

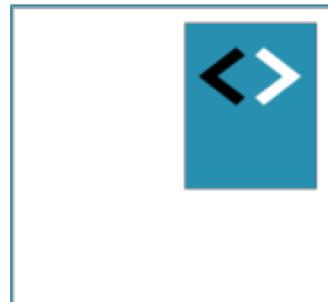


# Angular Fundamentals

## Module 7 – Forms



A CANON COMPANY



Peter Kassenaar  
[info@kassenaar.com](mailto:info@kassenaar.com)

# Contents

- Form Fundamentals
- (Template Driven Forms)
- Reactive Forms (aka *Model Driven Forms*)
- Subscribing to Form events

# Forms in Web Applications - Tasks

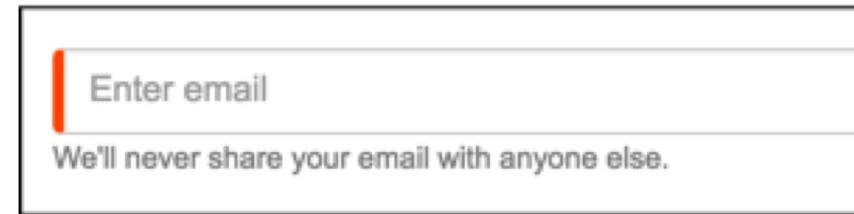
- Initialize Default Values



A screenshot of a web form showing an email input field. The field contains the placeholder text "peter@kassenaar.nl". Below the input field is a small text message: "We'll never share your email with anyone else.".

# Forms in Web Applications - Tasks

- Initialize Default Values
- Validate Data



The image shows a simple web form. It consists of a single input field with a placeholder "Enter email" and a horizontal line below it. Below the input field is a small text message: "We'll never share your email with anyone else." The entire form is enclosed in a thin black border.

Enter email

We'll never share your email with anyone else.

# Forms in Web Applications - Tasks

- Initialize Default Values
- Validate Data
- Display Validation messages

The image shows a web form with a single input field. The input field has a placeholder "Connect ID". Below the input field, there is a red error message: "Please enter ID in the pattern AA-XX where A is letter and X is a number". The entire form is enclosed in a black border.

Angular Connect ID
Connect ID
Please enter ID in the pattern AA-XX where A is letter and X is a number

# Forms in Web Applications - Tasks

- Initialize Default Values
- Validate Data
- Display Validation messages
- Serialize User Data



# Forms in Web Applications - Tasks

- Initialize Default Values
- Validate Data
- Display Validation messages
- Serialize User Data
- Dynamic Forms & Dynamic Controls

```
{  
  key: 'email',  
  type: 'input',  
  templateOptions: {  
    type: 'email',  
    label: 'Email address',  
    placeholder: 'Enter email'  
  },  
  {  
    key: 'password',  
    type: 'input',  
    templateOptions: {  
      type: 'password',  
      label: 'Password',  
      placeholder: 'Password'  
    },  
  },  
},
```



FULL NAME

EMAIL ADDRESS

SEX  
 Male  Female

DATE OF BIRTH

TIME OF ARRIVAL

# Forms in Web Applications - Tasks

- Initialize Default Values
- Validate Data
- Display Validation messages
- Serialize User Data
- Dynamic Forms & Dynamic Controls
- Custom Controls & Custom Validation

	Inv No	Date	Name	Amount	Price	Cost	Note
690	Inv No 690	7/15/2012	Name 690	444	671	297924	Note 690
691	Inv No 691	7/15/2012	Name 691	657	865	568305	Note 691
692	Inv No 692	7/15/2012	Name 692	804	92	73968	Note 692
693	Inv No 693	7/15/2012	Name 693	625	135	84375	Note 693
694	Inv No 694	7/15/2012	Name 694	906	608	550848	Note 694
695	Inv No 695	7/15/2012	Name 695	360	393	141480	Note 695
696	Inv No 696	7/15/2012	Name 696	293	600	175800	Note 696
697	Inv No 697	7/15/2012	Name 697	166	309	51294	Note 697

# **Angular 2 – Types of Forms**

**Template  
Driven forms**

**Model Driven  
forms**

# Angular 2 – Types of Forms

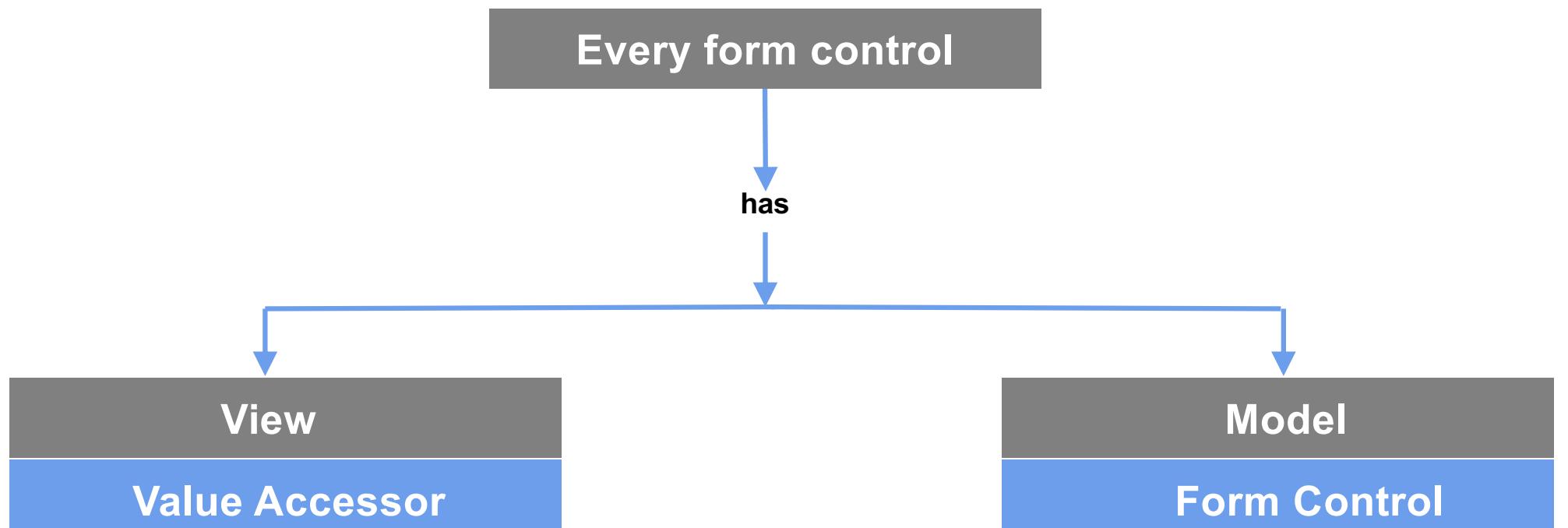
## Template Driven Forms

- Source of truth is the Template
- Define templates. Angular generates form model o/t fly
- Less descriptive
- Quickly Build simple forms – Less control
- Less testable

## Model Driven (Reactive Forms)

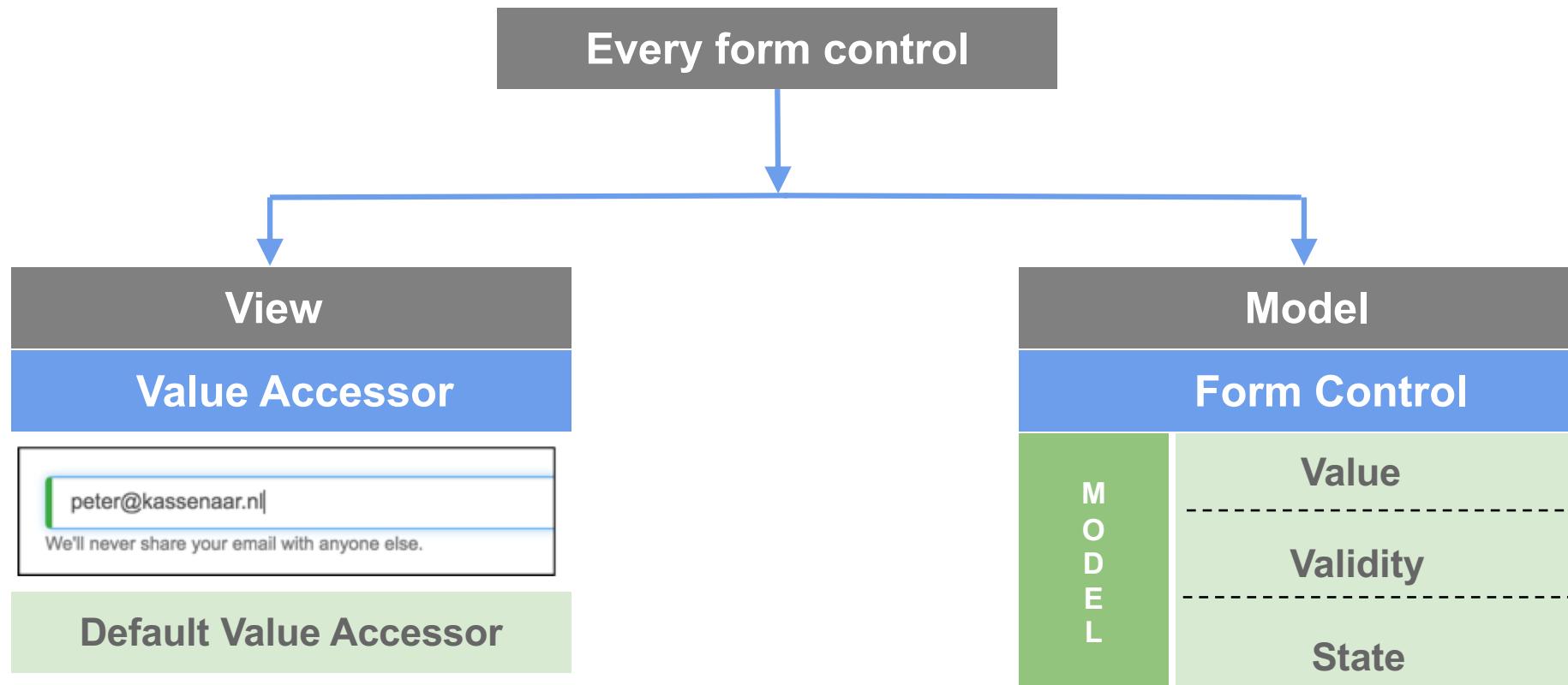
- Source of truth is the component class / directive
- Instantiate Form model and Control model yourself
- More Descriptive
- Code all the details. Takes more time, gives more control
- Very good testable

# Angular 2 Forms - Fundamentals



Retrieves value from HTML controls

Maintains model in component



```
export abstract class AbstractControl {  
    _value: any;  
  
    ...  
  
    private _status: string;  
    private _errors: {[key: string]: any};  
    private _pristine: boolean = true;  
    private _touched: boolean = false;  
  
    ...  
    get value(): any { return this._value; }  
    get valid(): boolean { return this._status === VALID; }  
  
    ...  
    abstract setValue(value: any, options?: Object): void;  
  
    ...  
}
```

<https://github.com/angular/angular/blob/master/modules/%40angular/forms/src/model.ts>

# Summary – what have we learned so far

1

**Template Driven Forms**

Less to code

2

**Model Driven Forms**

More to code

3

**Model**

**Value/Validity/State**

# **Angular 2 – Types of Forms**

**Template  
Driven forms**

**Model Driven  
forms**

# Let's build a template driven form!

- Step 1 – Add (or check) FormsModule in app/main.ts

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {FormsModule} from '@angular/forms'; 
import {AppModule} from './app.module';
```

## Step 2 – Add FormsModule to app.module.ts

```
import {NgModule}      from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {FormsModule}  from '@angular/forms';
```



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

```
@NgModule({
  imports      : [BrowserModule, FormsModule],
  declarations: [AppComponent],
  bootstrap    : [AppComponent]
})
export class AppModule {
```



# Step 3 – write form in HTML

```
<form novalidate>

  <div class="form-group">
    <label for="inputEmail">Email address</label>
    <input type="email" class="form-control" id="inputEmail"
           placeholder="Enter email" name="email">
    <small class="form-text text-muted">
      We'll never share your email with anyone else.
    </small>
  </div>

  <div class="form-group">
    <label for="inputPassword">Password</label>
    <input type="password" class="form-control" id="inputPassword"
           placeholder="Password" name="password">
  </div>

  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

This is just plain HTML. No Angular stuff here...

# Step 4. Defining a Template Driven Form

- Add `#myForm="ngForm"` to the `<form>` tag
  - This declares a local variable with the name `#myForm` to the `<form>` element. It is of type `NgForm`
- Add `ngModel` to each and every form field
  - No value necessary

```
<form #myForm="ngForm">
  <div class="form-group">
    <input type="email" class="form-control" id="inputEmail"
      placeholder="Enter email" name="email" ngModel>
  </div>
  <div class="form-group">
    <input type="password" class="form-control" ngModel
      id="inputPassword" placeholder="Password" name="password">
  </div>
```

# Just checking – Sample results pane

```
<div class="form-result">  
  <h3>Validity</h3>  
  <div class="validity" [ngClass]="'invalid-form': !myForm.valid">  
    <div *ngIf="myForm.valid">Valid</div>  
    <div *ngIf="!myForm.valid">Invalid</div>  
  </div>  
  <h3>Results</h3>  
  <div class="result">  
    {{ myForm.value | json }}  
  </div>  
</div>
```

Just to show runtime results of the Validity and Value of the form using

myForm.valid

myForm.value

# Results so far

## 17a - Angular 2 - Template Driven Forms

Email address

Enter email

We'll never share your email with anyone else.

Password

Password

Submit

Validity

Valid

Results

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

# Checkpoint

- The `#myForm` exposes the value and the validity of the form as a whole.
- `ngModel` adds the individual controls to the `#myForm`.
- You can now check it's value and state in the results pane
- Try what happens if you remove one of the `ngModel` directives!
- Check for yourself: the value of a form is a JSON-object.



# **Addressing individual controls**

# Retrieve values from individual controls

- Do the same as with the form
- Add for example `#email="ngModel"` to input field
- Now, the value, validity and state (i.e. its `ValueAccessors!`) are accessible through the local template variable

```
<label for="inputEmail">Email address</label>
<pre>value: {{ email.value }} - valid : {{ email.valid}}</pre>
<input type="email" class="form-control" id="inputEmail"
       placeholder="Enter email" name="email" ngModel #email="ngModel">
<small class="form-text text-muted">
    We'll never share your email with anyone else.
</small>
```

# Required fields

- Add HTML5 attribute required to the input field.
- No checking on type yet!
  - It's just required.

```
<input type="email" class="form-control" id="inputEmail"
       placeholder="Enter email" name="email" ngModel #email="ngModel" required>
```

## 17a - Template Driven Forms

/app.component2.html | .ts

Email address

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

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
```

We'll never share your email with anyone else.

Validity  
Valid

Results

# 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

The image shows two side-by-side browser developer tool windows, likely from Chrome DevTools, illustrating the addition of Angular classes to rendered HTML.

**Left Window:** Shows the state of an input field when it is empty and has not been focused. The input has the classes `form-control ng-untouched ng-pristine ng-invalid`. A red circle highlights the `ng-invalid` class.

```
<h2>17a - Template Driven Forms /app.component2.html | .ts</h2>
<hr>
<form novalidate class="ng-untouched ng-pristine ng-invalid">
  <div class="form-group">
    <label for="inputEmail">Email address</label>
    <input class="form-control ng-untouched ng-pristine ng-invalid" id="inputEmail" name="email" ngmodel="" type="email" ng-reflect-require-model="" ng-reflect-value="<pre>value: - valid : false</pre>">
    <small class="form-text text-muted">We'll never share your email with anyone else.</small>
  </div>
```

**Right Window:** Shows the same input field after it has been filled with the value "test". The input now has the classes `form-control ng-dirty ng-valid ng-touched`. A red circle highlights the `ng-touched` class.

```
<h2>17a - Template Driven Forms /app.component2.html | .ts</h2>
<hr>
<form novalidate class="ng-dirty ng-valid ng-touched">
  <div class="form-group">
    <label for="inputEmail">Email address</label>
    <input class="form-control ng-dirty ng-valid ng-touched" id="inputEmail" name="email" ngmodel="" placeholder="Enter email" required="" type="email" ng-reflect-required="" ng-reflect-name="email" ng-reflect-model="" ng-reflect-value="test">
    <small class="form-text text-muted">We'll never share your email with anyone else.</small>
  </div>
  <div class="form-group">...</div>
```

At the bottom of the right window, the status bar shows the selected element's class: `input#inputEmail.form-control.ng-dirty.ng-valid.ng-touched`.



# Using ngModelGroup

# Adding ngModelGroup

- Combining form fields into logical groups

```
<div ngModelGroup="customer" #customer="ngModelGroup">  
  <div class="form-group">  
    ...  
  </div>  
</div>
```

Use a local template variable (i.e. `#customer="ngModelGroup"`) only if you want to have access to the state and validity of the group as a whole.

# ngModelGroup creates a nested object

17a - ngModelGroup  
/app.component3.html | .ts

Email address

```
value: info@kassenaar.com - valid : true
```

```
info@kassenaar.com
```

We'll never share your email with anyone else.

Password

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

```
....
```

Prefix

```
Mr.
```

First Name

```
Peter
```

Last Name

```
Kassenaar
```

```
value {  
  "namePrefix": "Mr.",  
  "firstName": "Peter",  
  "lastName": "Kassenaar"  
}  
valid: true
```

Submit

Validity

Valid

Results

```
{ "email": "info@kassenaar.com", "password": "test", "customer":  
{ "namePrefix": "Mr.", "firstName": "Peter", "lastName":  
"Kassenaar" } }
```



# Submitting forms

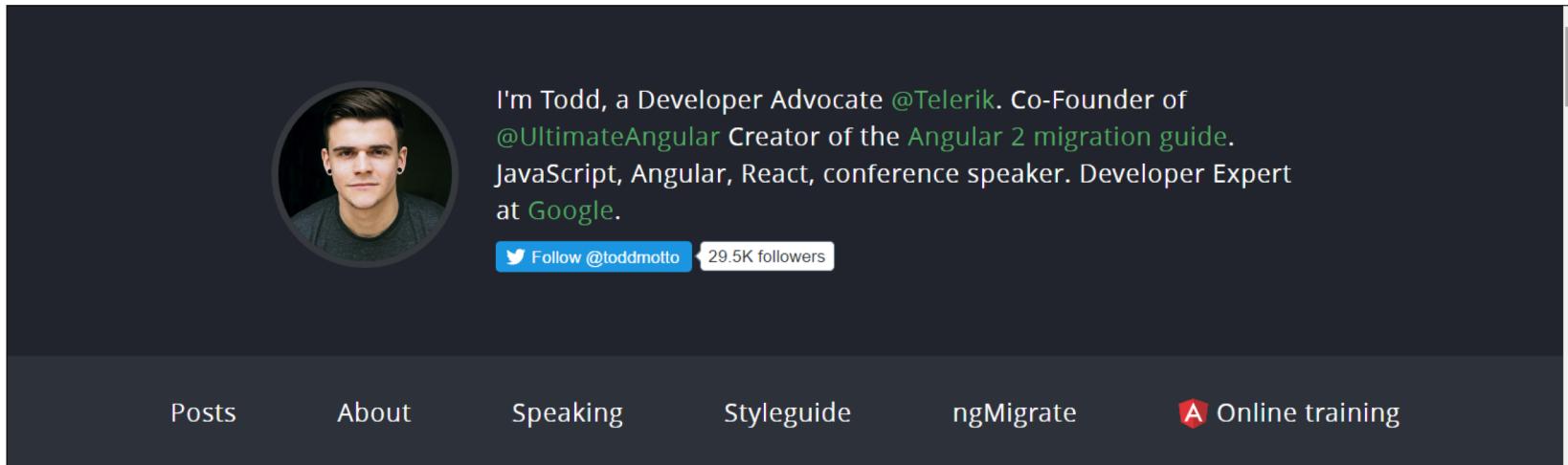
# Define a (click) handler on the button

- Only activate the button if the form is valid
- Pass `myForm` as a parameter
- Note: no actual need for two-way databinding with `[ (ngModel) ]`

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

```
onSubmit(form){  
  console.log('Form submitted: ', form.value);  
  alert('Form submitted!' + JSON.stringify(form.value))  
}
```

# More on Template Driven Forms



I'm Todd, a Developer Advocate [@Telerik](#). Co-Founder of [@UltimateAngular](#) Creator of the [Angular 2 migration guide](#). JavaScript, Angular, React, conference speaker. Developer Expert at [Google](#).

[Follow @toddmotto](#) 29.5K followers

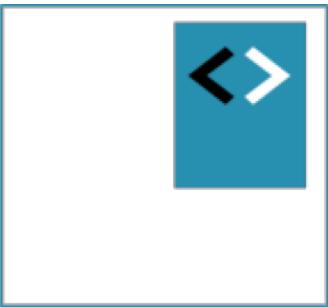
Posts    About    Speaking    Styleguide    ngMigrate     Online training

## *Angular 2 form fundamentals: template-driven forms*

POSTED ON OCT 18, 2016 - [Edit this page on GitHub](#)

Angular 2 presents two different methods for creating forms, template-driven (what we were used to in Angular 1.x), or reactive. We're going to explore the absolute fundamentals of the template-driven Angular 2 forms, covering `ngForm`, `ngModel`, `ngModelGroup`, submit events, validation and error messages.

<https://toddmotto.com/angular-2-forms-template-driven>



# Model Driven Forms

Or: *Reactive Forms*

# Reactive Forms

- 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;  
    }  
    ...  
}
```

# Differences - key things to remember

- No more `ngForm` → use `[formGroup]`
- No more `ngModel` → use `formControlName`
- Import `{ReactiveFormsModule}` from '`@angular/forms`'
- Form state lives in the Component, *not* in the View
- Possible validations are in the Component, not in the View
- The view is *not* generated for you.
- You need to write the HTML yourself

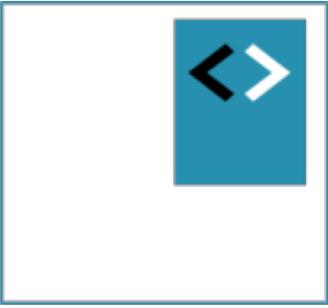
# Form Controls are observables

- 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);  
});
```



# Building reactive forms

# Step 1 – import ReactiveFormsModuleModule

- app.module.ts

```
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 {  
}
```

## Step 2 – 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>
```

# Step 3 – Build your form in 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
}
```



# Form Validation

# **1. Validating Template driven forms**

Use HTML5-attributes like `required`, `pattern`,  
`minlength` and so on.

Under the hood, these are actually Angular directives!  
Angular adds/removes corresponding classes.

```
<input type="password" class="form-control" ngModel  
id="inputPassword" placeholder="Password" name="password"  
  
#pw="ngModel" required minlength="6">
```

# **Validating reactive forms**

No more declarative attributes required, minlength,  
maxlength and so on.

Add Validator on the component class instead.

Configure validator per your needs.

# Angular built-in validators

[angular/modules/@angular/forms/src/validators.ts](https://github.com/angular/angular/blob/master/packages/forms/src/validators.ts)

```
export class Validators {  
  
    static required(control: AbstractControl): {[key: string]: boolean} {  
    }  
  
    static minLength(minLength: number): ValidatorFn {  
    }  
    static maxLength(maxLength: number): ValidatorFn {  
    }  
    static pattern(pattern: string): ValidatorFn {  
    }  
    static nullValidator(c: AbstractControl): {  
    }  
    . . .  
}
```

# Adding default Validators

- Adding Validators to class definition

- email : ['', Validators.required],

- Multiple validations? Add an array of Validators, using

Validators.compose()

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

# More on FormBuilder class

- <https://angular.io/docs/ts/latest/api/forms/index/FormBuilder-class.html>
- Information on using and configuring FormBuilder

The screenshot shows the Angular documentation for the `FormBuilder` class. The top navigation bar includes links for `Forms API`, `Angular`, `Components`, `Router`, `Material`, `Angular Flex Layout`, and `Angular Universal`. The `FormBuilder` page has a blue header with the title and a green "STABLE" badge. Below the header, it says "CLASS". The "What it does" section explains that it creates an `AbstractControl` from a user-specified configuration, described as syntactic sugar for building forms. The "How to use" section shows an example code snippet:

```
1. import {Component, Inject} from '@angular/core';
2. import {FormBuilder, FormGroup, Validators} from '@angular/forms';
3.
4. @Component({
5.   selector: 'example-app',
6.   template: `
```



# Subscribing to form events

Working with Observables (again). Typeahead demo

# Define a form

```
<form novalidate [formGroup]="searchForm">  
  <div class="form-group">  
    <label for="searchYouTube">Search YouTube</label>  
    <input type="text" class="form-control" id="searchYouTube"  
           formControlName="searchYouTube"  
           placeholder="Search YouTube" name="search">  
  </div>  
</form>
```

# Define component

- Compose a class, subscribe to `.valueChanges()` event

```
import {Http, Response} from '@angular/http';
import {Observable} from 'rxjs/Observable'
import {FormControl, FormGroup} from "@angular/forms";
...
// import just the operators we need, not import 'rxjs/Rx'
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/debounceTime';

// define some constants
const BASE_URL = 'https://www.googleapis.com/youtube/v3/search';
const API_KEY   = 'AIzaSyBdi3LXzf1xWXOAvgAwNkGvjnM1TwSV4VU';
// compose a url to search for, based on a query/keyword
const makeURL = (query: string) => `${BASE_URL}?q=${query}&part=snippet&key=${API_KEY}`;
```

```

@Component({
  selector : 'component1',
  templateUrl: 'app/component1/app.component1.html'
})
export class AppComponent1 implements OnInit {
  videos: Observable<any[]>

  // compose our form
  searchYouTube = new FormControl();
  searchForm    = new FormGroup({
    searchYouTube: this.searchYouTube,
  });

  constructor(private http: Http) {
  }

  ngOnInit() {
    // subscribe to Youtube input textbox and bind async (see html)
    this.videos = this.searchYouTube.valueChanges
      .debounceTime(600)           // wait for 600ms to hit the API
      .map(query => makeURL(query)) // turn keyword into a real youtube-URL
      .switchMap(url => this.http.get(url)) // wait for, and switch to the Observable that my http get ...
      .map((res: Response) => res.json()) // map its response to json
      .map(response => response.items); // unwrap the response and return only the items array
  }
}

```

# **Reactive forms examples**

- See 502 as an example
  - YouTube Search
  - Wikipedia Search

# More on Reactive Forms

I'm Todd, a Developer Advocate @Telerik. Co-Founder of @UltimateAngular Creator of the Angular 2 migration guide. JavaScript, Angular, React, conference speaker. Developer Expert at Google.

Follow @toddmotto 29.5K followers

Posts About Speaking Styleguide ngMigrate A Online training

## Angular 2 form fundamentals: reactive forms

POSTED ON OCT 19, 2016 - [Edit this page on GitHub](#)

Angular 2 presents two different methods for creating forms, template-driven (what we were used to in Angular 1.x), or reactive. We're going to explore the absolute fundamentals of the reactive Angular 2 forms, covering `ngForm`, `ngModel`, `ngModelGroup`, submit events, validation and error messages.

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

# Kara Erickson on Angular Forms

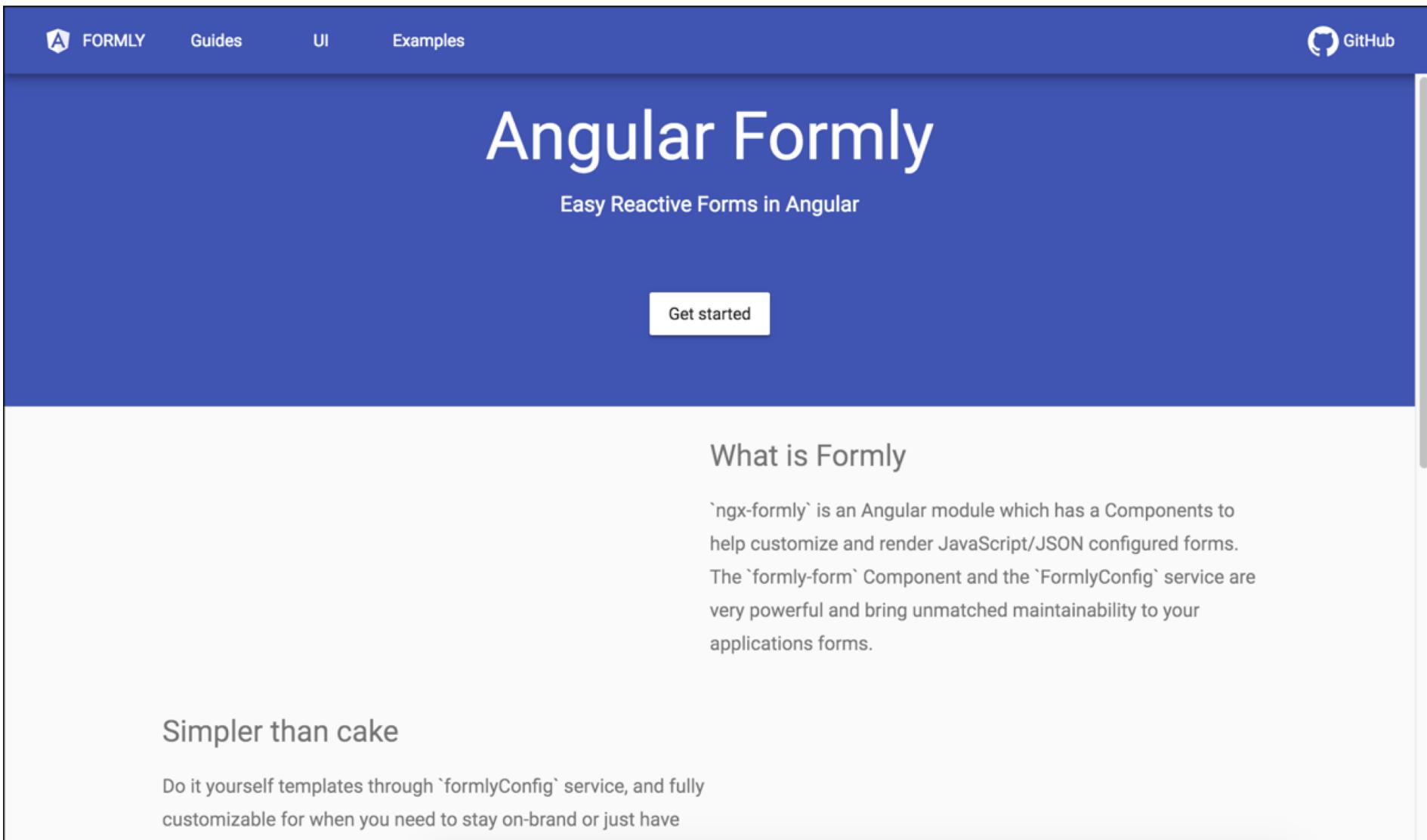
The screenshot shows a comparison between two Angular form modules:

FormsModule	ReactiveFormsModule
Implicit creation of FormControl() by directives	Explicit creation of FormControl()
Source of truth: template	Source of truth: component class
Async	Sync

The video player interface includes a play button, volume control, timestamp (4:15 / 23:17), and other standard video controls. Below the video, the title "Angular 2 Forms | Kara Erickson" is displayed, along with the Angular Connect logo, a "Geabonneerd" button, a notification bell icon, a view count of 8.523, and a view count of 7.965 weergaven.

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

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



The screenshot shows the official website for Angular Formly. The header is blue with the 'FORMLY' logo, 'Guides', 'UI', and 'Examples' navigation items, and a 'GitHub' link. The main title 'Angular Formly' is prominently displayed in large white font, with the subtitle 'Easy Reactive Forms in Angular' below it. A 'Get started' button is visible. The main content area has a white background and features sections like 'What is Formly' and 'Simpler than cake', each with descriptive text. The overall design is clean and professional.

FORMLY

Guides UI Examples

**Angular Formly**

Easy Reactive Forms in Angular

Get started

## What is Formly

`ngx-formly` is an Angular module which has a Components to help customize and render JavaScript/JSON configured forms. The `formly-form` Component and the `FormlyConfig` service are very powerful and bring unmatched maintainability to your applications forms.

## Simpler than cake

Do it yourself templates through `formlyConfig` service, and fully customizable for when you need to stay on-brand or just have

<https://formly-js.github.io/ngx-formly/>