# Source of Learning

<http://www.pragimtech.com/kudvenkat_angular2_tutorial_videos_download.aspx>

1. How app loads
   1. Bootstrapping
   2. How your html renders on browser
2. Interpolation
3. Property binding
4. Tow way binding
5. Basic building blocks of Angular
   1. Component
6. Decorator
   * 1. ii. Class
     2. Different properties/options
   1. Service
   2. Directive
   3. Pipe
7. Life hooks of Component/Directive
8. Basics of TS
9. Communication between two components
   1. Input/Output
   2. Observable theory
   3. Event emitters
   4. Rxjs - (Reactive Extensions)
   5. Observable theory + actual implementation after reading/implementating RxJs
10. Routing
11. What is modulazations in Angular
    1. Concept
    2. Lazy loading of Modularization
12. Build & Deployment strategy
    1. AoT - Ahead of Time Compilation
13. Different versions of Angular 2/4/5/6
14. Dependency injection in Angular
15. Jasmine Tests

# Features

* Progresive web app
* Perfomance
* Type Script
* Lazy Loading
* Forms
* RXJS
* Fully featured router
* Animation
* Server side rendering
* Mobile friendly
* Angular Language Service
* ngUpgrate Liabrary

# Property binding in Angular 2

bound imagePath property of the component class to <img> element src property using interpolation

<img src='{{imagePath}}'/>

Both Interpolation and Property binding flows a value in one direction, i.e from a component's data property into a target element property.

## What is the difference between Interpolation and Property binding

* Interpolation is a special syntax that Angular converts into a property binding.
* Interpolation is just a convenient alternative to property binding.
* In some cases like when we need to concatenate strings we have to use interpolation instead of property binding as shown in the example below.

<img src='http://www.pragimtech.com/{{imagePath}}' />

* When setting an element property to a non-string data value, you must use property binding. In the following example, we are disabling a button by binding to the boolean property isDisabled.

<button [disabled]='isDisabled'>Click me</button>

* We can also use the alternate syntax with bind- prefix. This is known as canonical form

<button bind-disabled='isDisabled'>Click me</button>

* Angular interpolation sanitizes the malicious content and displays the following in the browser

Hello <script>alert("Hacked");</script> World

In this example below we are using property binding.

'<div [innerHtml]="badHtml"></div>'

* Property binding sanitizes the malicious content slightly differently and we get the following output, but the important point to keep in mind is both the techniques protect us from malicious content and render the content harmlessly.

Hello alert("Hacked"); World

# HTML DOM

* DOM contains the HTML elements as objects, their properties, methods and events and it is a standard for accessing, modifying, adding or deleting HTML elements.
* Angular data-binding is all about binding to DOM object properties and not HTML element attributes.
* **What is the difference between HTML element attribute and DOM property**
* Attributes are defined by HTML, where as properties are defined by the DOM.
* Attributes initialize DOM properties. Once the initialization complete, the attributes job is done.
* Property values can change, where as attribute values can't.

# DOM event list

<https://developer.mozilla.org/en-US/docs/Web/Events>

# Basic data type in anguler 2

<https://dzone.com/articles/what-are-the-basic-data-types-in-typescript>

# Understanding the File Structure

app/app.component.ts - this is where we define our root component

app/app.module.ts - the entry Angular Module to be bootstrapped

index.html - this is the page the component will be rendered in

app/main.ts - is the glue that combines the component and page together

* **app/app.component.ts**

import { Component } from '@angular/core'

@Component({

selector: 'app-root',

template: '<b>Bootstrapping an Angular Application</b>'

})

export class AppComponent { }

* **app/app.module.ts**

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

import { NgModule } '@angular/core';

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

@NgModule({

imports: [BrowserModule],

declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule {

}

* **app/main.ts**

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

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

platformBrowserDynamic().bootstrapModule(AppModule);

# Bootstrapping process:

The bootstrap process loads main.ts which is the main entry point of the application. The AppModule operates as the root module of our application. The module is configured to use AppComponent as the component to bootstrap, and will be rendered on any app-root HTML element encountered.

