**Angular**

Typescript:

Typescriptlang.org/docs/handb…

* To install globally via npm: (but preferably install locally)

In the Terminal of your file (test.ts) in VS Code:

npm install -g typescript / npm install typescript -g

npm uninstall typescript -g (removes the package)

* To install locally

npm init (to initialize the project first) => package.json is created

npm install typescript => node\_modules is created

in package.json under dependencies: typescript is shown

* To check your version of the installed typesctipt:

tsc --version

* To compile your code: (compilation output is plain JavaScript => test.js)

tsc ./test.ts (typescript compiler on test.ts file)

* When installed locally to get access to tsc you need to use npx instead of npm (which has locally installed typescript that will run it)

npx tsc test.ts

Or use: ./node\_modules/.bin/tsc ./test.ts

Or in package.json scripts: “build”: “tsc test.ts” and then npm run build

* When compiling we can set certain configurations for the output file. Those can be stated in the Terminal command or all in a separate tsc config file

Angular:

Blog.angular.io

Creating a new app

* Install globally via npm

npm install -g @angular/cli@latest=> creates ng command that can be used further

ex. ng --version

* Create new project

ng new some-app

cd some-app (enter the directory)

code .(open the app in VS Code)

* Start dev server on port 4200

ng serve / ng s => localhost:4200

* Building the app in production environment

ng build -c production

* When we need to refresh vs code, press ctrl+P, in the field type “>reload window” (if not working install Angular language service plug-in) (angular.json “strict:true”)
* Continue: angular project

Files in Angular project:

* tsconfig.json – settings/configuration how the typescript files are compiled, also how the angular is compiling tsc to js
* tsconfig.app.json – it is extending the tsconfig.json file and is used for our app
* tsconfig.spec.json – it is extending the tsconfig.json file and is used for running tests of our app
* angular.js – configuration for the angular cli, how our project is build, if any additional scripts to be included in the bundle; external libraries we list in scripts, like google maps example. File replacement is mostly used for “environments” viriables. Also the created components prefix can be changed from prefix: ‘app’ to anything
* editorconfig – for the IDE
* browserslistcr – for which browsers is our app building
* app – the main component
* app -> styles.css – if we want a style to be applied globally, this is where we write it
* app -> main.ts – shows in what environment is our app working (browser, server, other)
* app -> pollyfills.ts – tracks async operations in our app
* Index.html – the starting point of our app
* In the Terminal ng build => scripts are built in index.html > scripts (under app-root) and our app is bootstrapped
* App -> app.module.ts – a class with Ng decorator that connects all the elements in this module
  + declarations(stores all the components that we want to use in the module)
  + imports – browser module for browser app, that contains browser specific things that we use to run our app
  + providers – array of different service providers
  + bootstrap – the component that we want to use to bootstrap or to start the app

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

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

**import { AppRoutingModule } from './app-routing.module';**

**@NgModule({**

**declarations: [**

**AppComponent**

**],**

**imports: [**

**BrowserModule,**

**AppRoutingModule**

**],**

**providers: [],**

**bootstrap: [AppComponent]**

**})**

**export class AppModule { }**

* continue

Using Google maps example: by importing the google.maps api

* In angular.json => “scripts”: [“ ” ->/\*link for the google maps location js file\*/]
* In index.html => after body we can add the script that will load the location

Components:

* A component is a class with Component decorator and contains 3 properties (selector, templateUrl, syleUrls). Or it can be called a directive with a specific template
* Generated in the terminal with ng g c component-name(ex. user-list-item)

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

**@Component({**

**selector: 'app-root',**

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

**styleUrls: ['./app.component.css']**

**})**

**export class AppComponent {**

**title = 'Article Site';**

**}**

* Creating a component:
  + when manually creating component files (test.component.ts, .html, .css) it has to be registered in app.module.ts> NgModule > declarations > TestComponent(the exported class name in the component.ts file) => this way we can use <app-test> selector in the .html files
  + when using the Angular CLI: ng generate component home (cd is articles-app that was created with ng new articles-app > cd articles-app)

a new folder is created scr/app/home and the cli directly imports the component in the app module

