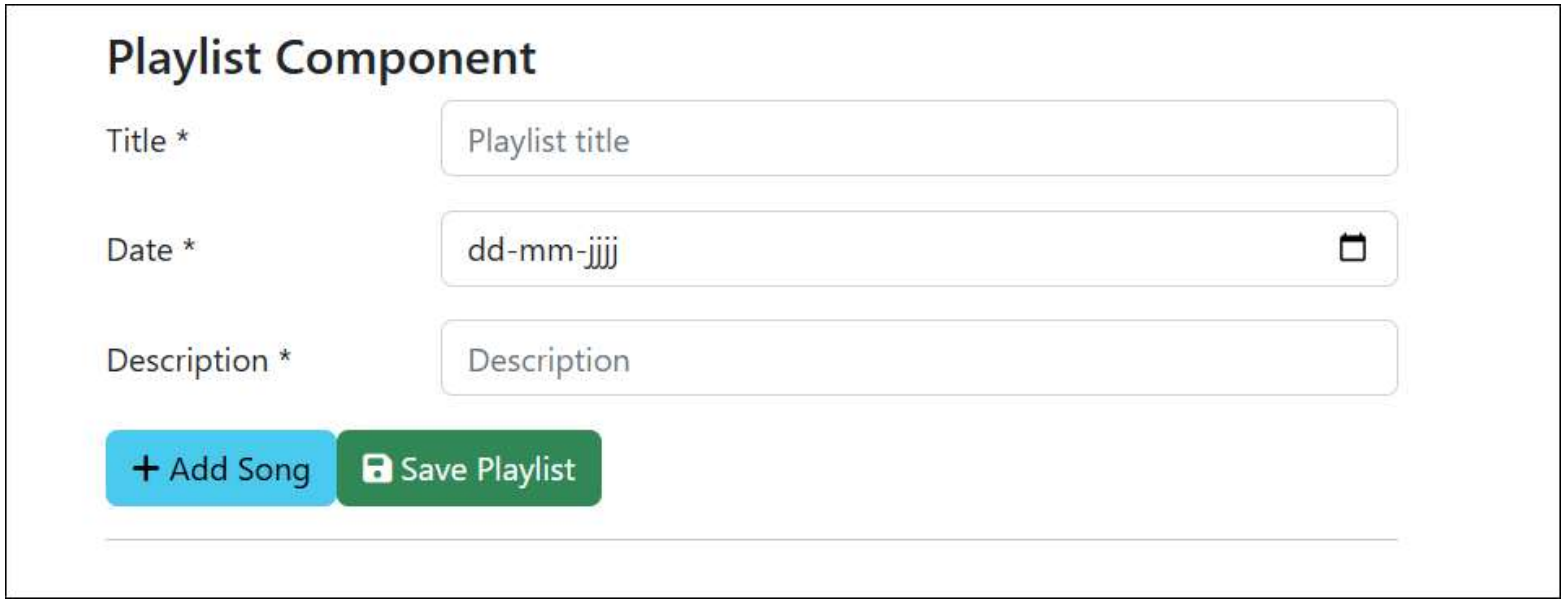
- Just write `(click)` eventhandlers to add/remove songs from the `FormArray`:

```
// Adding a new song by pushing a new song item to the formArray
addSong() {
  const newSong = this.fb.group({
    artist: ['', Validators.required],
    title: ['', Validators.required],
    duration: ['00:00', Validators.required]
  })
  this.songs.push(newSong);
}

// delete a song
deleteSong(index: number) {
  this.songs.removeAt(index);
}
```

Summary

- Of course, this example can be finetuned
  - Refine layout/CSS classes
  - Validate songs/duration
  - Add added songs to `<table>` instead of showing the in form fields.
  - …



Article on dynamic forms

Medium Search

Mastering Dynamic Form Generation in Angular with FormArray

DharmendraSingh Negi Follow 8 min read · Mar 22, 2024

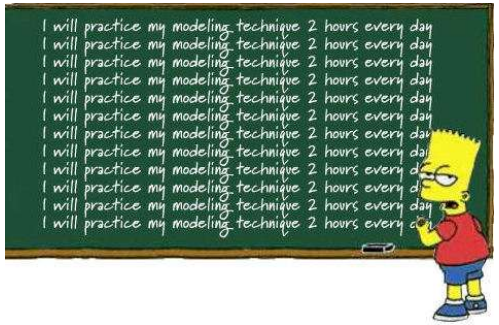
86 1

Angular dynamic forms allow developers to create flexible and interactive forms that can adapt to user input and changing data requirements. With dynamic forms, developers can easily add or remove form fields, change validation rules, and alter the form layout based on user actions or external data sources. This flexibility is particularly useful in scenarios where the

https://medium.com/@negidharmendra98/mastering-dynamic-form-generation-in-angular-with-formarray-60dc5e3997f3

Workshop

- Create your own Form with a dynamic FormArray for a sample application
- For example
  - Create a grocery list for shopping
  - Create a list of hobbies
  - Create a list of family members
- Optional: try to add multiple types of fields
  - Textfields, checkboxes, etc.

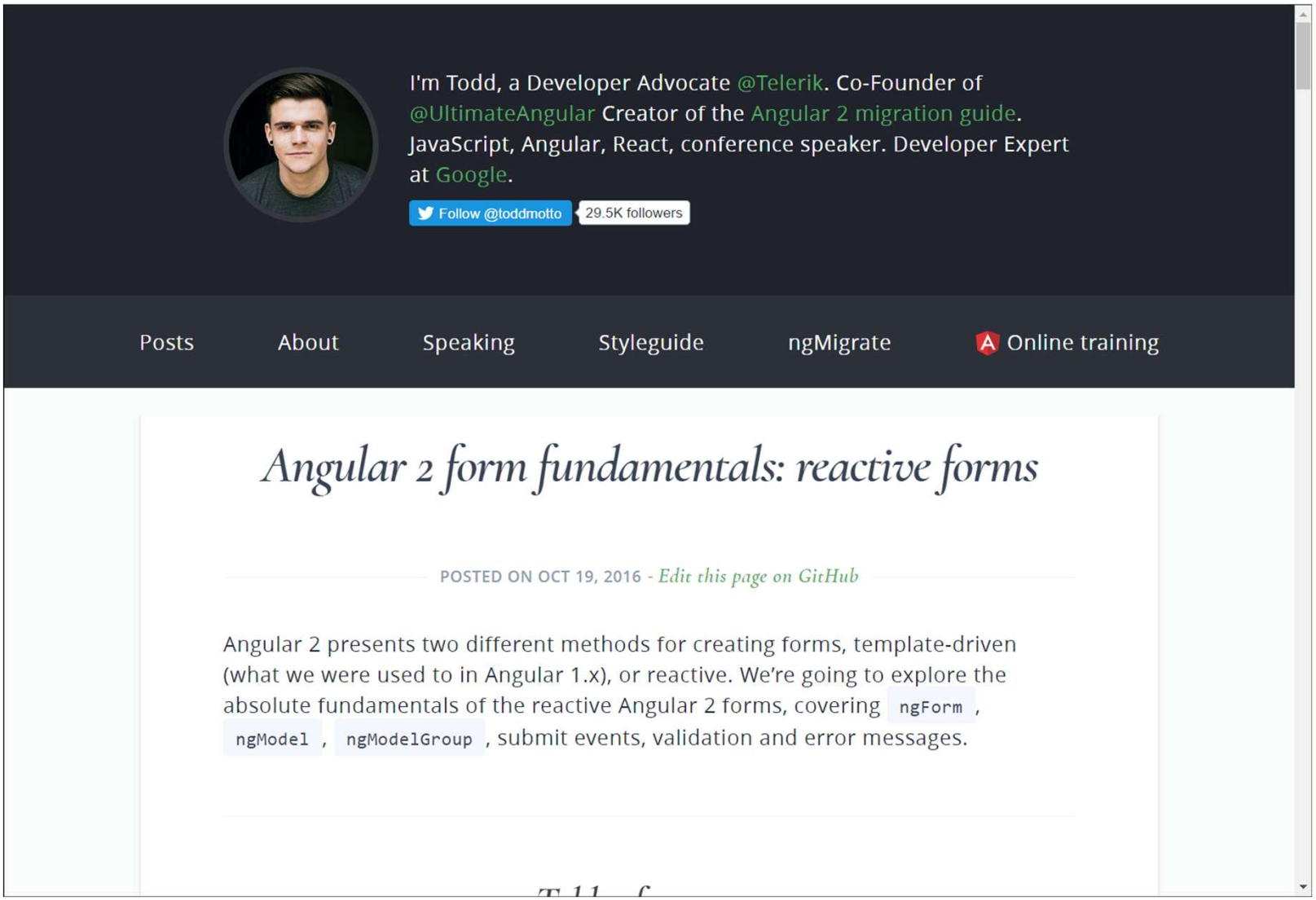



```
../examples/160-dynamic-formarray
```