There is an app HTML element in the index.html file, and we use app/main.ts to import the AppModule component and the platformBrowserDynamic().bootstrapModule function and kickstart the process. As shown above, you may optionally use AoT in which case you will be working with Factories, in the example, AppModuleNgFactory and bootstrapModuleFactory.

Why does Angular bootstrap itself in this way? Well there is actually a very good reason. Since Angular is not a web-only based framework, we can write components that will run in NativeScript, or Cordova, or any other environment that can host Angular applications.

The magic is then in our bootstrapping process - we can import which platform we would like to use, depending on the environment we're operating under. In our example, since we were running our Angular application in the browser, we used the bootstrapping process found in @angular/platform-browser-dynamic.

It's also a good idea to leave the bootstrapping process in its own separate main.ts file. This makes it easier to test (since the components are isolated from the bootstrap call), easier to reuse and gives better organization and structure to our application.

# Angular2 event binding

* **JS code:**

export class EmployeeComponent {

columnSpan: number = 2;

firstName: string = 'Tom';

lastName: string = 'Hopkins';

gender: string = 'Male';

age: number = 20;

showDetails: boolean = false;

toggleDetails(): void {

this.showDetails = !this.showDetails;

}

}

* **HTML code:**

<button (click)='toggleDetails()'>

{{showDetails ? 'Hide' : 'Show'}} Details

</button>

# Tow way binding:

* + Two-way data binding in Angular is a combination of both Property Binding and Event Binding

Name : <input [value]='name' (input)='name = $event.target.value'>

OR

<input [(ngModel)]='name'>

# \*ngFor:

<tbody>

<tr \*ngFor='let employee of employees; let isFirst = first; let isLast = last; let i=index; let isEven = even; let isOdd = odd'>

<td>{{employee.code}}</td>

<td>{{employee.name}}</td>

<td>{{employee.gender}}</td>

<td>{{employee.annualSalary}}</td>

<td>{{employee.dateOfBirth}}</td>

<td>{{isFirst}}</td>

<td>{{isLast}}</td>

<td>{{i}}</td>

<td>{{isEven}}</td>

<td>{{isOdd}}</td>

</tr>

<tr \*ngIf="!employees || employees.length==0">

<td colspan="5">

No employees to display

</td>

</tr>

</tbody>

</tbody>

# Pipes in Angular:

* Transform data before display
* Built in pipes include lowercase, uppercase, decimal, date, percent, currency etc
* To apply a pipe on a bound property use the pipe character " | "
* Transform data before display
* Built in pipes include lowercase, uppercase, decimal, date, percent, currency etc
* To apply a pipe on a bound property use the pipe character " | "

<td>{{employee.code | uppercase}}</td>

* We can also chain pipes <td>{{employee.dateOfBirth | date:'fullDate' | uppercase }}</td>

Pass parameters to pipe using colon " : "

<td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>

<td>{{employee.dateOfBirth | date:'fullDate'}}</td>

<td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>

# To read more about angular built-in pipes

* **Date** <https://angular.io/api/common/DatePipe>
* **Decimal** <https://angular.io/api/common/DecimalPipe>
* **Currency** <https://angular.io/api/common/CurrencyPipe>
* **Percent** <https://angular.io/api/common/PercentPipe>

# Custom Pipes:

* **Component file:**

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'employeeTitle'

})

export class EmployeeTitlePipe implements PipeTransform {

transform(value: string, gender: string): string {

if (gender.toLowerCase() == "male")

return "Mr." + value;

else

return "Miss." + value;

}

}

* **HTML File:**

<tr \*ngFor='let employee of employees;'>

<td>{{employee.code}}</td>

<td>{{employee.name | employeeTitle:employee.gender}}</td>

<td>{{employee.gender}}</td>

<td>{{employee.annualSalary}}</td>

<td>{{employee.dateOfBirth}}</td>

</tr>

* **Note :** In the filter method we are using tripple equals (===) instead of double equals (==). The table below explains single, double and tripple equals in TypeScript.