Data binding: in the .html file

* Using interpullation with {{}}, ex. {{title}} – used only for showing text somewhere in our template, i.e a property that has string value
* Using [] for property values, ex. <input [value] = “title”> </input>, i.e we give value to the input that is taken from the title property of the class in the .component.ts file
* Using () for event change, ex. <button (click)=”buttonClickHandler()”>Click Here</button>, where in the class in the .component.ts file we need to create the function for this event, i.e. buttonClickHandler(): void{ console.log(‘Button was clicked’);}

Templates – a form of HTML that tells Angular how to render the component. \*ngIf and \*ngFor come from the app.module.ts>imports>BrowserModule, that contains all common modules such as ngIf and ngFor and others

* To render nested properties of an object
* Using \*ngFor to render array properties

**export class GamesComponent {**

**games : Game[];  
 constructor() {**

**this.games = [ *// Array of games* ]**

**}**

**}**

**<h1>Games List</h1>**

**<p>Pick a game to Buy</p>**

**<ul>**

**<li \*ngFor="let game of games">**

**{{game.title}}**

**</li>**

**</ul>**

* Using \*ngIf for condition statements

**<h1>Games List</h1>**

**<p>Pick a game to Buy</p>**

**<ul>**

**<li \*ngFor="let game of games">**

**<div>**

**{{game.title}}**

**</div>**

**<span \*ngIf="game.price >= 100">**

**Price: {{game.price}}**

**</span>**

**</li>**

**</ul>**

* To attach events and handle them in the component

**<button (click)="showContent($event)">Show Content</button>**

**export class GamesComponent {**

**public games: Game[];**

**showContent: boolean;**

**constructor() {**

**this.games = [ *// Array of games* ]**

**}**

**showAdditionalContent($event) {**

**this.showContent = true;**

**}**

**}**

* To reference other elements

**<input #phone placeholder="phone number">**

**<button (click)="callPhone(phone.value)">Call</button>**

callPhone(phone: HTMLInputElement){

const {value: phoneNumber} = phone;

phone.value = “”;

}

Home.component.html

<div \*ngFor="let item of toDoList">

    <todo-list-app-item [toDoItem] = "item"></todo-list-app-item>

</div> //we pass the value that the item.ts will need from the home html

Item.component.ts

export class ItemComponent implements OnInit {

  @Input() toDoItem!: IToDoItem;

  constructor() { }

  ngOnInit(): void {

  }

  handleStatusChange(item: IToDoItem): void {

    item.isCompleted = !item.isCompleted;

  }

}

When using @Output() addUser = new EventEmitter<IUser>();

This event emitter has to be imported from @angular/cli

Also in the app component .ts we need to add handler for this emitter, i.e.

addNewUserHandler(newUser: IUser): void{

this.users.push(newUser);

}

In app component .html we also add the handler

<app-user-list [userArray] = “users” (addUser) = “addNewUserHandler($event)”></app-user-list>

<https://github.com/IliaIdakiev/softuni-angular-07-21>

<https://github.com/IliaIdakiev/softuni-angular-07-21/tree/master/2/lecture2/src/app>

Text

Description automatically generated

* To use null-safe operator

**<div>The current hero's name is {{game?.title}}</div>**

**<div>The null hero's name is {{game && game.name}}</div>**

All lifecycle hooks

* + **ngOnInit() –** when component is created, at this stage only static elements are rendered, all @Input() values will be taken at this stage
  + **ngAfterViewInit()** – when the whole view has been rendered
  + **ngOnDestroy()**  - when we want to remove the component
  + **ngOnChanges**() - when data is changed
  + **ngDoCheck**() - detect your own changes
  + **ngAfterContentInit**() - when external content is received
  + **ngAfterContentChecked**() - when external content is checked
  + **ngAfterViewInit**() - when the views and child views are created
  + **ngAfterViewChecked**() - when the above are checked

More at: <https://angular.io/guide/lifecycle-hooks>

Component Interaction:

* @Input() user!: IUser; which is the same as user: IUser | undefined
* @Input() decorator – used for transferring data from parent to child
* @Output() decorator – used for transferring data from child to parent

SOLID:

