Angular2

GitHub: <https://github.com/lightforsin/angular-seed>

Commands:

1. **npm install** 🡪 installs dependencies
2. **npm start** 🡪 starts the server

All dependencies and possible npm commands can be found in the *packages.json* file.

# The index.html file

<!-- 1. Load libraries -->

<!-- IE required polyfills, in this exact order -->

<script src="node\_modules/es6-shim/es6-shim.min.js"></script>

<script src="node\_modules/systemjs/dist/system-polyfills.js"></script>

<script src="node\_modules/angular2/bundles/angular2-polyfills.js"></script>

<script src="node\_modules/systemjs/dist/system.src.js"></script>

<script src="node\_modules/rxjs/bundles/Rx.js"></script>

<script src="node\_modules/angular2/bundles/angular2.dev.js"></script>

<!-- 2. Configure SystemJS -->

<script>

System.config({

packages: {

app: {

format: 'register',

defaultExtension: 'js'

}

}

});

System.import('app/boot')

.then(null, console.error.bind(console));

</script>

<!-- 3. Display the application -->

<body>

<my-app>Loading...</my-app>

</body>

# boot.ts

import {bootstrap} from 'angular2/platform/browser'

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

bootstrap(AppComponent);

# course.service.ts

export class CourseService {

getCourses(): string[] {

return ["Course 1", "Course 2", "Course 4"];

}

}

# auto-grow.directive.ts

import {Directive, ElementRef, Renderer} from 'angular2/core';

// **ElementRef (service)** is used for getting access to the **host** element.

// **Renderer (service)**, used for modifying elements

@Directive({ // directive decorator

selector: '[autoGrow]', // the [ ] means it will be applied as an HTML attribute (e.g. <input autoGrow />)

host: { // subscribe to events raised by this element

'(focus)': 'onFocus()', // when the **focus** event is fired, the onFocus() method is called

'(blur)': 'onBlur()' //\* events are placed between parentheses

}

})

export class AutoGrowDirective {

constructor(private el: ElementRef, private renderer: Renderer){

// when using “private” before the argument, the TS compiler automatically creates the **el** and **renderer** fields within the class

};

onFocus() {

this.renderer.setElementStyle(this.el.nativeElement, 'width', '200');

}

onBlur() {

this.renderer.setElementStyle(this.el.nativeElement, 'width', '120');

}

}

# courses.component.ts

import {Component} from 'angular2/core';

import {CourseService} from './course.service';

@Component({ // attribute, metadata, annotation 🡪 Component decorator

selector: 'courses', // the actual HTML tag 🡪 <courses>

template: `

<h2>Courses</h2>

{{ title }} // interpolation (one-way binding)

<ul>

<li \*ngFor="#course of courses">{{course}}</li> // iterating through a collection

</ul>`,

providers: [CourseService] // declaring dependencies

})

export class CoursesComponent { // exposing the component through “export”

title: string = "Course title";

courses: string[];

constructor(courseService: CourseService) { // dependency injection 🡪 the courseService is injected in the constructor

this.courses = courseService.getCourses();

};

}

# app.component.ts

import {Component} from 'angular2/core';

import {CoursesComponent} from './courses.component'; // importing component

@Component({

selector: 'my-app', // usage: <my-app>

template: '<h1>AJS 2 app</h1><courses></courses>',

directives: [CoursesComponent] // declaring directive dependencies

})

export class AppComponent { }

**Decorators**

They are prefixed with @.

They are functions and must be called as one, passing it a JSON object:

**Conventions**

A component should be named like: **<name>.component**

A service should be named like: **<name>.service**

A directive should be named like: **<name>.directive**

**Other aspects**

Each class is considered a module. That’s why we need to “export” classes to be used in other modules.

A component (@Component) has a template.

A directive (@Directive) does NOT have a template.

*Import {CoursesComponent} from ‘./courses.component’;* 🡪 “**./**” means “current folder”

The HTML template can be written on multiple lines if used between `` (so NOT ‘’)

*template: `*

*<h2>Courses</h2>*

*{{ title }}`*

**Binding**

The following *img* tags do the same thing:

template: `

<img src="{{ imageUrl }}" /> // recommended when displaying text

<img [src]="imageUrl" /> // this is the preferred way for DOM properties

<img bind-src="imageUrl" />`

**Class binding**

<button class="btn btn-primary" [class.active]="isActive">Submit</button>

// *isActive* is a bool property of the components’ class

**Style binding**

<button class="btn btn-primary" [style.backgroundColor]="isActive ? 'blue' : 'gray'">Submit</button>

**Event binding**

We use () for event binding and [] for property binding.

<button (click)="onClick()">Submit 1</button>

<button on-click="onClick()">Submit 2</button>

onClick($event) {

$event.stopPropagation();

console.log("Button clicked", $event);

}

**Two-way binding**

<input type="text" [value]="title" /> 🡪 one-way binding

<input type="text" [(ngModel)]="title" /> 🡪 two-way binding

<input type="text" bindon-ngModel="title" /> 🡪 two-way binding

**The component API**

Input properties

1st method:

import {Input} from 'angular2/core';

export class FavoriteComponent {

@Input() title: string; 🡪 the “title” property is exposed by declaring it as @Input()

}

This property can be assigned to in the HTML using property binding:

template: `<favorite [title]="the title"></favorite>`

We can also use an alias for the property:

@Input(‘alias-title’) title: string;

This way, we bind the property via the alias: <favorite [alias-title]="the title"></favorite>

2nd method:

@Component({

selector: '…',

template: ‘…’

inputs: ['isFavorite'] 🡪 simply specify the input properties inside the *inputs* array

})

export class FavoriteComponent {

*isFavorite*: boolean = false;

}

This method also allows aliases:

inputs: ['isFavorite: is-favorite'] 🡪 the alias is “is-favorite”

The downside with method 2 is the ‘magic strings’: inputs are strings, not actual entities

Output properties

The syntax is very similar to the Input properties, just use “Output” instead of “Input”. The aliases also work in the same manner.

import {Output} from 'angular2/core';

export class FavoriteComponent {

@Output() title: string; 🡪 the “title” property is exposed by declaring it as @Output()

}

Or the second method:

@Component({

selector: '…',

template: ‘…’

outputs: ['isFavorite'] 🡪 simply specify the input properties inside the *inputs* array

})

Templates

The template can be supplied via an external html template. Inside the component this needs to be specified in the “templateUrl” property and the path needs to be relative to the **root** of the website.

Downside: an extra HTTP request is needed for the HTML.

Styles

@Component({

selector: 'favorite',

styles: [`

.glyphicon-star {

color: orange;

}

`]

})

**!!!** The .glyphicon-star is applied only to the component in which it is declared. Angular does this by adding a generated random HTML attribute on the element:

<i \_ngcontent-wnm-2="" class="glyphicon glyphicon-star"></i>

Built-in directives

**ngIf** 🡪 <div \*ngIf="courses.length > 0">…</div>

* if condition is false, it removes the element from the DOM

An alternative to not show an element is to hide it but it will still exist in the DOM:

<div [hidden]="courses.length == 0">…</div>

**ngSwitch** 🡪

<ul class="nav nav-pills">

<li><a (click)="viewMode='map'">Map view</a></li>

<li><a (click)="viewMode='list'">List view</a></li>

</ul>

<div [ngSwitch]="viewMode">

<template [ngSwitchWhen]="'map'" ngSwitchDefault>Map view content</template>

<template [ngSwitchWhen]="'list'">List view content</template>

</div>

**ngFor**

<div \*ngFor="#**course** of courses">

{{ **course** }}

</div>

*#course* – local variable (specified by the # symbol)

<div \*ngFor="#course of courses, #i = index"> 🡪 accessing the index (0-based)

{{ i + 1 }} + {{ course }}

</div>

* other properties: *first, last, even, odd*

The \*ngFor is just syntactic sugar for [ngFor] so we don’t need to define the <templates> ourselves, like it was the case for ngSwitch.

|  |  |
| --- | --- |
| \*ngFor | [ngFor] |
| <div \*ngFor="#course of courses, #i = index">  {{ i + 1 }} - {{ course }}  </div> | <template ngFor [ngForOf]="courses" #course #i = index>  <div>{{ i + 1 }} - {{ course }}</div>  </template> |

Pipes

*Uppercase, lowercase, decimal, currency, date, json*

Examples:

{{ course.title | uppercase}}<br/>

Rating: {{ course.rating | number:'2.2-2'}} <br />

Students: {{ course.students | number }} <br />

Price: {{ course.price | currency:'EUR' }} <br />

Date: {{ course.startDate | date }} <br />

Custom pipes

import {Pipe, PipeTransform} from 'angular2/core';

@Pipe({name: 'summary'})

export class SummaryPipe implements PipeTransform {

transform(value: string, args: string[]) {

var limit = (args && args[0]) ? parseInt(args[0]) : 50;

if (value) {

return value.substring(0, limit) + "...";

}

}

}

import {SummaryPipe} from '../../pipes/summary.pipe';

@Component({

selector: 'my-app',

pipes: [SummaryPipe]

…

Description: {{ course.description | summary:20 }}

**ngClass** and **ngStyle**

<i [ngClass]="{

'glyphicon glyphicon-heart': true,

'highlighted': iLike // local variable

}"></i>

<button

[ngStyle]=”{

backgroundColor: canSave ? ‘blue’ : ‘gray’,

color: canSave ? ‘white’ : ‘black’

}”>

</button>

**ng-content** (the ng-transclude from AJS 1.x)

|  |  |
| --- | --- |
| @Component({  selector: 'bs-panel',  template: `  <div class="panel panel-default">  <div class="panel-heading">  **<ng-content select=".heading"></ng-content>**  </div>  <div class="panel-body">  **<ng-content select=".body"></ng-content>**  </div>  </div>  `  }) | <bs-panel>  <div class="**heading**">My heading</div>  <div class="**body**">My body</div>  </bs-panel> |

Elvis operator

{{ task.assignee**?.**name }}

* task.assignee == null 🡪 it renders an empty string; otherwise, it renders the
* task.assignee != null 🡪 it renders the ‘name’ property

Form CONTROLS

*ControlGroup* – several controls grouped together which can be validated as a single unit.

*Control* and *ControlGroup* both have the same properties inherited from AbstractControl:

* value, touched, dirty, pristine, valid, invalid, …

To create control objects we need to associate them with the input fields from the form.

<input **ngControl**="firstName" id="firstName" type="text" class="form-control">

* **ngControl** tells Angular to associate the input control with the control object named “firstName”
* The “firstName” from ngController is in no way related to the id with the same name

<input ngControl="firstName" #**firstName**="ngForm" (change)="log(**firstName**)" type="text" class="form-control">

* **#firstName** is a temporary local variable; when its value is set to “ngForm”, Angular creates a reference to the ngControl applied on the input field.

FORM VALIDATION

There are only 3 validation rules: required, minlength, maxlength

<input

ngControl="firstName"

#firstName="ngForm"

id="firstName"

type="text"

required

minlength="3">

<div \*ngIf="firstName.touched && firstName.errors">

<div class="alert alert-danger" \*ngIf="firstName.errors.required">

First name is required.

</div>

<div class="alert alert-danger" \*ngIf="firstName.errors.minlength">

First name should be mininum {{ firstName.errors.minlength.requiredLength }} characters.

</div>

</div>

Complex validation

*Using control groups*

|  |  |
| --- | --- |
| <form [ngFormModel]="**form**">  <div class="form-group">  <label for="username">Username</label>  <input ngControl="**username**">  <div \*ngIf="!**form**.**controls**[**'username'**].valid"  class="alert alert-danger">Username is required</div> | export class SignUpFormComponent {  **form** = new ControlGroup({  **username**: new Control('', Validators.required),  **password**: new Control('', Validators.required)  });  } |

We can simplify the syntax by using *temporary variables* (# notation):

<form [ngFormModel]="form" (ngSubmit)=”signup()”>

// the ”signup” method has access to the “form” object in the component

<div class="form-group">

<label for="username">Username</label>

<input ngControl="username" #**username**=”ngForm”> // this points to the control

<div \*ngIf="!**username**.valid" class="alert alert-danger">Username is required</div>

A form built like this:

form = new ControlGroup({

username: new Control('', Validators.required),

password: new Control('', Validators.required)

});

it’s called “**model-driven form**”.

A cleaner way to create a form is by using the **FormBuilder** class:

export class SignUpFormComponent {

form: ControlGroup;

constructor(fb: **FormBuilder**) {

this.form = fb.**group**({

username: ['', Validators.required],

password: ['', Validators.required]

})

}

}

Custom validators

A custom validator is mostly a method which takes a **Control** object as parameter.

import {Control} from 'angular2/common';

export class UsernameValidators {

static cannotContainSpace(control: **Control**) {

if(**control.value**.indexOf(' ') >= 0)

return { cannotContainSpace: true };

return null;

}

}

If the validation passes, Angular returns **null**.

If the validation fails, Angular returns a JSON object which has as key the validationRule and as value it can have anything, but usually it has several information about why the validation failed (see examples below).

Example:

If the *required* validator fails, it will return

{required: true}

If the *minLength* validator fails, it will return

{

minLength: {

requiredLength: 3, // additional info

actualLength: 1 // additional info

}

}

Using the custom validator

import {**UsernameValidators**} from './usernameValidators'; // defined above

…

this.form = fb.group({

username: ['', **Validators.compose**([ // we use ‘**compose**’ for more than one validation per control

Validators.required,

**UsernameValidators.cannotContainSpace**

])],

password: ['', Validators.required]

})

Async validators

export class UsernameValidators {

static shouldBeUnique(control: Control) {

**return new Promise**((resolve, reject) => {

setTimeout(function() {

if(control.value === "abc")

{

resolve({shouldBeUnique: true});

}

else{

resolve(null);

}

}, 1000);

});

}

}

this.form = fb.group({

// the third element of the array contains the **async** validators

username: ['', Validators.required, **UsernameValidators.shouldBeUnique**],

password: ['', Validators.required]

})

Reactive extensions

Reactive Extensions is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.

ReactiveX is not related to Angular but is being used in several parts of it. ([www.reactivex.io](http://www.reactivex.io))

In index.html we have a reference to that library:

<script src="node\_modules/rxjs/bundles/Rx.js"></script>

Async data streams: DOM events, Web Sockets, Web Workers, AJAX calls

Observables

/// <reference path="../../../typings/tsd.d.ts" />

import {Component} from 'angular2/core';

import {**Observable**} from **'rxjs/Rx'**;

@Component({

selector: 'my-app',

template: `

<input id="search" type="text" class="form-control">

`

})

export class AppComponent {

constructor() {

var keyups = Observable.fromEvent($("#search"), "keyup")

.map(e => e.target.value) // map the data

.filter(text => text.length >=3) // filter the data

.debounceTime(400) // delay when inserting data

.distinctUntilChanged(); // eliminates duplicates entries

keyups.**subscribe**(data => console.log(data)); // subscribing to the observables

}

}