# Работа с компонентами

# Создание проекта с использованием webpack

Разработаем стандартное приложение angular с использованием webpack. Для этого создадим папку с проектом и проинициализируем его, используя команду

npm init

Добавим зависимости проекта в файл package.json:

{

  "name": "angularapp",

  "version": "1.0.0",

  "description": "first angularapp",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "dev": "webpack-dev-server --hot --open",

    "build": "webpack"

  },

  "dependencies": {

    "@angular/common": "~8.0.0",

    "@angular/compiler": "~8.0.0",

    "@angular/core": "~8.0.0",

    "@angular/forms": "~8.0.0",

    "@angular/platform-browser": "~8.0.0",

    "@angular/platform-browser-dynamic": "~8.0.0",

    "@angular/router": "~8.0.0",

    "core-js": "^3.1.3",

    "rxjs": "^6.5.2",

    "zone.js": "^0.9.1"

  },

  "devDependencies": {

    "@types/node": "^12.0.0",

    "typescript": "^3.5.0",

    "webpack": "^4.32.0",

    "webpack-cli": "^3.3.2",

    "webpack-dev-server": "^3.4.1",

    "angular2-template-loader": "^0.6.2",

    "awesome-typescript-loader": "^5.2.1"

  },

  "author": "Samovnick",

  "license": "ISC"

}

Также добавим в папку проекта новый файл tsconfig.json, определяющий конфигурацию компилятора typescript:

{

    "compilerOptions": {

        "target": "es5",

        "module": "es2015",

        "moduleResolution": "node",

        "sourceMap": true,

        "emitDecoratorMetadata": true,

        "experimentalDecorators": true,

        "lib": [

            "es2015",

            "dom"

        ],

        "noImplicitAny": true,

        "suppressImplicitAnyIndexErrors": true,

        "typeRoots": [

            "node\_modules/@types/"

        ]

    },

    "exclude": [

        "node\_modules"

    ]

}

Для сборки проекта будем использовать сборщик webpack, поэтому также определим в папке проекта файл webpack.config.js:

const path = require('path');

const webpack = require('webpack');

module.exports = {

    entry: {

        'polyfills': './src/polyfills.ts',

        'app': './src/main.ts'

    },

    output: {

        path: path.resolve(\_\_dirname, './public'),     // путь к каталогу выходных файлов - папка public

        publicPath: '/public/',

        filename: "[name].js"       // название создаваемого файла

    },

    resolve: {

        extensions: ['.ts', '.js']

    },

    module: {

        rules: [   //загрузчик для ts

            {

                test: /\.ts$/, // определяем тип файлов

                use: [

                    {

                        loader: 'awesome-typescript-loader',

                        options: { configFileName: path.resolve(\_\_dirname, 'tsconfig.json') }

                    },

                    'angular2-template-loader'

                ]

            },

            {

                test: /\.html$/,

                loader: 'html-loader'

            },

            {

                test: /\.css$/,

                include: path.resolve(\_\_dirname, 'src/app'),

                loader: 'raw-loader'

            }

        ]

    },

    plugins: [

        new webpack.ContextReplacementPlugin(

            /angular(\\|\/)core/,

            path.resolve(\_\_dirname, 'src'), // каталог с исходными файлами

            {} // карта маршрутов

        )

    ]

}

Проинсталлируем проект, запустив команду:

npm install

Приступим к настройке проекта. Для этого создадим каталог src, а в этом каталоге определим папку app.

В каталог src/app добавим новый файл app.component.ts, определяющий главный компонент приложения:

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

@Component({

    selector: 'angular-app',

    templateUrl: './src/app/app.component.html',

    styleUrls: ['./src/app/app.component.css'],

})

export class AppComponent {

    title = 'Angular-app';

}

Создадим файлы app.component.html

<h1>{{ title }} is running!!!</h1>

и app.component.css

h1 {

    font-size: 3em;

    color: navy;

    text-align: center;

}

, отвечающие за html-разметку и стили компонента.