<https://www.youtube.com/watch?v=QHnLmvDxGTY&t=760s>

Solid principles by uncle Bob

* Single Responsibility Principle – every class should have only one responsibility, i.e. only one reason to change
* Open-Closed Principle – every class should be open for extension and closed for modification
* Liskov Substitute Principle – derived classes only extend the functionality of the base class but don’t remove functions
* Interface Segregation Principle - Classes that implement interfaces, should not be forced to implement methods they do not use
* Dependency Inversion Principle - Dependency Injection is a popular design pattern. Inversion of control (IoC) – the dependencies are pushed in the constructor and the class does not instantiate it’s dependencies

Change Detection strategy:

* Angular uses zone.js (pollyfill.ts) to detect events(asynchronous) that invoke changes, such as:
* const xhr = new XMLHttpRequest();
* setTimeout
* setInterval
* Promise.resolve()
* AddEventListener()
* Angular performs change detection on all components (from top to bottom) every time something changes
* **OnPush: 0** -CheckOne strategy
* Automatic change detection is deactivated until reactivated by setting the strategy to Default
* This strategy applies to all child directives and cannot be overridden
* **Default: 1** -CheckAlways strategy
* Use the default CheckAlways strategy
* Change detection is automatic until explicitly deactivated

Services:

* Services in Angular are just normal TypeScript classes that   
  handle data manipulation
* Components **shouldn't** fetch or save **data** directly. They should focus on **presenting data** and delegate data access to a service
* Services are a great way to share information among classes that **don't know**  
  about each other and also avoid **code duplication**
* In services there is also one lifecycle hook, that is ngOnDestroy():void{}, which is needed when the service is provided directly in the component and later this component is destroyed
* Services are injected into components via **constructor injection.** Before that they should be **provided** from inside the **decorator**
* The first and most general place where providers could be stated is the main.ts file > platformBrowserDynamic and also in >BootstarpModule> providers; this service will be available all the way down through the tree
* Next place is in the app.module.ts > providers: [{

Provide: UserService,

//useClass: UserService or

useValue: {

users; [{name: ‘Gosho’, age: 20}],

addNewuserHandler() {alert(‘Ne!’);}]

* Next is app.compoment.ts > @Component{…,providers:[]}; - the important point here is that when the component is destroyed all dependencies are also destroyed, i.e providers, and when it’s create they are also created. For this reason we cannot reply on consistency here
* Next place is the service.ts class by decorating it @Injectable{providedIn: ‘root’};

RxJS – reactive extensions for JavaScript:

* Install library with npm install rxjs
* import {of} from ‘rxjs’; or const {of} = require(‘rxjs’);

import {map} from ‘rxjs/operators’;

* observables are defined by the key word ‘of’
* observables are lazy by default, we use .subscribe to start them
* piping is done with the key word ‘pipe’, instead of .then like in promises, i.e.

Promise.resolve(1000).then(x=> x+1).then(console.log);

of(1000).pipe(map(x=> x+1)).subscribe{(x) => {console.log(x); )};

* cold and hot observables – movie example (in airplane vs in cinema), cold (of, from, range, interval and timer) vs hot(fromEvent)
* tap and console.log aew so called ‘side effects’, tap is like map but does not return result, it is undefined and is commonly used for debugging
* commonly used rxjs operators – map, filter, reduce

**const obs = range(1, 10).pipe(map(i => i \*\* 2));**

**const obs = range(1, 10).pipe(filter(i => i % 2 === 0));**

**const obs = range(1, 10).pipe(reduce((prevVal, val) => prevVal + val, 0)))**

Http Client – fetching data from a remote api:

* in app.module.ts – **import { HttpClientModule } from '@angular/common/http'** and add the module in imports array

**@NgModule({**

**declarations:[ *// App Components* ],**

**imports:[**

**BrowserModule,**

**HttpClientModule**

**],**

* using the Http Client Service:

**@Injectable()**

**export class PostsService {  
 constructor(**

**private http : HttpClient**

**) { }**

**getAllPosts() : Observable<Post[]> {**

**const url = 'https://jsonplaceholder.typicode.com/posts';**

**return this.http.get<Post[]>(url);**

**}**

**}**

Formatting by Ilia Idakiev:

* removing the “” from a copied text

search “id” > in the opened box we do a replace where in Find we click the .\* box and we type a regular expression “(.\*)”: and in Replace we take only the group with $1: