Angular

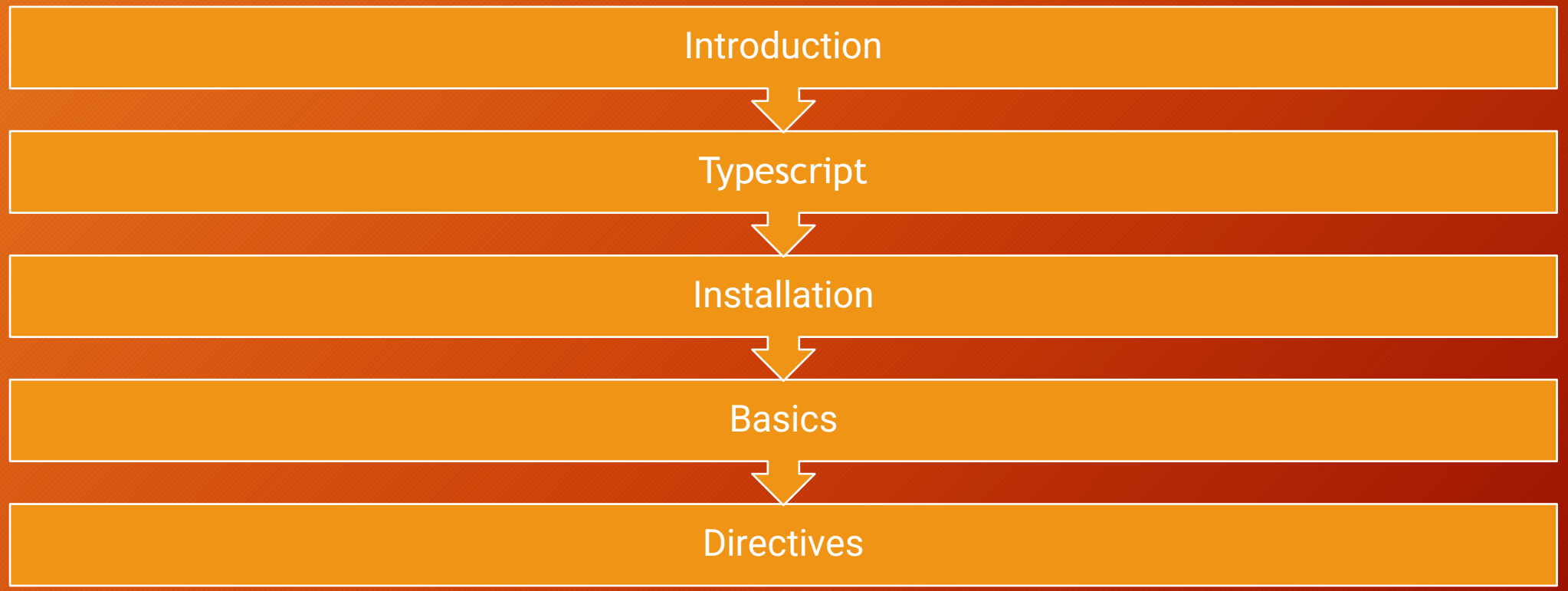
Instructor : Youssef RAFII

October 2022

Training plan

First Day	Second Day	Third Day	Fourth Day
Introduction	Databinding	Routing	Forms: Reactive
Typescript	Pipes	Observables	Making Http Requests
Setup & Installation	Services & Dependency Injection	Forms: Template driven	**Authentication**
Basics			
Directives			

First Day


Introduction
Typescript
Installation
Basics
Directives

GENERALE PERFORMANCE

Introduction

Introduction to Angular

Angular is a platform and framework for building single-page client applications using HTML and TypeScript.

Angular is a development platform, built on TypeScript.

Angular is Open Source and primarily maintained by Google.

Single Page Application(SPA)

A single page web application, is a web application that fits itself on to a single web page.

In older application concept(for each view a web page is created), several problems were detected:

- rewrite of Html
- each time we request a new page the server compile and send the page to the browser

The browser downloads the entire app data when you visit SPA web applications.

So, you can browse through different parts of the app seamlessly and the page won't reload every time you click on something.

This is because a single page application executes the logic in the browser, instead of a server.

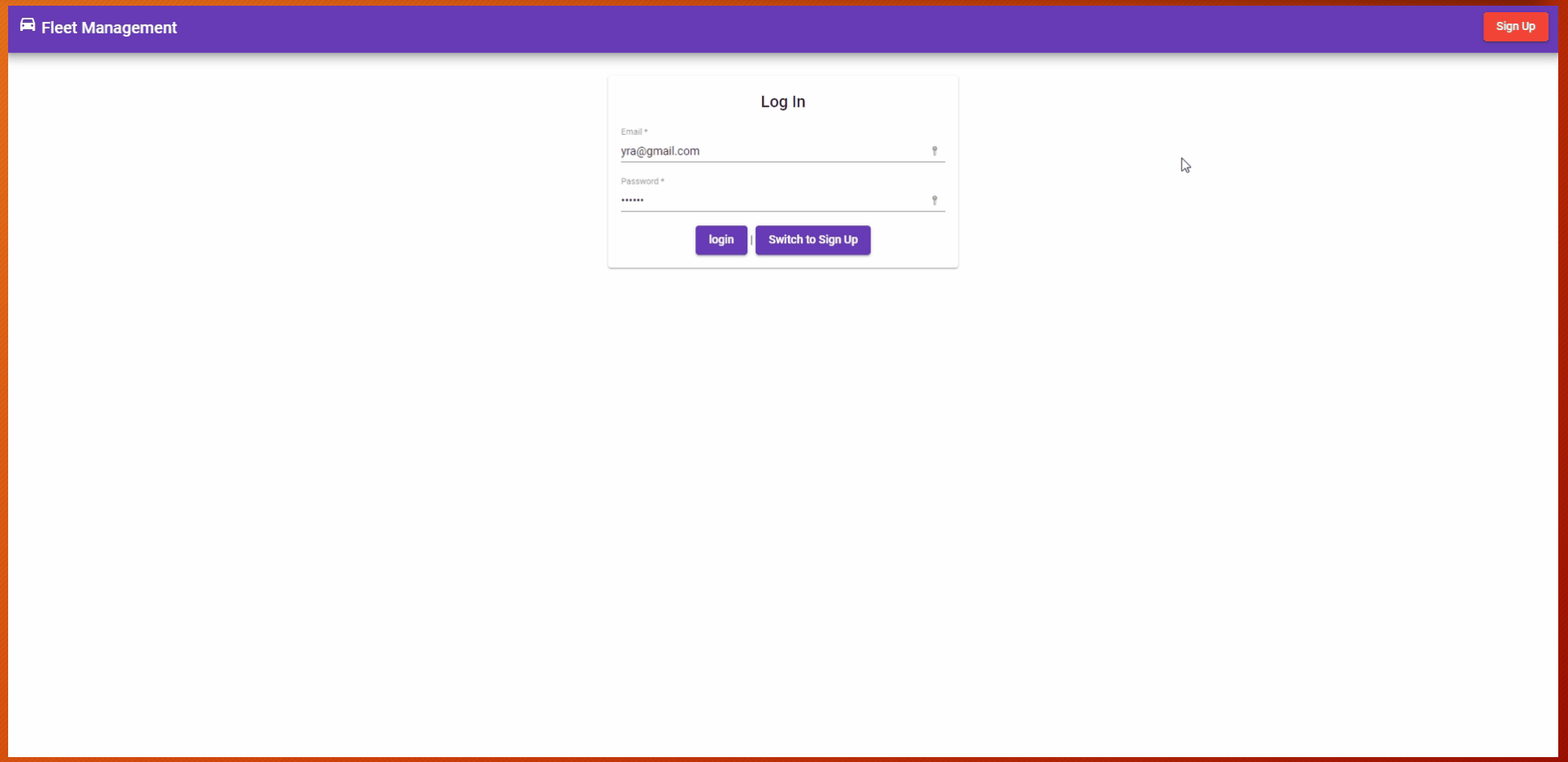
As a result, SPAs are known to deliver fast and efficient performance.

Support policy

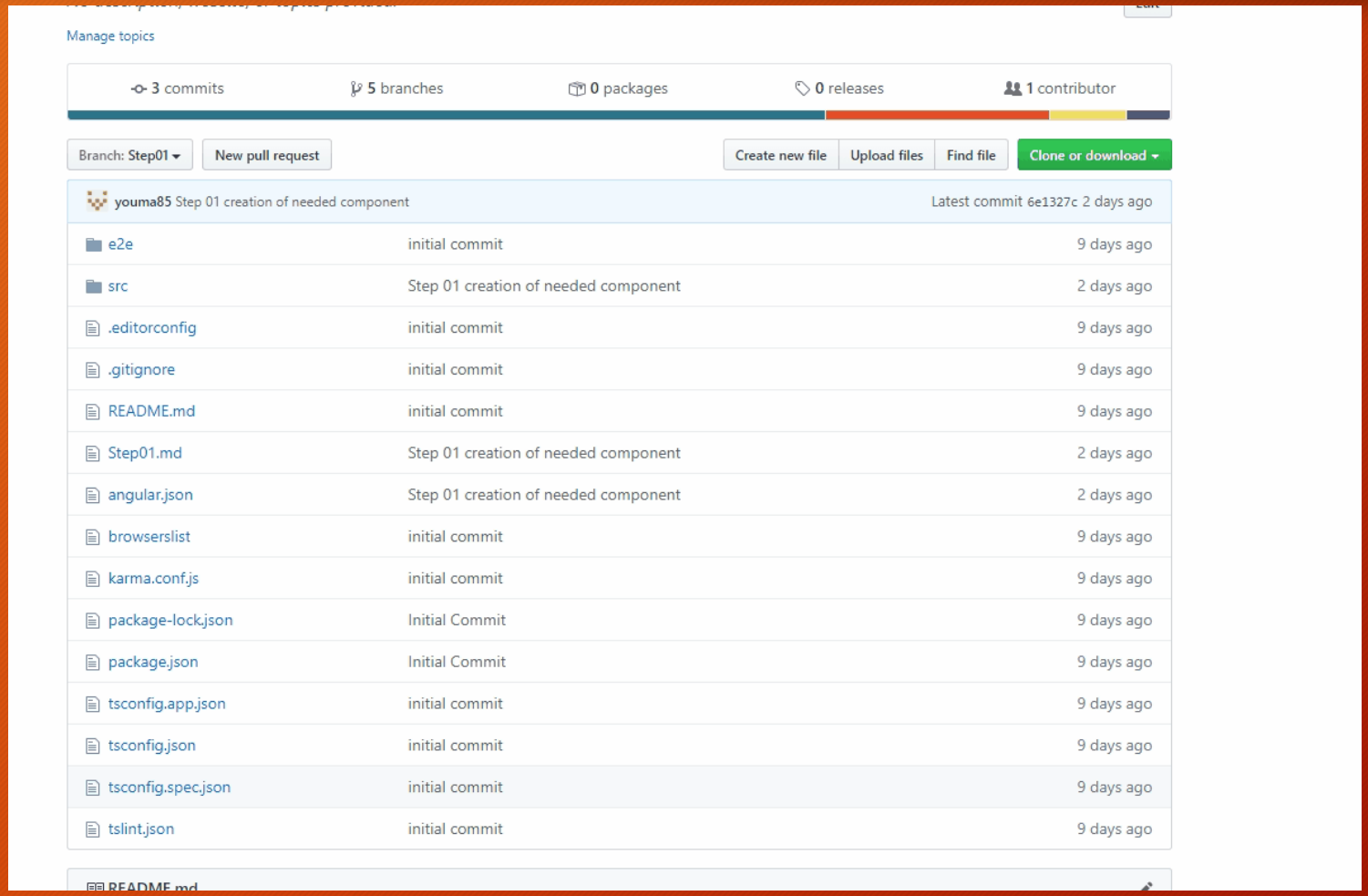
All major releases are typically supported for 18 months.

Angular versions v2 to v11 are no longer under support.

What will be done during this training



Code in a git repository by Steps



Manage topics

3 commits | 5 branches | 0 packages | 0 releases | 1 contributor

Branch: Step01 | New pull request | Create new file | Upload files | Find file | Clone or download

youma85 Step 01 creation of needed component — Latest commit 6e1327c 2 days ago

e2e	initial commit	9 days ago
src	Step 01 creation of needed component	2 days ago
.editorconfig	initial commit	9 days ago
.gitignore	initial commit	9 days ago
README.md	initial commit	9 days ago
Step01.md	Step 01 creation of needed component	2 days ago
angular.json	Step 01 creation of needed component	2 days ago
browserslist	initial commit	9 days ago
karma.conf.js	initial commit	9 days ago
package-lock.json	Initial Commit	9 days ago
package.json	Initial Commit	9 days ago
tsconfig.app.json	initial commit	9 days ago
tsconfig.json	initial commit	9 days ago
tsconfig.spec.json	initial commit	9 days ago
tslint.json	initial commit	9 days ago

README.md

GENERALE PERFORMANCE

TypeScript

Introduction to Typescript

- A strongly typed programming language.
- Created by Microsoft on 2012.
- TS is used to provide type safety to a program:
  - Keeps the app free from type errors as the case for JS.
  - Keeps the JS code free from undefined and null values.
- TS is supported by all major libraries and frameworks.
- TS is used only for development purpose, it’s output is a JS code that will be used on browsers.

Data Types

- **string**: let name: string;
- **number**: let age: number;
- **boolean**: let isValid: boolean;
- **array**
  - let driverList: string[];
  - let driverList= string['ahmed', 'Fatima'];
- **enum**
  - enum Color { Red, Green, Black}
  - let c: Color = Color.Red;
- **tuple**: employee: [number, string] = [1, "Ahmed"];
- **any**
- **void**: let speech: void = sayHi();

Functions

- Different ways to write a function:
  - function add(x:number,y:number):number {
    return x+y;
    }
  - const add=(x:number, y:number):number => x+y;
  - const add= function (x:number, y:number) { return x+y };

- Optional params:
  - function add(x:number,y:number, **z?**:number):number {
    return z? x+y+z: x+y;
    }

- Default param:
  - const add=(x:number, y:number, **z=10**): number => x+y+z;
- Rest parameters:
  - const add=(x:number, y:number, **…z**:number[]): number => x+y+z.reduce((a,b)=> a+b, 0
    );

ES6+: Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the curly brackets.

They must be defined **before** they are used.

```
// ES5
var x = function(x, y) {
   return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Classes

```
class Driver {
	constructor(public id: number, public name: string, public age: number);
}
```

```
class Driver {
	#id: number;
	name: string;
	age: number;
	constructor(id: number, name: string, age: number) {
		this.#id = id;
		this.name = name;
		this. age = age;
	}
	get driverId() :number { return this.#id;}
	set driverId(id:number) { this.#id=id;}
}
```

Decorators

A Decorator is a special kind of declaration that can be attached to a class declaration, method, property or parameter to modify their behaviour.

Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

```
@Component({
        …
})
export class HomeComponent {
        …
}
```

var vs let

var	let
The var keyword was introduced with JavaScript.	The let keyword was added in ES6 (ES 2015) version of JavaScript.
It has global scope.	It is limited to block scope.

Variable declared with var keyword can be re-declared and updated in the same scope.
function varGreeter(){
var a = 10;
var a = 20; //a is replaced
console.log(a);
}
	Variable declared with let keyword can be updated but not re-declared.
	function varGreeter(){
	let a = 10;
	let a = 20; //SyntaxError: //Identifier 'a' has already been declared
	console.log(a);
	}

It is hoisted(declaration moves to top).
{
console.log(c); // undefined. //Due to hoisting
var c = 2;
}
	It is not hoisted.
	{
	console.log(b); // ReferenceError: //b is not defined
	let b = 3;
	}

GENERALE PERFORMANCE

Setup & Installation

Setup Environment

	**Node JS**	**https://nodejs.org/en/download/releases/**
	VsCode	https://code.visualstudio.com/
	Angular Cli	https://cli.angular.io/

Node Js

Download the current version from here: https://nodejs.org/en/download/current/

To check your version, run:

node -v in a terminal/console window.

Installing Angular CLI

Installation:

```
npm install -g @angular/cli
```

If you have an older version, Update to the latest version:

```
npm uninstall -g @angular/cli
npm cache clean -force
npm install -g @angular/cli
```

Verify the installation:

```
ng v
```

https://cli.angular.io/

CLI Overview

ng add: Adds support for an external library to your project.

ng build(ng b): Compiles an Angular app into an output directory named dist/ at the given output path.

ng deploy: Invokes the deploy builder for a specified project or for the default project in the workspace.

ng doc (ng d): Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.

ng generate(ng g): Generates files based on a schematic.

For more informations: https://angular.io/cli

Create new Angular App

- Run the CLI command ng new and provide the name as shown here:

```
ng new my-app
```

The ng new command prompts you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key.

- Go to the workspace folder (my-app).
- Launch the server by using the CLI command ng serve, with the --open option:

```
cd my-app
ng serve
```

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

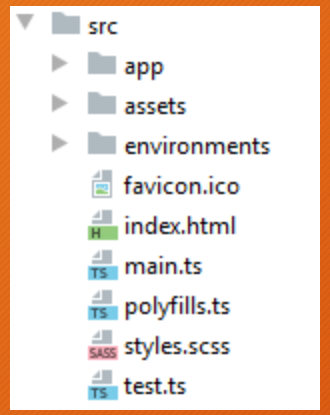
GENERALE PERFORMANCE

Basics

Workspace walkthrough

Asset	Role
node_modules	Contains all external modules and libraries used in the application
src/	The main workarea/app code resides inside this folder
angular.json	Configuration file that contains information related to the workspace
karma.conf.js	Configuration file of karma runner of unit test
package.json	Manifest of the project that includes the dependencies, information about its unique source control, project's name, description, and author, etc
package-lock.json	describe the exact dependency tree currently installed
tsconfig.app.json	Configuration file that extends the tsconfig.json, and used to compile the application codes
tsconfig.json	Contains the TS configuration
tsconfig.spec.json	Configuration file that extends the tsconfig.json, and used to compile the test codes

The contents src folder



Asset	Role
app	Contains the primary App components, modules, directives,…
assets	Store static assets like images, styles, …
environments	Environment configurations to allow you to build for different targets, like dev, qualification or production
favicon.ico	Image displayed as browser favorite icon
index.html	Root HTML file for the application
main.ts	Booting the web application
polyfills.ts	Few lines of code which make your application compatible for different browsers
styles.css	Contains all global stylesheet
test.ts	Unit test entry point, not part of application

angular.json: important properties

- sourceRoot: the source folderof the project.
- outputPath: the final compiled code will be generated in this folder.
- index: location of the root page.
- main: location of the main typescript file.
- tsConfig: location of the ts configuration file.
- inlineStyleLanguage: type of stylesheet choosen.
- assets: location of assets.
- styles: location of global styles and other styles libraries.

Bootstrapping the app

main.ts

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

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

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

If production is enabled, turn off Angular developer mode.

Bootstraps the App module

index.html

```
<body>
  <app-root></app-root>
</body>
```

Angular will look for the app-root element and replace it with the rendered component.

App module

The App module is the packaging that tells Angular what's available to render

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

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

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

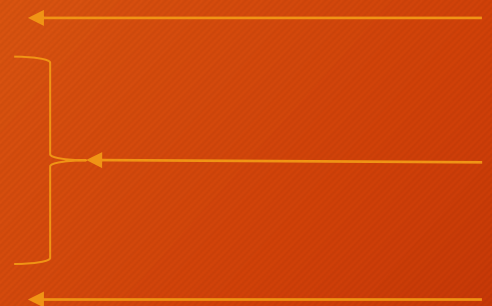
- Imports Angular dependencies needed
- Imports the App component
- A decorator to define a module by passing an object
- Declarations are to list any components and directives used in the app.
- Imports are other modules that are used in the app.
- provides services that are essential to launch and run a browser application
- Providers are any services used in the app.
- Bootstrap declares which component to use as the first to bootstrap the application.

The "root module" is a classic module whose particularity is to define the "root component" of the application via the bootstrap property.

App component

- App component is the root of the application

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

}
```



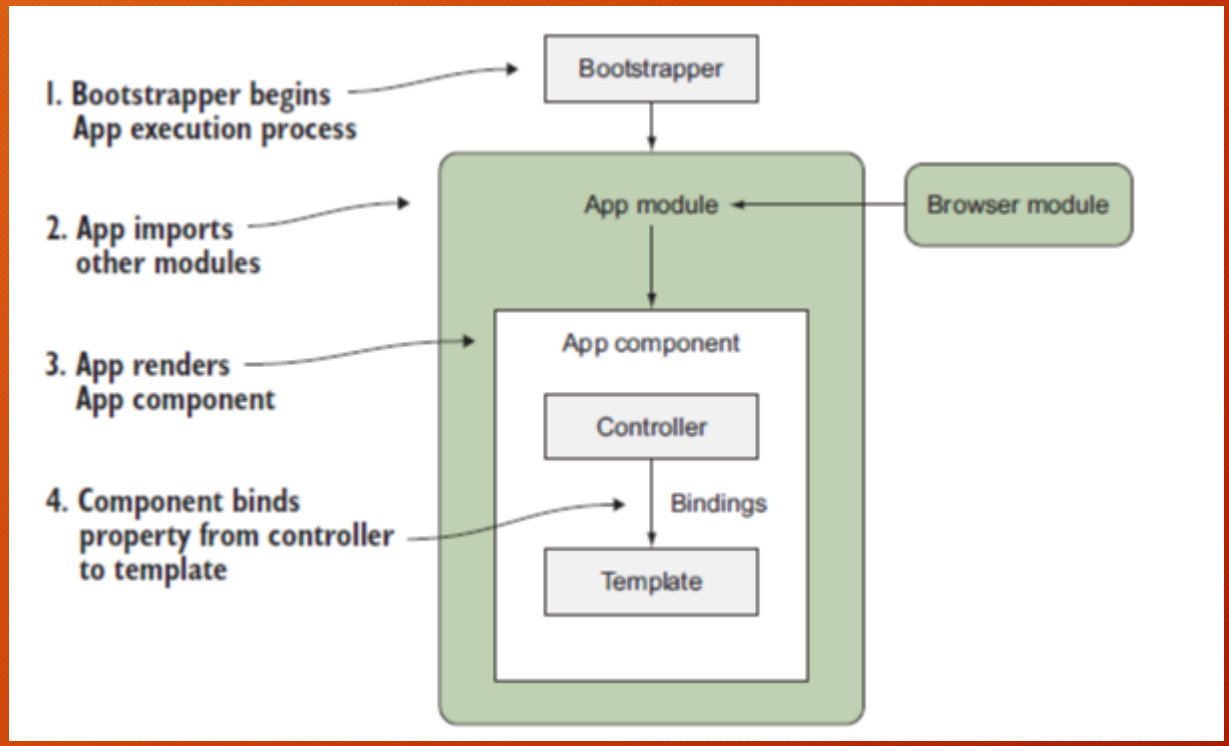
Import the component annotation

Define the component
and its properties

Create the component controller,

This component is used in the src/index.html file which is the HTML page hosting the Angular application.

Bootstrapping the app



Basics: Modules (definition)

Angular offers a concept of modules to better structure the code and facilitate reuse and sharing.

An Angular module is a mechanism allowing to:

- group together components (but also services, directives, pipes, etc.),
- define their dependencies,
- and define their visibility.

An Angular module is defined simply with a class (usually empty) and the NgModule decorator.

Basics: Modules(definition)

declarations:

Defines the list of components (or directives, pipes etc ...) contained in this module.

exports:

Defines the list of components that can be used by modules that import this one.

imports:

Defines the list of module dependencies. This is usually the list of modules containing the components used by the components in the declarations section

Basics: Modules(Types)

Root Module

- The App Module can also be called the root module. Every app must contain at least one module and that is the App Module. We launch our applications by bootstrapping the root module.

Core Module

- The Core Module is where we want to put our shared singleton services.

Feature Module

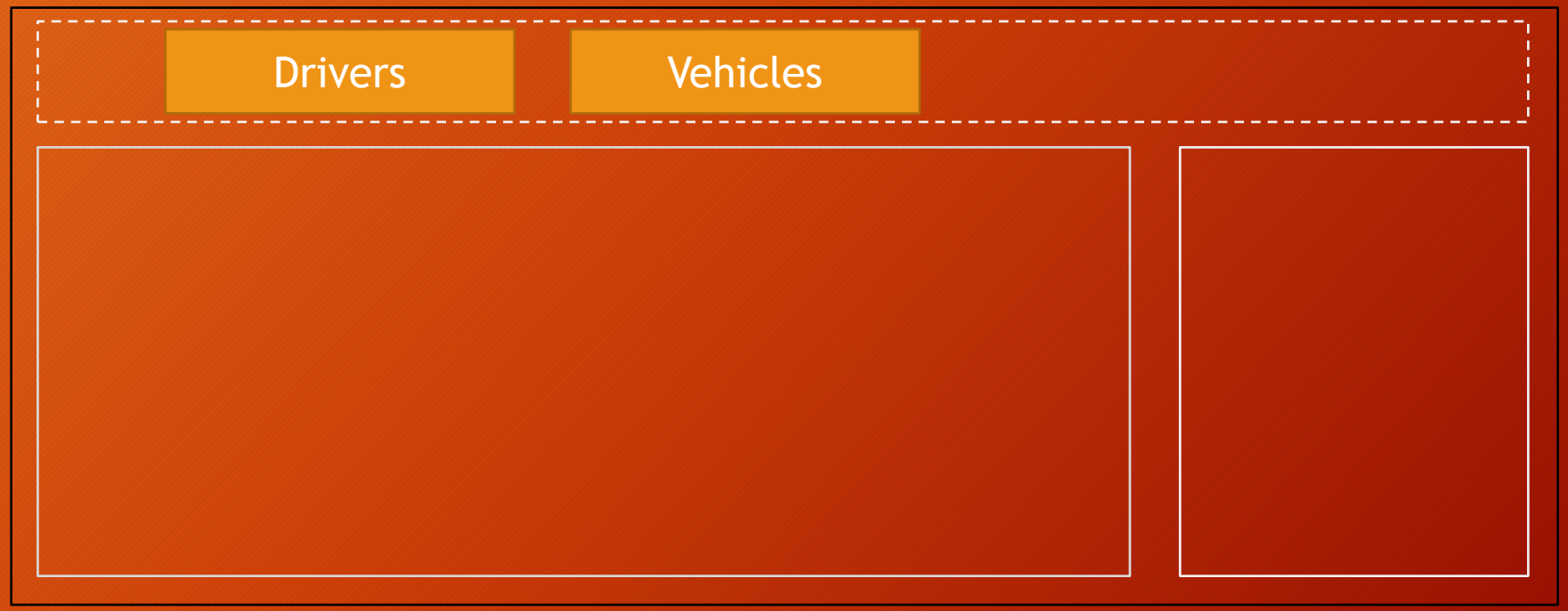
- The components (as well as the directives, pipes, etc.) are grouped together in modules by functionality. These modules are then called "feature module".

Shared Module

- The Shared Module is where we want to store common modules, components, etc. Everything is shared throughout the application.

Basics: Components

- Components are the most basic UI building block of an Angular app. An Angular app contains a tree of Angular components.



Basics: @Component

A component is a class decorated with @Component,

This decorator takes as parameter a configuration which contains at least:

- selector: the CSS selector which will link the HTML tag of the element and the code of the component.
- template: the HTML template used by Angular to generate the element's content in the DOM.

Basics: Components creation

We can create component in 2 ways:

- Using angular cli:

ng generate component drivers **or shortly (*ng g c drivers*)**

- Manually by creating the class and declare it in the app module file:

```
@Component({
  selector: 'app-driver',
  templateUrl: './driver.component.html',
  styleUrls: ['./driver.component.scss']
})
export class DriverComponent {

}
```

```
@NgModule({
  declarations: [
    AppComponent,
    DriverComponent,
  ]
```

Basics: Component selector type

- app-root: used as <app-root>
- [app-root]: used as <div app-root>
- .app-root: used as

GENERALE PERFORMANCE

Data Binding

Basics: Databinding

Databinding = communication between ts code and what you see (html)

String Interpolation

Allow us to display information from typescript in the html code like variables or functions. E.g.: `{{driver.firstName}}`

Property binding

[property]: indicate to angular that we want to dynamically bind some property of the tag. E.g.:

```
<button [disabled]="false"></button>

<button [disabled]="driver.noExperiences>5"></button>

<button [disabled]="isValid()"></button>
```

Basics: Databinding

Event binding

Event binding allows you to listen for certain events such as keystrokes, mouse movements, clicks, and touches.

```
<button (click)="onNewDriver()">New Driver</button>
```

```
onNewDriver(){
  ...
}
```

Two-Way-databinding

For Two-Way-Binding to work, you need to enable the ngModel directive by adding the **FormsModule** to the imports[] array in the AppModule.

With two way databinding we combine property and event binding.

```
<input  name="lastName" [(ngModel)]="driver.lastName" >
```

GENERALE PERFORMANCE

Directives

Directives: Definition

Directives are classes used to change the behavior and appearance of DOM element.

There are two kinds of directives in Angular:

1.Structural directives: change the DOM layout by adding and removing DOM elements, for example *ngFor and *ngIf.

2.Attribute directives: change the appearance or behavior of an element, component, or another directive, for example ngStyle.

Angular gives the possibility to create custom directives.

Directives: NgIf

```
<p *ngIf="boolean"></p>
```

With else

```
<p *ngIf="form.isValid; else elseBlock ">
 Form is Valid
</p>

<ng-template #elseBlock>
 Form is not Valid
</ng-template>
```

Directives: NgStyle

with a Object:

```
<button [ngStyle]="{background: 'red'}">Click Me!</button>
```

with a function:

```
<p [ngStyle]="{backgroundColor:getColor()}"></p>
```

```
getColor(){
  return this.status==='done'? 'green': 'red';
}
```

Directives: NgClass

with a class name:

```
<p [ngClass]="'online'">paragraph 1</p>
```

with a multiple class name:

```
<p [ngClass]="'online active'">paragraph 1</p>
```

Conditionnal application of classes

```
<p [ngClass]="{ online: false, active: true }">paragraph 1</p>
```

With databinding

```
<p [ngClass]="{ online: isValid }">paragraph 1</p>
```

```
.online {
  color: blue;
}

.active {
  background-color: yellow;
}
```

Directives : NgFor

```
<div  *ngFor="let drv of drivers">
   {{drv.firstName}}
</div>
```

Get the index

```
*ngFor="let drv of drivers; let i= index"
```

Directives: **NgSwitch**

```
<div [ngSwitch]="value">
  <p *ngSwitchCase="5">Value is 5</p>
  <p *ngSwitchCase="10">Value is 10</p>
  <p *ngSwitchCase="100">Value is 100</p>
  <p *ngSwitchDefault>Value is default</p>
</div>
```

Directives: Custom directives

Creation with CLI: ng g d directive_name

```
@Directive({
  selector: '[appShadow]'
})
export class ShadowDirective{

  constructor(elem: ElementRef, renderer: Renderer2) {
    renderer.setStyle(elem.nativeElement, 'background-color', 'yellow');
  }

}
```

Must be declared in the app.module

```
@NgModule({
  declarations: [
    AppComponent,
    DriverComponent,
    ShadowDirective
  ],
```

```
<div appShadow>
  ...
</div>
```

ElementRef : The "Dependency Injection" allows to retrieve via the ElementRef class, a reference to the object allowing to handle the associated DOM element

Directives Listener

```
@HostListener('mouseenter')
mouseOver(eventData:Event){
 this.elementRef.nativeElement.style.backgroundColor='yellow';
}

@HostListener('mouseleave')
mouseLeave(eventData:Event){
 this.elementRef.nativeElement.style.backgroundColor='transparent';
}
```

@HostListener () is a decorator allowing to add a "listener" on the element on which the directive is applied ("host element").

Using HostBinding to Bind to Host Properties

Bind style.backgroundColor to the property backgroudColor:

```
@HostBinding('style.backgroundColor') backgroundColor:string='transparent';

@HostListener('mouseenter') mouseOver(eventData:Event){
  this.backgroundColor='yellow';
}

@HostListener('mouseleave') mouseLeave(eventData:Event){
  this.backgroundColor='transparent';
}
```

Second Day

Components & Databinding

Pipes

Services & Dependency Injection

GENERALE PERFORMANCE

Components & Databinding

Input: Property Binding

To transmit data to a "child component", we will communicate with it in the same way as we control the properties of a native element, that is to say using the Property Binding:

```
<app-driver-item
  *ngFor="let drv of drivers"
  [driver]="drv">
</app-driver-item>
```

This block of code will call an implicit "set" of the «driver» property of the instance of the child component.

Input: @Input Decorator

By default, no component property can be changed by Property Binding.

It is therefore necessary to define the properties that can serve as "input" to the component by simply adding the decorator @Input().

```
export class DriverItemComponent implements OnInit {

  @Input() driver: Driver;
}
```

Input: Binding To custom property

```
@Component({
 selector: 'app-driver-list',
 template: `
<app-driver-item
   *ngFor="let drv of drivers"
   [driver]="drv">
</app-driver-item>`
})
export class DriverListComponent implements OnInit {
 drivers: Driver[] =  [
   ...
];
}
```

```
@Component({
 selector: 'app-driver-item',
 template: `
 <h3>Name: {{driver.name}}</h3>
 ` })
export class DriverItemComponent {
 @Input() driver: Driver;
}
```

Output: Event Binding

In the same way that Inputs allow you to communicate data to a "child component", Output can transmit data to the "parent component" via an "Output" mechanism similar to the Event Binding used previously to capture natives events.

```
<app-driver-item
  *ngFor="let drv of drivers"
  (driverSelected)="onSelectingDriver($event)">

</app-driver-item>
```

The expression onSelectingDriver(drv) allows to register a listener for the «driverSelected» event.

Note the similarity with the Event Binding on click event(or Other DOM events).

```
<button (click)="onSave()">Save</button>
```

Output: Declaration of property and decorator @Output ()

By simply declaring the **driverSelected** property on the app-driver-item component:

```
export class DriverItemComponent{
    driverSelected= new EventEmitter<Driver>();
}
```

You must add the decorator @Output():

```
export class DriverItemComponent{
  @Input() driver: Driver;
  @Output() driverSelected= new EventEmitter<Driver>();
}
```

@Output allows you to make the eventEmitter listenable from the outside

Output: Sending values

As indicated by his name, an EventEmitter allows you to emit values. It can therefore be used anywhere in the DriverItemComponent class to pass values to the parent component via the emit method.

```
<button mat-raised-button (click)="onClick()">Show</button>
```

```
onClick() {
  this.driverSelected.emit(this.driver);
}
```

Output: Binding To custom event

```
@Component({
  selector: 'app-driver-list',
  template: `
    <app-driver-item
    *ngFor="let drv of drivers"
    [driver]="drv"
    (driverSelected)="onDriverSelected($event)"></app-driver-item>
  `
})
export class DriverListComponent implements OnInit {
  drivers: Driver[] =  [
  ];

  onDriverSelected(driver: Driver) {
    console.log(driver);
  }
}
```

```
@Component({
  selector: 'app-driver-item',
  template: `
    {{driver.firstName}} {{driver.lastName}}
    <button mat-raised-button (click)="onClick()">Show</button>
  `
})
export class DriverItemComponent{
  @Input() driver: Driver;
  @Output() driverSelected = new EventEmitter<Driver>();

  onClick() {
    this.driverSelected.emit(this.driver);
  }
}
```

Local reference

```
<input
    type="text"
    class="form-control"
    #driverName>

<button
    class="btn btn-primary"
      (click)="onAddDriver(driverName)">Add Driver</button>
```

```
onAddDriver(name:HtmlInputElement){
  console.log(name.value);
}
```

Or, we can manipulate the local reference from typeScript via ViewChild:

```
@ViewChild('driverName') name: ElementRef;
```

GENERALE PERFORMANCE

Pipes

Pipe

Pipes are filters that can be used directly from the view in order to transform the values during the binding.

The Pipes syntax is simply inspired by UNIX shell Pipes found in many templating systems.