Чтобы задействовать этот компонент, добавим в каталог src/app файл модуля 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 { }

Уровнем выше в каталоге src определим файл main.ts для запуска проекта:

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

import { AppModule } from './app/app.module';

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);

Также в каталоге src определим файл polyfills.ts, который необходим для запуска приложения:

import 'core-js';

// zone используется angular

import 'zone.js/dist/zone';

В конце определим главную страницу index.html в корневой папке проекта:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Angular 8</title>

</head>

<body>

    <angular-app></angular-app>

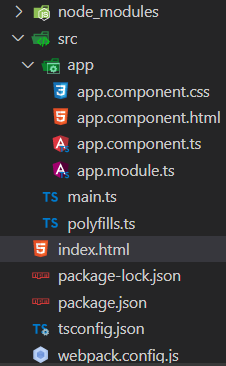
    <script src="public/polyfills.js"></script>

    <script src="public/app.js"></script>

</body>

</html>

В итоге у нас получится следующая структура проекта:



Для построения проекта используем команду

npm run build.

Для запуска проекта используем команду

npm run dev.

# Добавление компонентов

Кроме основных компонентов в приложении мы также можем определять какие-то вспомогательные компоненты, которые управляют каким-то участком разметки html. Более того в приложении на странице может быть ряд разных блоков с какой-то определенной задачей. И для каждого такого блока можно создать отдельный компонент, чтобы упростить управление блоками на странице.

Добавим в проект второй компонент. Для этого добавим в папку src/app новый файл app.childcomponent.ts:

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

@Component({

    selector: 'child-comp',

    template: `

        <h2>Hello {{ name }}!</h2>

        <p>Since you are {{ age }} years old you can do everything</p>

        `,

    styles: [`h2, p {color:red;}`]

})

export class ChildComponent {

    name = "Nick";

    age = 25

}

Теперь изменим html-разметку компонента AppComponent в файле app.component.html:

<h1>{{ title }} is running!!!</h1>

<child-comp></child-comp>

Чтобы использовать все определенные в проекте компоненты, они должны быть указаны в главном модуле приложения. Определим в файле app.module.ts следующий модуль:

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

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

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

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

import { ChildComponent } from './app.child.component';

@NgModule({

    imports: [BrowserModule, FormsModule],

    declarations: [AppComponent, ChildComponent],

    bootstrap: [AppComponent]

})

export class AppModule { }

# Взаимодействие между компонентами

По умолчанию компоненты никак не взаимодействуют, они независимы. Каждый компонент определяет свои выражения привязки. Однако, если мы хотим свойства дочернего компонента привязать к свойствам из главного компонента или наоборот, необходимо использовать декоратор @Input(). Чтобы использовать декоратор, его надо импортировать:

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

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

@Component({

    selector: 'child-comp',

    template: `

        <h2>Hello {{ name }}!</h2>

        <p>Since you are {{ age }} years old you can do everything</p>

        <p>Title - {{ title }} - was received from the main component<p>

        `,

    styles: [`h2, p {color:red;}`]

})

export class ChildComponent {

    @Input() title: string;

    name = "Nick";

    age = 25

}

Данные передаются как атрибуты элемента и фактически осуществляют привязку свойств:

<child-comp [title]="title"></child-comp>

Input используется для получения данных, в то время как Output для их отправки. Output отправляет данные выставляя их в качестве производителей событий, обычно как объекты класса EventEmitter.

Изменим содержимое дочернего компонента и его html-разметки.

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

@Component({

    selector: 'child-comp',

    templateUrl: './app.child.component.html',

    styles: [`h2, p {color:red;}`]

})

export class ChildComponent {

    @Input() title: string;

    @Output() onChanged = new EventEmitter<boolean>();

    change(increased: any) {

        this.onChanged.emit(increased);

    }

    name = "Nick";

    age = 25;

}

<h2>Hello {{ name }}!</h2>

<p>Since you are {{ age }} years old you can do everything</p>

<p>Title - {{ title }} - was received from the main component</p>

<p>Change name: <input type="text" [(ngModel)]="name"></p>

<button (click)="change(true)">+</button>

<button (click)="change(false)">-</button>

В этом компоненте у кнопки используется событие click, которое вызывает метод change, передавая ему значение true или false. Здесь же в дочернем компоненте мы можем и обработать события. Но если мы должны передавать его родительскому компоненту, то для этого нам надо использовать свойство типа EventEmitter, коим здесь является свойство onChanged. Поскольку мы будем передавать значение типа true или false, то данное свойство типизируется типом boolean. При этом свойство onChanged должно быть выходным, поэтому оно помечается с помощью декоратора @Output.

Далее переопределим код главного компонента и его разметки:

export class AppComponent {

    title = 'Angular-app';

    clicks: number = 0;

    getData(increased: any) {

        increased == true ? this.clicks++ : this.clicks--;

    }

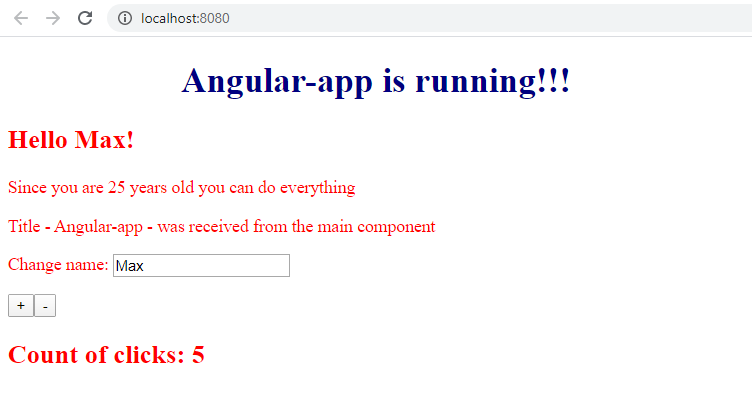
}

<h1>{{ title }} is running!!!</h1>

<child-comp [title]="title" (onChanged)="getData($event)"></child-comp>

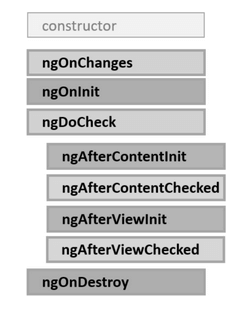
<h2>Count of clicks: {{clicks}}</h2>

В итоге получим;



# Жизненный цикл компонента

После создания компонента фреймворк Angular вызывает у этого компонента ряд методов, которые представляют различные этапы жизненного цикла:



* **ngOnChanges**: вызывается до метода ngOnInit() при начальной установке свойств, которые связаны механизмом привязки, а также при любой их переустановке или изменении их значений. Данный метод в качестве параметра принимает объект класса SimpleChanges, который содержит предыдущие и текущие значения свойства.
* **ngOnInit**: вызывается один раз после установки свойств компонента, которые участвуют в привязке. Выполняет инициализацию компонента
* **ngDoCheck**: вызывается при каждой проверке изменений свойств компонента сразу после методов ngOnChanges и ngOnInit
* **ngAfterContentInit**: вызывается один раз после метода ngDoCheck() после вставки содержимого в представление компонента кода html
* **ngAfterContentChecked**: вызывается фреймворком Angular при проверке изменений содержимого, которое добавляется в представление компонента. Вызывается после метода ngAfterContentInit() и после каждого последующего вызова метода ngDoCheck().
* **ngAfterViewInit**: вызывается фреймворком Angular после инициализации представления компонента, а также представлений дочерних компонентов. Вызывается только один раз сразу после первого вызова метода ngAfterContentChecked()
* **ngAfterViewChecked**: вызывается фреймворком Angular после проверки на изменения в представлении компонента, а также проверки представлений дочерних компонентов. Вызывается после первого вызова метода ngAfterViewInit() и после каждого последующего вызова ngAfterContentChecked()
* **ngOnDestroy**: вызывается перед тем, как фреймворк Angular удалит компонент.

Каждый такой метод определен в отдельном интерфейсе, который называется по имени метода без префикса "ng". Например, метод ngOnInit определен в интерфейсе OnInit. Поэтому, если мы хотим отслеживать какие-то этапы жизненного цикла компонента, то класс компонента должен применять соответствующие интерфейсы:

import {

    Component,

    Input,

    OnInit,

    DoCheck,

    OnChanges,

    AfterContentInit,

    AfterContentChecked,

    AfterViewChecked,

    AfterViewInit

} from '@angular/core';

@Component({

    selector: 'child-comp',

    template: `<p>Hello {{name}}</p>`

})

export class ChildComponent implements OnInit,

    DoCheck,

    OnChanges,

    AfterContentInit,

    AfterContentChecked,

    AfterViewChecked,

    AfterViewInit {

    @Input() name: string;

    count: number = 1;

    ngOnInit() {

        this.log(`ngOnInit`);

    }

    ngOnChanges() {

        this.log(`OnChanges`);

    }

    ngDoCheck() {

        this.log(`ngDoCheck`);

    }

    ngAfterViewInit() {

        this.log(`ngAfterViewInit`);

    }

    ngAfterViewChecked() {

        this.log(`ngAfterViewChecked`);

    }

    ngAfterContentInit() {

        this.log(`ngAfterContentInit`);

    }

    ngAfterContentChecked() {

        this.log(`ngAfterContentChecked`);

    }

    private log(msg: string) {

        console.log(this.count + ". " + msg);

        this.count++;

    }

}

Внесем изменения в файл app.module.ts:

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

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

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

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

import { ChildComponent } from './app.child.component';

@NgModule({

    imports: [BrowserModule, FormsModule],

    declarations: [AppComponent, ChildComponent],

    bootstrap: [AppComponent]

})

export class AppModule { }

в файл app.component.ts:

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

@Component({

    selector: 'my-app',

    templateUrl: `./app.component.html`,

    styles: [`

        h2 {text-align: center}

    `]

})

export class AppComponent {

    name = "Nick";

}

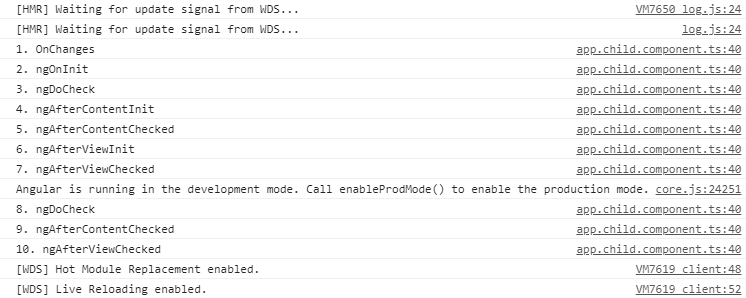
И используем дочерний компонент в главном компоненте, изменив представление:

<h2>Hello Angular! Welcome Webpack!</h2>

<child-comp [name]="name"></child-comp>

<input type="text" [(ngModel)]="name">

И при обращении к приложению мы получим следующую цепочку вызовов:



А при изменении поля ввода главного компонента:



# Шаблонные переменные, ViewChild

Шаблонные переменные позволяют определить некоторые переменные внутри шаблона компонента и затем ссылаться к этим переменным из этого же шаблона. Для определения подобных переменных применяется знак решетки (#). Например, определим шаблонную переменную date в главном компоненте:

<button (click)="onClick(date)">Show date</button>

<p #date></p>

Определение шаблонной переменной date в элементе параграфа означает, что она будет представлять данный параграф, то есть элемент p разметки html. И далее мы можем обращаться к этому параграфу через данную переменную.

Добавим обработчик события onClick в главный компонент:

export class AppComponent {

    name = "Nick";

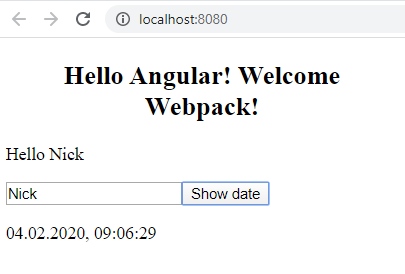
    onClick(date: any) {

       date.innerHTML = (new Date()).toLocaleString();

    }

}

В результате нажатия кнопки в приложении изменится содержимое абзаца и выведется текущая дата:



Использование шаблонных переменных открывает нам дополнительный способ взаимодействия между родительским и дочерним компонентом. Например, определим следующий дочерний компонент ChildComponent:

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

@Component({

    selector: 'child-comp',

    template: `<p>Couinter: {{counter}}</p>`

})

export class ChildComponent {

    counter: number = 0;

    increment() { this.counter++; }

    decrement() { this.counter--; }

}

В этом компоненте определяется переменная счетчика counter. Для ее увеличения или уменьшения применяются методы increment и decrement.

В коде главного компонента будем вызывать дочерний компонент:

<child-comp #counter></child-comp>

<button (click)="counter.increment()">+</button>

<button (click)="counter.decrement()">-</button>

В данном случае шаблонная переменная counter, определенная внутри тега <child-comp>, поэтому она будет представлять компонент ChildComponent.

Соответственно далее мы можем ссылаться к компоненту ChildComponent через эту переменную, например, установить для событий кнопок привязку к методам ChildComponent. В итоге по нажатию на кнопки в главном компоненте будут вызываться методы из дочернего компонента:



# ViewChild

Шаблонные переменные имеют свои ограничения: они не могут применяться вне шаблона, даже в коде класса компонента.

Чтобы все-таки иметь возможность обращаться к методам и другой функциональности дочернего компонента используется декоратор ViewChild. Данный декоратор применяется к свойству и получает селектор элемента DOM, который необходимо отслеживать. И если отслеживаемый по селектору элемент изменяется, то ViewChild изменяет состояние свойства. Так, изменим главный компонент следующим образом:

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

import { ChildComponent } from './app.child.component';

@Component({

    selector: 'my-app',

    template: `<child-comp></child-comp>

                <button (click)="increment()">+</button>

                <button (click)="decrement()">-</button>`

})

export class AppComponent {

    @ViewChild(ChildComponent, { static: false })

    private counterComponent: ChildComponent;

    increment() { this.counterComponent.increment(); }

    decrement() { this.counterComponent.decrement(); }

}

Первый параметр декоратора ViewChild указывает на селектор элемента, который будет отслеживаться. В качестве селектора может использоваться класс с декоратором @Component, то есть класс компонента, например, ChildComponent. Второй параметр - static - указывает, как будет производиться отслеживание изменений.

C помощью декоратора ViewChild также можно связать свойство и переменную из шаблона. Так, изменим код главного компонента:

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

@Component({

    selector: 'my-app',

    template: `<p #nameText>{{name}}</p>

               <p>{{nameText.textContent}}</p>

               <button (click)="change()">Change</button>`

})

export class AppComponent {

    @ViewChild("nameText", { static: false })

    nameParagraph: ElementRef;

    name = "Nick";

    change() {

        console.log(this.nameParagraph.nativeElement.textContent);

        this.nameParagraph.nativeElement.textContent = "Alex";

    }

}

Здесь в шаблоне определяется переменная nameText, которая представляет код параграфа. А в декоратор ViewChild передается имя этой переменной. Поэтому свойство nameParagraph, к которому применяется декоратор, будет указывать на эту переменную nameText. Причем свойство nameParagraph представляет тип ElementRef, который используется для ссылки на элементы html.

По нажатию на кнопку выводится и изменяется текстовое содержимое этой переменной.

# Взаимодействие между модулями

В приложении Angular мы можем определять и использовать множество модулей. Нередко модуль создается для объединения компонентов и других классов, которые работают с каким-то одним аспектом приложения. И если приложение большое, то логично разбить его на модули, которые выполняют различные задачи. Однако в этой связи может возникнуть вопрос, как подключать и использовать одни модули в других.

Для создания модуля используется декоратор @NgModule, а в его свойствах указываются зависимости и экспортируемые компоненты.

Определим класс модуля DataModule в файле data.module.ts:

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

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

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

import { ChildComponent }   from './app.child.component';

@NgModule({

    imports:      [ BrowserModule, FormsModule ],

    declarations: [ ChildComponent ],

    exports: [ ChildComponent ]       // экспортируем компонент

})

export class DataModule { }

В данном модуле мы можем подключать другие модули, которые мы собираемся использовать рамках текущего модуля и его компонентов и директив.

В секции declarations добавляем компонент DataComponent, который будет принадлежать данному модулю. Но чтобы этот компонент можно было подключать в других модулях, DataComponent также добавляется в секцию exports.

Для доступа к модулю извне его необходимо импортировать в главном модуле app.module.ts:

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

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

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

import { ChildComponent } from './app.child.component';

@NgModule({

    imports: [BrowserModule, FormsModule],

    declarations: [ChildComponent],

    exports: [ChildComponent]       // экспортируем компонент

})

export class DataModule { }

Модули, которые определили мы сами, подключаются также, как и все остальные модули в секции imports.

После этого мы сможем использовать функциональность DataModule в компонентах, которые принадлежат AppModule. Например, используем DataComponent в компоненте AppComponent в файле app.component.ts.

# Директивы

Директивы определяют набор инструкций, которые применяются при рендеринге html-кода. Директива представляет класс с директивными метаданными. В TypeScript для прикрепления метаданных к классу применяется декоратор @Directive.

В Angular есть три типа директив:

* **Компоненты**: компонент по сути также является директивой, а декоратор @Component расширяет возможности декоратора @Directive с помощью добавления функционала по работе с шаблонами.
* **Атрибутивные**: они изменяют поведение уже существующего элемента, к которому они применяются. Например, ngModel, ngStyle, ngClass
* **Структурные**: они изменяют структуру DOM с помощью добавления, изменения или удаления элементов html. Например, это директивы ngFor и ngIf

**Директива ngClass** позволяет определить набор классов, которые будут применяться к элементу. В качестве значения она принимает набор классов в следующем виде:

[ngClass]={

"класс1": true/false,

"класс2": true/false,

...................

}

Например, определим следующий компонент:

@Component({

    selector: 'my-app',

    template: `<p #nameText [ngClass]="{header: true}">{{name}}</p>

               <p [ngClass]="{simple:true}">{{nameText.textContent}}</p>

               <button (click)="change()">Change</button>`,

    styles: [`

        .header {font-size: 3em; color: navy; text-align: center}

        .simple {color: darkgrey}

    `]

})

в секции styles определяет два класса, которые устанавливают различные стили: header и simple.

В шаблоне для привязки класса к элементу применяется директива [ngClass]="{header: true}". Эта директива принимает js-объект, в котором ключи – это названия классов. Этим названиям присваиваются булевые значения true (если класс применяется) и false (если класс не применяется).

**Директива ngStyle** позволяет задать набор стилей, которые применяются к элементу. В качестве значения директива принимает js-объект, в котором ключи – названия свойств CSS:

@Component({

    selector: 'my-app',

    template: `<p #nameText [ngClass]="{header: true}">{{name}}</p>

               <p [ngClass]="{simple:true}">{{nameText.textContent}}</p>

               <button (click)="change()" [ngStyle]="{'font-size':'14px', 'color':'yellow', 'background':'brown'}">Change</button>`,

    styles: [`

        .header {font-size: 3em; color: navy; text-align: center}

        .simple {color: darkgrey}

    `]

})

# Создание атрибутивных директив

Атрибутивные директивы меняют поведение элемента, к которому они применяются. Например, директива ngClass позволяет установить для элемента класс CSS. При этом сама директива применяется к элементу в виде атрибута. Но при необходимости мы можем сами создавать какие-то свои директивы атрибутов для каких-то определенных целей. Итак, создадим свою директиву. Добавим в папку src/app новый файл, который назовем bold.directive.ts:

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

@Directive({

    selector: '[bold]'

})

export class BoldDirective {

    constructor(private elementRef: ElementRef) {

        this.elementRef.nativeElement.style.fontWeight = "bold";

    }

}

Директива – это обычный класс на TS, к которому применяется декоратор Directive, соответственно нам надо импортировать эту директиву из "angular/core". Кроме того, здесь импортируется класс "ElementRef". Он представляет ссылку на элемент, к которому будет применяться директива.

При применении декоратора @Directive необходимо определить селектор CSS, с которым будет ассоциирована директива. Селектор CSS для атрибута должен определяться в квадратных скобках. В данном случае в качестве селектора выступает [bold].

Сам декоратор @Directive применяется к классу, который называется BoldDirective. Это собственно и есть класс директивы, который определяет ее логику.

Для получения элемента, к которому применяется данная директива, в классе определен конструктор, имеющий один параметр: private elementRef: ElementRef. Через этот параметр Angular будет передавать или инжектировать тот элемент из шаблона, в котором применяется директива.

Поскольку параметр определен с ключевым словом private, то для него будет создаваться одноименная приватная переменная, через которую мы можем получить объект ElementRef и произвести с ним какие-либо манипуляции. В частности, здесь идет обращение к вложенному свойству nativeElement, через которое у элемента устанавливается жирный шрифт.

Директива применяется как

 <p bold>Bold in paragraph</p>

Но сама по себе директива не заработает. Нам еще надо ее подключить в модуле приложения – классе AppModule:

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

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

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

import { DataModule } from './data.module';

import { BoldDirective } from './bold.directive';

@NgModule({

    imports: [BrowserModule, DataModule],

    declarations: [AppComponent, BoldDirective],

    bootstrap: [AppComponent]

})

export class AppModule { }

# Взаимодействие с пользователем, HostListener и HostBinding

Кроме простой установки значений атрибутивная директива может взаимодействовать с пользователем. Для этого применяется декоратор **HostListener**.

Добавим в директиву BoldDirective взаимодействие с пользователем:

import { Directive, ElementRef, Renderer2, HostListener } from '@angular/core';

@Directive({

    selector: '[bold]'

})

export class BoldDirective {

    constructor(private elementRef: ElementRef, private renderer: Renderer2) {

        this.elementRef.nativeElement.style.fontWeight = "bold";

        this.elementRef.nativeElement.style.color = "green";

    }

    @HostListener("mouseenter") onMouseEnter() {

        this.setFontWeight("bold");

    }

    @HostListener("mouseleave") onMouseLeave() {

        this.setFontWeight("normal");

    }

    private setFontWeight(val: string) {

        this.renderer.setStyle(this.elementRef.nativeElement, "font-weight", val);

    }

}

Декоратор @HostListener позволяет связать события DOM и методы директивы. В частности, в декоратор передается название события, по которому будет вызываться метод. В данном случае мы привязываем события mouseenter (наведения указателя мыши на элемент) и mouseleave (уведение указателя мыши с элемента) к методу setFontWeight(), который устанавливает стилевое свойство font-weight у элемента. Если мы наводим на элемент, то устанавливается выделение жирным. При отводе мыши выделение сбрасывается.

Кроме кода директивы менять больше ничего не надо, код компонента и модуля остаются теми же.

Еще один декоратор – **HostBinding** позволяет связать обычное свойство класса со свойством элемента, к которому применяется директива. Например, изменим код директивы следующим образом:

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

@Directive({

    selector: '[bold]'

})

export class BoldDirective {

    private fontWeight = "normal";

    @HostBinding("style.fontWeight") get getFontWeight() {

        return this.fontWeight;

    }

    @HostBinding("style.cursor") get getCursor() {

        return "pointer";

    }

    @HostListener("mouseenter") onMouseEnter() {

        this.fontWeight = "bold";

    }

    @HostListener("mouseleave") onMouseLeave() {

        this.fontWeight = "normal";

    }

}

Инструкция @HostBinding("style.fontWeight") get getFontWeight() связывает со свойством "style.fontWeight" значение, которое возвращается этим геттером getFontWeight. А он возвращает значение свойства fontWeight, которое также меняется при наведении указателя мыши.

Вместо применения декораторов HostListener и HostBinding для реагирования директивы на действия пользователя мы можем определить обработчики событий в декораторе Directive с помощью его свойства **host**. Так, перепишем директиву следующим образом:

import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({

    selector: '[bold]',

    host: {

        '(mouseenter)': 'onMouseEnter()',

        '(mouseleave)': 'onMouseLeave()'

    }

})

export class BoldDirective {

    constructor(private element: ElementRef, private renderer: Renderer2) {

        this.renderer.setStyle(this.element.nativeElement, "cursor", "pointer");

    }

    onMouseEnter() {

        this.setFontWeight("bold");

    }

    onMouseLeave() {

        this.setFontWeight("normal");

    }

    private setFontWeight(val: string) {

        this.renderer.setStyle(this.element.nativeElement, "font-weight", val);

    }

}

Результат работы директивы в данном случае будет аналогичен

# Структурные директивы ngIf, ngFor, ngSwitch

Структурные директивы изменяют структуру DOM с помощью добавления или удаления html-элементов. Рассмотрим три структурных директивы: ngIf, ngSwitch и ngFor.

**Директива ngIf** позволяет удалить или, наоборот, добавить элемент при определенном условии. Например, определим следующий компонент:

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

@Component({

    selector: 'my-app',

    template: `<p \*ngIf="condition">

                  Hello world!!!

                </p>

                <p \*ngIf="!condition">

                  Good by world!!!

                </p>

                <button (click)="toggle()">Toggle</button>`

})

export class AppComponent {

    condition: boolean = true;

    toggle() {

        this.condition = !this.condition;

    }

}

В зависимости от значения свойства condition будет отображаться либо первый, либо второй параграф.

**Директива ngFor** позволяет перебрать в шаблоне элементы массива. Например:

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

@Component({

    selector: 'my-app',

    template: `<ul>

                  <li \*ngFor="let item of items">{{item}}</li>

                </ul>

                <div>

                    <p \*ngFor="let item of items; let i = index">{{i+1}}.{{item}}</p>

                </div>

                `

})

export class AppComponent {

    items = ["Apple iPhone 7", "Huawei Mate 9", "Samsung Galaxy S7", "Motorola Moto Z"];

}

В качестве значения директива принимает значение перебора в стиле foreach: let item of items. Каждый перебираемый элемент помещается в переменную item, которую мы можем вывести на страницу. При переборе элементов нам доступен текущий индекс элемента через переменную index, которую мы также можем использовать.

С помощью директивы **ngSwitch** можно встроить в шаблон конструкцию switch...case и в зависимости от ее результата выполнения выводить тот или иной блок. Например:

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

@Component({

    selector: 'my-app',

    template: `<div [ngSwitch]="count">

                  <ng-template ngSwitchCase="1">{{count \* 10}}</ng-template>

                  <ng-template ngSwitchCase="2">{{count \* 100}}</ng-template>

                  <ng-template ngSwitchDefault>{{count \* 1000}}</ng-template>

                </div>`

})

export class AppComponent {

    count: number = 5;

}

Директива ngSwitch в качестве значения принимает некоторое выражение. В данном случае это свойство count. В элемент ng-template помещается инструкция ngSwitchCase, которая сравнивает значение выражения из ngSwitch с другим выражением. Если оба выражения равны, то используется данный элемент template. Иначе выполнение переходит к следующим инструкциям ngSwitchCase. Если же ни одна из инструкций ngSwitchCase не была выполнена, то вызывается инструкция ngSwitchDefault.