Operator Use to

= Assign a value

== Compare two values

=== Compare two values and their types

# Interfaces:

* Use interface keyword to create an interface
* It is common to prefix the interface name with capital letter "I". However, some interfaces in Angular does not have the prefix "I". For example, OnInit interface
* Interface members are public by default and does not require explicit access modifiers. It is a compile time error to include an explicit access modifier. You will see an error message like - public modifier cannot appear on a type member.
* A class that implements an interface must provide implementation for all the interface members unless the members are marked as optional using the ‘?’ operator
* Use the implements keyword to make a class implement an interface
* TypeScript interfaces exist for developer convenience and are not used by Angular at runtime. During transpilation, no JavaScript code is generated for an interface. It is only used by Typescript for type checking during development.
* To reduce the amount of code you have to write, consider using short-hand syntax to initialise class properties with constructor parameters

## Interface declareation:

export interface IEmployee {

Code: string;

FirstName: string;

LastName: string;

Mobile: string;

Age?: number;

Gender: string;

}

## Interface implementation:

import { IEmployee } from './IEmployee';

export class Employee implements IEmployee {

//public FirstName: string;

//public LastName: string;

//public Mobile: string;

//public Age?: number;

//public Gender: string;

//constructor(FirstName: string, LastName: string, Gender: string, Age: number, Mobile: string)

//{

// this.FirstName = FirstName;

// this.LastName = LastName;

// this.Gender = Gender;

// this.Age = Age;

// this.Mobile = Mobile;

//}

// shrotcut

constructor(public Code: string,

public FirstName: string,

public LastName: string,

public Gender: string,

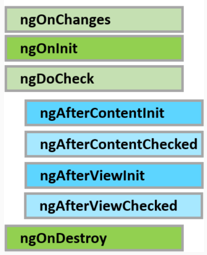
public Age: number,

public Mobile: string) { }

}

# Component Life cycle

* Angular offers several lifecycle hooks as shown in the image below:



* **The 3 most commonly used hooks are**

|  |  |
| --- | --- |
| **Life Cycle Hook** | **Purpose** |
| ngOnChanges | Executes, every time the value of an input property changes. The hook method receives a SimpleChanges object containing current and previous property values. This is called before ngOnInit |
| ngOnInit | Executes after the constructor and after ngOnChange hook for the first time. It is most commonly used for component initialisation and retrieving data from a database |
| ngOnDestroy | Executes just before angular destroys the component and generally used for performing cleanup |

## OnInit and OnDestroy Example

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

@Component({

selector: 'my-app',

templateUrl: './app.component.html'

})

export class AppComponent implements OnInit, OnDestroy {

constructor() { }

ngOnInit() {

console.log('Component Init');

}

ngOnDestroy() {

console.log('Component Destroy');

}

}

## OnDoChange example

.ts

// Step 1 : Import OnChanges and SimpleChanges

import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

// The selector "simple" will be used as the directive

// where we want to use this component. Notice we are

// also using the simpleInput property with interpolation

// to display the value it receives from the parent

// component

@Component({

    selector: 'simple',

    template: `You entered : {{simpleInput}}`

})

// Step 2 : Implement OnChanges Life Cycle Hook interface

export class SimpleComponent implements OnChanges {

    // Input property. As and when this property changes

    // ngOnChanges life cycle hook method is called

    @Input() simpleInput: string;

    // Step 3 : Implementation for the hook method

    // This code logs the current and previous value

    // to the console.

    ngOnChanges(changes: SimpleChanges) {

        for (let propertyName in changes) {

            let change = changes[propertyName];

            let current = JSON.stringify(change.currentValue);

            let previous = JSON.stringify(change.previousValue);

            console.log(propertyName + ': currentValue = '

                + current + ', previousValue = ' + previous);

            // The above line can be rewritten using

            // placeholder syntax as shown below

            // console.log(`${propertyName}: currentValue

            // = ${current }, previousValue = ${previous }`);

        }

    }

}

* .HTML pass parameter to componets

<simple [simpleInput]='userText'></simple>

## ngDoCheck:

ngDoCheck is triggered every time the input properties of a component or a directive are checked.

We can use this lifecycle hook to extend the check with our own custom check logic.

It can also be useful if we want to accelerate the change detection by checking the bare minimum and not using the default algorithm (although we usually do not use this).

## ngAfterContentInit:

The ngAfterContentInit lifecycle hook is called after ngOnInit when the component or directive’s content has been initialised;

basically when all the bindings of the component have been checked for the first time.

## ngAfterContentChecked:

Called after every check of the component or directive’s content, effectively when all the bindings of the components have been checked; even if they haven’t changed.

## ngAfterViewInit:

Called after ngAfterContentInit when the component’s view has been initialised. Applies to components only.

## ngAfterViewChecked:

Called after every check of the component’s view. Applies to components only when all the bindings of the children directives have been checked.

Even if they haven’t changed. It can be useful if the component is waiting for something coming from its child components.

# @Input:- Input parameter to component/ component communication

* **Child component to receive input variable name**

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

@Component({

selector: 'app-name-child',

template: '<h3>"{{name}}"</h3>'

})

export class NameChildComponent {

private \_name = '';

@Input()

set name(name: string) {

this.\_name = (name && name.trim()) || '<no name set>';

}

get name(): string { return this.\_name; }

}

* **Parent component to send parameter to child component**

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

@Component({

selector: 'app-name-parent',

template: `

<h2>Master controls {{names.length}} names</h2>

<app-name-child \*ngFor="let name of names" [name]="name"></app-name-child>

`

})

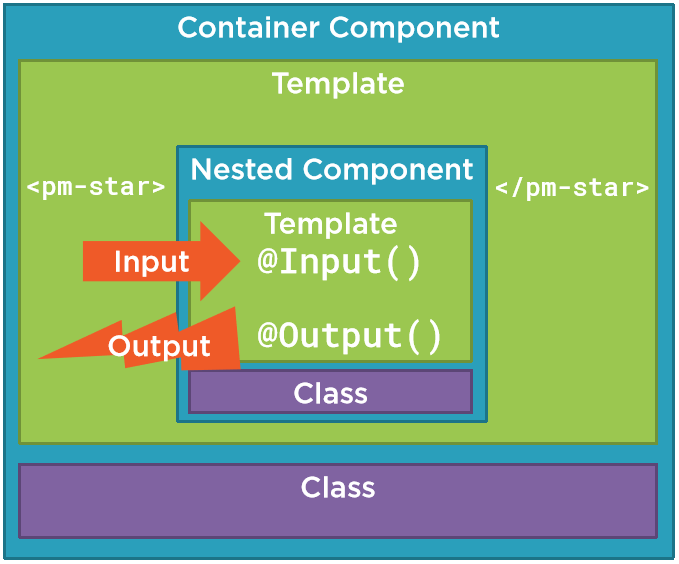
export class NameParentComponent {

// Displays 'Mr. IQ', '<no name set>', 'Bombasto'

names = ['Mr. IQ', ' ', ' Bombasto '];

}

# @Output: Raising event from component



* Parent component html:

<td><app-star [rating]='product.starRating' (ratingClicked)='onRatingClicked($event)'></app-star></td>

* Child component html:

<div class="crop" [style.width.px]="starWidth" [title]="rating" (click)="onClick()">

* Parent component .ts:

onRatingClicked(message: string): void {

this.pageTitle = `Page title: ${message}`;

}

* Child component .ts:

@Output() ratingClicked: EventEmitter<string> = new EventEmitter();

onClick(): void {

this.ratingClicked.emit(`Rating is ${this.rating}`);

}

# Services or @Injectable: Dependency injection

The @Injectable() decorator marks a class a service that can be injected, but Angular can't actually inject it anywhere until you configure an Angular dependency injector with a provider of that service.

A provider tells an injector how to create the service. You must configure an injector with a provider before that injector can create a service (or provide any other kind of dependency).

A provider tells an injector how to create the service. You must configure an injector with a provider before that injector can create a service (or provide any other kind of dependency).

## Service class

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

import { IProduct } from '../products/product';

@Injectable({

providedIn: 'root'

})

//we can inject service in 3 ways

//Inject in app-module using provider

//inject in component using provider

//inject in root module as given in above @injectable providedIn attribute i.e. Platform injector

export class ProductService {

constructor() { }

getProducts(): IProduct[] {

return [{

'productId': 1,

'productName': 'Leaf Rake',

'productCode': 'GDN-0011',

'releaseDate': new Date('March 19, 2016'),

'description': 'Leaf rake with 48-inch wooden handle.',

'price': 19.95,

'starRating': 3.2,

'imageUrl': './assets/leafrake.jfif'

}];

}

}

## Class where we use service

import { ProductService } from '../services/product.service';

export class ProductsListComponent implements OnInit {

constructor(private \_productService: ProductService) { }

ngOnInit() {

this.products = this.\_productService.getProducts();

this.filteredProducts = this.products;

this.filterBy = '';

}

}

## Optional dependencies

In constructor we can have optional attribute using @Optional directives

import { [Optional](https://angular.io/api/core/Optional) } from '@angular/core';

@Injectable()

export class EmployeeService {

constructor(@[Optional](https://angular.io/api/core/Optional)() private logger: Logger) { if (this.logger) { this.logger.log(some\_message); } }

# Observable

* Observable is an asynchronous pattern. In the Observable pattern we have an Observable and an Observer. Observer observes the Observable
* An Observable can have many Observers (also called Subscribers).
* Observable emits items or notifications over time to which an Observer (also called Subscriber) can subscribe.
* When a subscriber subscribes to an Observable, the subscriber also specifies a callback function.
* This subscriber callback function is notified as and when the Observable emits items or notifications.
* Within this callback function we write code to handle data itmes or notifications received from the Observable.
* **Service class**

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

import { IEmployee } from '../models/IEmployee';

import { Http, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/catch';

import 'rxjs/add/Observable/throw';

import 'rxjs/add/operator/toPromise';

@Injectable()

export class EmployeeService {

private retryCount: number;

constructor(private \_http: Http) {

this.retryCount = 0;

}

getAllEmployees(): Observable<IEmployee[]> {

return this.\_http.get('http://localhost:55846/employee/getallemployees')

.map((response: Response) => <IEmployee[]>response.json()).catch(this.handleError)

.catch (this.handleError);

}

getEmployeeByCode(empcode: string): Observable<IEmployee>

{

return this.\_http.get("http://localhost:55846/employee/getemployeebycode/" + empcode)

.map((response: Response) => <IEmployee>response.json())

.catch(this.handleError);

}

getEmployeeByCodePromise(empcode: string): Promise<IEmployee> {

return this.\_http.get("http://localhost:55846/employee/getemployeebycode/" + empcode)

.map((response: Response) => <IEmployee>response.json())

.toPromise();

//.catch(this.handlePromiseError);

}

handleError(error: Response) {

console.error("Error is :" + error);

return Observable.throw(error);

}

handlePromiseError(error: Response) {

console.error(error);

throw (error);

}

# Filtering

## Filtering by custom pipes

**Code in Pipe**

import { Component, Pipe, Injectable, PipeTransform } from '@angular/core';

@Pipe({

name: 'filter'

})

@Injectable()

export class FilterPipe implements PipeTransform {

transform(items: any[], field: string, value: string): any[] {

if (!items) return [];

if (!value || value.length == 0) return items;

return items.filter(it =>

it[field].toLowerCase().indexOf(value.toLowerCase()) != -1);

}

}

**HTML file code**

<li \*ngFor="let it of its | filter : 'name' : 'value or variable'">**{{**it**}}**</li>

## Filtering in component method

export class NameParentComponent {

// Displays 'Mr. IQ', '<no name set>', 'Bombasto'

names = ['Mr. IQ', ' ', ' Bombasto '];

\_filterBy: string;

get filterBy(): string {

return this.\_filterBy;

}

set filterBy(value: string) {

this.\_filterBy = value;

this.filteredProducts = this.filterBy ? this.performFilter(this.\_filterBy) : this.products;

}

products: IProduct[] = null;

filteredProducts: IProduct[] = null;

constructor() { }

ngOnInit() {

this.products = [{}];

this.filteredProducts = this.products;

this.filterBy = 'cart';

}

performFilter(filterBy: string): IProduct[] {

return this.products.filter((product: IProduct) =>

product.productName.toLowerCase().indexOf(filterBy) !== -1);

}

}

# Routing

* **The following are the directives provided by the RouterModule**
* **routerLink**

Tells the router where to navigate when the user clicks the navigation link

* **routerLinkActive**

When a route is active the routerLinkActive directive adds the active CSS class. When a route becomes inactive, the routerLinkActive directive removes the active CSS class.

The routerLinkActive directive can be applied on the link element itself or it's parent. In this example, for the active route styling to work correctly, routerLinkActive directive must be applied on the <li> element and not the <a> element.

* **router-outlet**  
  Specifies the location at which the routed component view template should be displayed

## How routing works

<http://csharp-video-tutorials.blogspot.com/2017/11/how-routing-works-in-angular.html>

## Module.cs

/ Import NgModule decorator to decorate AppRoutingModule class

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

// Import RouterModule and Routes type from angular router library

import { RouterModule, Routes } from '@angular/router';

// Import the following 3 components as we will reference

// them in the route definitions below

import { HomeComponent } from './home/home.component';

import { EmployeesComponent } from './employees/employees.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

// Configure the routes. The Routes type and the

// referenced components are imported above

const appRoutes: Routes = [

  { path: 'home', component: HomeComponent },

  { path: 'employees', component: EmployeesComponent },

  { path: '', redirectTo: '/home', pathMatch: 'full' },

  { path: '\*\*', component: PageNotFoundComponent }

];

// The NgModule decorator is imported above

// Pass the configured routes to the forRoot() method

// to let the angular router know about our routes

// Export the imported RouterModule so it is available

// to the module that imports this AppRoutingModule

@NgModule({

  imports: [RouterModule.forRoot(appRoutes)],

  exports: [RouterModule],

})

export class AppRoutingModule { }

## Modify the code in the root AppModule in app.module.ts.

@NgModule({

  declarations: [

    AppComponent,

    HomeComponent,

    EmployeesComponent,

    PageNotFoundComponent

  ],

  // Import AppRoutingModule which contains our routing code

  // AppRoutingModule has also exported angular RouterModule, so

  // all the RouterModule features are also availble to this module

  // including the <router-outlet> directive used in the AppComponent

  // If AppRoutingModule module did not export RouterModule we get

  // 'router-outlet' is not a known element error

  imports: [BrowserModule, AppRoutingModule],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

## HTML

<div style="padding:5px">

<ul class="nav nav-tabs">

<li routerLinkActive="active">

<a routerLink="home">Home</a>

</li>

<li routerLinkActive="active">

<a routerLink="employees">Employees</a>

</li>

</ul>

<br />

<router-outlet></router-outlet>

</div>

## Routing through component

import { ActivatedRoute, Router } from '@angular/router';

@Component({

selector: 'employeedetails',

templateUrl: 'app/employee/EmployeeDetails.Component.html'

})

export class EmployeeDetailsComponent implements OnInit{

constructor(private \_router: Router) {

}

onBackButtonClick(): void {

this.\_router.navigate(['/employees']);

}

}

# Routing guard

* **CanActivate**

Checks to see if a user can visit a route or authorized to view page if not redirect to other page .

* **CanActivateChild**

Checks to see if a user can visit a routes children.

* **CanDeactivate**

Used to warn people if they are navigating away from a page where they have some unsaved changes

* **Resolve**

Performs route data retrieval before route activation.

* **CanLoad**

Checks to see if a user can route to a module that lazy loaded.

For a given route we can implement zero or any number of Guards.

It holds an array we could have multiple guards for a single route.

## Add guard to routing path

import { ProductDetailsGuard } from './products/product-details.guard';

const routes: Routes = [{ path: 'welcome', component: WelcomeComponent },

{ path: 'productdetails/:productId', **canActivate: [ProductDetailsGuard],** component: ProductDetailsComponent },

{ path: '', component: WelcomeComponent, pathMatch: 'full' },

{ path: '\*\*', component: PagenotfoundComponent, pathMatch: 'full' }];

## ProductDetailsGuard

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

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';

@Injectable({

providedIn: 'root'

})

export class ProductDetailsGuard implements CanActivate {

constructor(private \_router: Router) { }

canActivate(

next: ActivatedRouteSnapshot,

//state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {

state: RouterStateSnapshot): boolean {

let id = +next.url[0].path;

if (isNaN(id) || id < 1) {

alert("Ops! Invalid productId.");

this.\_router.navigate(['/products']);

return false;

}

return true;

}

## Candeactivete Guard

class UnsearchedTermGuard implements CanDeactivate<SearchComponent> {

canDeactivate(component: SearchComponent,

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean {

console.log("UnsearchedTermGuard");

console.log(route.params);

console.log(state.url);

return component.canDeactivate() || window.confirm("Are you sure?");

}

}

# Component

View+logic+Metadata

# Module

a module is a mechanism to group components, directives, pipes and services that are related, in such a way that can be combined with other modules to create an application

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule

An NgModule is defined by a class decorated with @NgModule(). The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

## App.Module.ts

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

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

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

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

import { PagenotfoundComponent } from './errorpages/pagenotfound.component';

@NgModule({

declarations: [

AppComponent,

PagenotfoundComponent],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

HttpClientModule],

exports: [],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

## Feature modules

## Shared modules

## **Declarations**

The components, directives, and pipes that belong to this NgModule.

Every component , directives, and pipes must belong to only one module

Not classes, services

Components, directives, pipes declared in one module are default private, accessible only in same module not in in other module.

## **Exports**

The subset of declarations that should be visible and usable in the component templates of other NgModules

Directive, pipe, component

Re-export even importing it to the module

Services/guards already shared through the application thus, no need to export it.

## Imports

Other modules whose exported classes are needed by component templates declared in this NgModule.

Only import required

Importing module does not provide access to imported module imports

## Providers

Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

## Bootstrap

The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

Alleast one component and only in root i.e. appmodule.

# What is bundling and why is it important for performance

angular application code is in many small files

Web browsers have a limit on how many scripts or CSS files they can download simultaneously.

Because of this browser limitation, your application may suffer from performance perspective, if it has many JavaScript and CSS files to download

Bundling can solve this problem by combining many small application and library files into a few bundles.

|  |  |
| --- | --- |
| **Bundle File** | **What it contains** |
| inline.bundle.js | WebPack runtime. Required for WebPack to do it's job |
| main.bundle.js | All our application code that we write |
| polyfills.bundle.js | Browser Polyfills |
| styles.bundle.js | Styles used by the application |
| vendor.bundle.js | Angular and 3rd party vendor files |

In addition to bundling, we can also use other optimisation techniques like Ahead-of-Time (AOT) Compilation, Minification, Uglification and TreeShaking to improve performance.

# What are Source Maps

To improve the performance, the application's JavaScript and CSS files are combined and compressed. It is extremely difficult to debug those compressed files. A source map holds information about the original files and can be used to map the code within a compressed file back to it’s original position in a source file. So with the help of these source maps we can easily debug our applications even after the the files are compressed and combined.

By default, a development build produce source maps where as a production build does not. However, we can change this default behaviour by using --sourcemaps option along with the ng build command. It's alias is -sm.

* Minification & Uglification :

The minified and uglified version of the file is smaller in size than the full version, resulting in faster response times and lower bandwidth costs.

A Prod Build is both minified and uglified, where as a Dev Build is not.

## What is Minification

The process of removing excess whitespace, comments, and optional tokens like curly brackets and semicolons is called Minification.

## What is Uglification

The process of transforming code to use short variable and function names is called uglification.

# What is Tree Shaking

Tree shaking is the process of removing any code that we are not actually using in our application from the final bundle. It's one of the most effective techniques to reduce the application size.

If you look at the bundles generated by the production build, they are significantly less in size compared with the bundles generated by the development build. This is beacause with the production build the code is tree shaked to remove dead code i.e the code that is not referenced by the application.

# Angular AOT vs JIT

In Angular we have 2 models of compilation

**JIT** - Just-in-Time Compilation : JIT compilation as the name implies, compiles the application Just-in-Time in the browser at runtime.

**AOT** - Ahead-of-Time Compilation : AOT compilation compiles the application at build time.

By default, with the development build we get JIT compilation. This is how it works. The application code along with the angular compiler is downloaded by the browser. At run-time, when a request is issued to the application, the JIT-compiler in the browser compiles the application code before it is executed. This means our user who made that first request has to wait for the application to compile first.

**With AOT compilation the angular application is pre-compiled.**

So this means the browser loads executable code so it can render the application immediately, without waiting to compile the application first.

This also mean with AOT, as the application is already pre-compiled, there is also no need for the browser to download the Angular compiler. As we already know, the compiler code is roughly half of the Angular framework, so omitting it dramatically reduces the application size.

**With AOT compilation**, template binding errors are detected and reported at build time itself as apposed to runtime

|  |  |  |
| --- | --- | --- |
| **Feature** | **Development Build** | **Production Build** |
| Source Maps | Yes | No |
| Extracts CSS | .js file | .css file |
| Minifaction | No | Yes |
| Uglification | No | Yes |
| Tree Shaking | No | Yes |
| AOT | No | Yes |

# Deploy Angular app on IIS

<http://csharp-video-tutorials.blogspot.com/2017/11/deploy-angular-app-to-iis.html>

# Angular CLI

Command Line Interface

<https://angular.io/cli>

<http://www.pragimtech.com/angular_cli_tutorial_for_beginners.aspx>

## Install cli

npm install -g @angular/cli

## Create Project

ng new ProductManagement

ng new ProductManagement –d

## Compile and Run app

ng serve –open

ng serve --aot

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Alias** | **Default** | **Purpose** |
| --watch | -w | true | Run build when files change |
| --live-reload | -lr | true | Whether to reload the page on change |
| --open | -o | false | Opens the url in default browser |
| --port | -p | 4200 | The port on which the server is listening |
| --extract-css | -ec |  | Extract css from global styles onto css files instead of js ones |

## Build/compile app

ng build

ng build –dev

ng build --dev -sm false

ng build --prod -sm true -ec true (ec extract CSS files from .js)

**ng serve**

Compiles and serves the application from memory

Does not write the build files to the disk

Typically used to run the application on local development machine

Cannot be used for deploying the build to another server (Ex. Testing, Staging or Production server)

**ng build**

Compiles the application to the "dist" folder

Can be used to produce both development & production builds

Typically used to deploy the application on another server

## Create Class

ng g class my-class-a

ng g cl employee

ng g cl employee/employee --spec=true

## Create Interface

ng g interface my-interface-a

ng g I interfacename

## Create Enum

ng g enum my-num

ng g e enumname

## Create Component

ng g component my-component-a

ng g c abc/componentname –flat (create component in given folder)

ng g c componentname –flat (create component without folder)

ng g c abc/componentname –d (dry run without creating files just shows a structure)

ng g c component –it –is –spec=false (it inline template, is inline style, –spec unittestcase file)

## Create Service

ng g service my-service-a

ng g s employee -m=app.module -d (Create service in given module)

ng g s student -d --spec=false --flat=false

## Create Module

ng g m moduleName

ng g m signup/signup -d -m=app.module --spec=true --flat=true --routing

## Create directive

ng g d directiveName

## Create Pipes

ng generate pipe pipeName

ng g p pipeName

## Create routing guard

ng generate guard guardName

ng g g guardName

## Linting Angular project

Similar to running code analysis

ng lint

install Visual Studio Code extension - TSLint.