More info

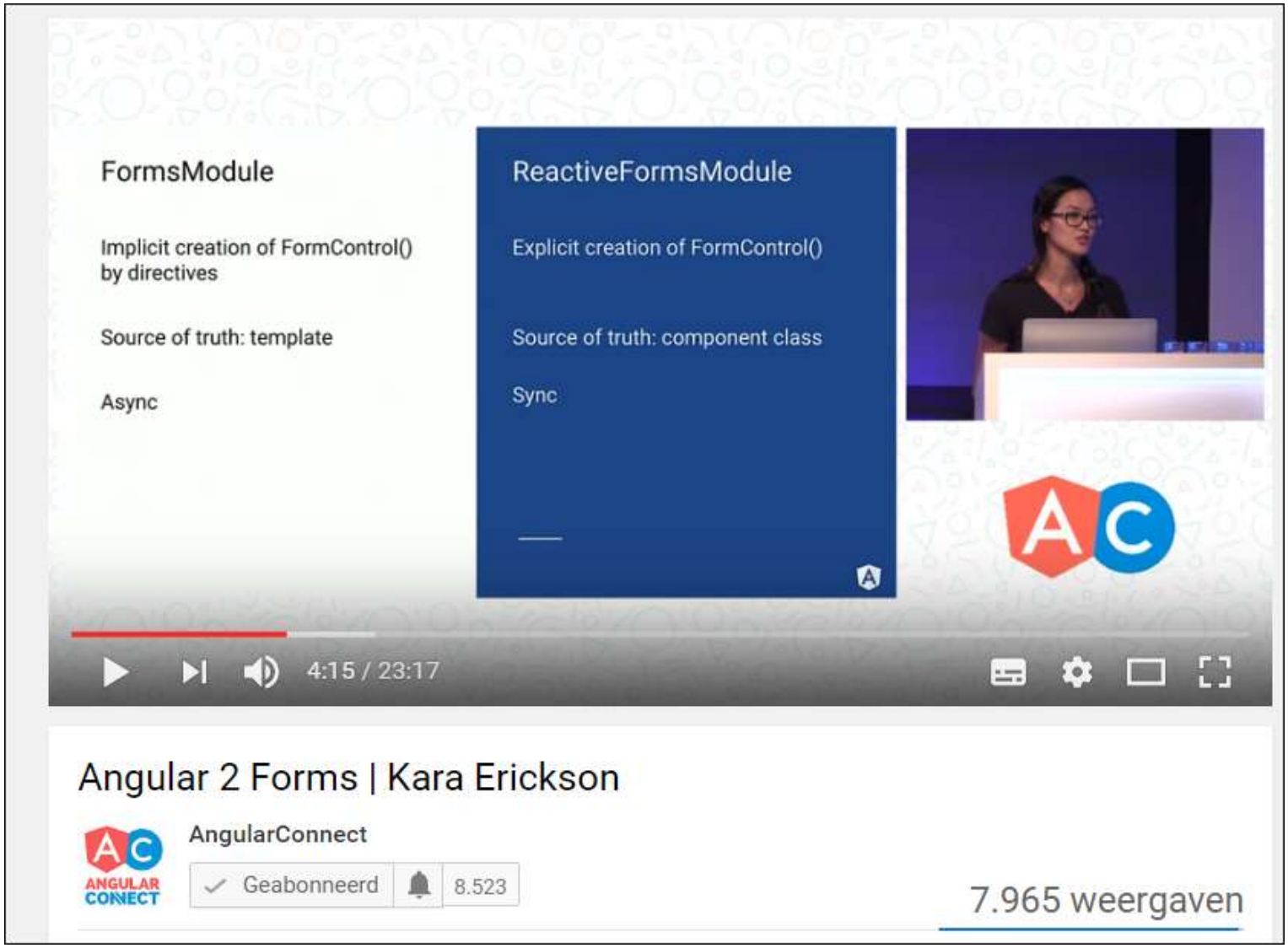
More info on Reactive Forms on the internetz…

More on Reactive Forms



https://toddmotto.com/angular-2-forms-reactive

Kara Erickson on Angular Forms



https://www.youtube.com/watch?v=xYv9lsrV0s4

Automated form and template generation, based on a form model:

If you like Formly, give it a star on GitHub and follow us on Twitter

Formly Guides UI Examples API v6 Search CTRL K

JSON powered / Dynamic forms in Angular

npm package 6.3.12 | downloads 370k/month | chat on gitter | CI failing | twitter @formlydev | Gitpod ready-to-code

Formly is a dynamic (JSON powered) form library for Angular that brings unmatched maintainability to your application's forms.

Features

- Automatic forms generation
- Easy to extend with custom field types, validation, wrappers and extensions.
- Supports multiple schemas:
  - Formly Schema (core)
  - JSON Schema

https://formly.dev/

github.com/PeterKassenaar/ngx-formly-demo

github.com/PeterKassenaar/ng2-form-edit