Pipes are added into template expressions using the pipe character (|).

```
<div>{{ user.firstName | lowercase }}</div>
```

```
{{ server.started | date}}
```

Angular has several native "pipes": https://angular.io/api?type=pipe

Built-in Pipes

- DatePipe
- UpperCasePipe
- LowerCasePipe
- CurrencyPipe
- DecimalPipe
- PercentPipe
- JsonPipe
- SlicePipe

Pipe with parameters

Pipes can take parameters that must be puted after the Pipe and separated with the symbol ":".

```
<div>{{ user.firstName | slice:0:10 }}</div>
```

```
{{ driver.birthDate | date:'short'}}
```

Chaining

The "pipes" can be chained.

```
<div>{{ driver.firstName | slice:0:10 | lowercase }}</div>
```

```
{{ server.started | date:'fullDate'| uppercase}}
```

Creating a Custom Pipe

To create a custom Pipe, you need:

1. Create a class that implements the PipeTransform interface,
2. decorate this class with the decorator @Pipe () by indicating the name of the Pipe.
3. add the class to the declarations (and exports) of the associated module.

Creation with angular CLI: ng g p namePipe

Creating a Custom Pipe

```
import { PipeTransform } from "@angular/core";

@Pipe({
  name:'shorten'
})
export class ShortenPipe implements PipeTransform{
  transform(value:any){
    if(value.length>10){
      return value.substr(0,10)+'...';
    }
    return value;
  }
}
```

```
{{ driver.firstName | shorten }}
```

parametrizing a custom pipe

```
import {PipeTransform} from "@angular/core";

@Pipe({
  name: 'shorten'
})
export class ShortenPipe implements PipeTransform {
  transform(value: any, limit: number) {
    if (value.length > limit) {
      return value.substr(0, limit) + '...';
    }
    return value;
  }
}
```

```
{{ driver.firstName | shorten:5  }}
```

```
{{ driver.firstName | shorten:5:arg2:arg3  }}
```

GENERALE PERFORMANCE

Services & Dependency Injection

What is "Dependency Injection"?

Dependencies are services or objects needed by classes to perfom some function.

The "Dependency Injection" is a "design pattern" which consists in separating the instantiation of a dependency and its use.

It decreases coupling between a class and its dependency.
Extending the application becomes easier.

Injection of an Angular Service

With Angular, a dependency is generally the instance of a class allowing to factorize certain functionalities or to access a state thus allowing the components to communicate with each other.

In the Angular vocabulary, these classes are called "services".

An Angular service can be injected by any Angular class (i.e.: component, Directive, Service or Pipe) via the parameters of its constructor.

```
export class DriverListComponent {

  constructor(private driverService: DriverService) {
  }
}
```

Declaration of a Service

To declare an Angular service, just create a TypeScript class and decorate it with the decorator @Injectable ().

```
@Injectable()
export class StudentService {
}
```

@Injectable(): Decorator that marks a class as available to be provided and injected as a dependency.

```
@NgModule({
  providers: [StudentService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

In order to instantiate a service, Angular needs a "provider" telling it how to produce the instance of this service.

This is generally done via the providers property in the app module.

Create Service : ng g s ServiceName

Services in Angular 6+

Since Angular 6, it is no longer necessary to declare the service in the "providers" at the module level.

Instead of adding a service class to the `providers[]` array in `AppModule`, you can set the following config in `@Injectable()`:

```
@Injectable({
  providedIn: 'root'
})
export class StudentService {
}
```

Scope of Services

AppModule

Same instance of Service is available **Application-wide**

AppComponent

Same instance of Service is available for all **Components**(but Not for other services)

Any Other Component

Same instance of Service is available for **the Component and all its child components**

Third Day

Routing

Observables

Forms: Template driven

GENERALE PERFORMANCE

Routing

Routing definition

Routing is the mechanism that allows you to navigate from one page to another on a website.

The Angular router is a core part of the Angular platform.

It enables developers to build Single Page Applications with multiple views and allow navigation between these views.

Setting up of "Routing"

The "Routing" configuration is transmitted to the RouterModule module when it is imported by the AppModule "root module".

It is recommended to place this configuration in a dedicated module AppRoutingModule imported by the "root module" AppModule.

```
const appRoutes: Routes = [
  {path: '', redirectTo: '/drivers', pathMatch: 'full'},
  {path: 'drivers', component: DriverComponent},
  {path: 'vehicles', component: VehicleComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(appRoutes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```

ng generate module app-routing --flat

Setting up routes with parameters

The path of a "route" can define mandatory parameters with the prefix «:»

```
{path: ‘driver/:id', component: DriverDetailsComponent}
```

<router-outlet>

The configuration of "Routing" allows you to define which component to display according to the route but this does not tell Angular where to inject the component in the page.

To indicate the the loading location of the routes, use the <router-outlet> directive directly in the "root component" AppComponent (or another one).

```
<router-outlet></router-outlet>
```

Navigating with Router Links

Using native links <a href="/drivers">, the "browser" will generate an HTTP GET request to the server and reload the entire application.

To avoid this problem, the Angular Routing module provides the routerLink directive which allows to intercept the click event on the links and to change the "route" without reloading the whole application.

```
<a routerLink="/drivers">Drivers</a>
```

Navigating Programmatically

The navigatation can be done programmatically with the Router object

```
import {Router} from "@angular/router";
...
constructor(private router: Router) { }

...

this.router.navigate(['/drivers']);
```

Styling Active Router Links

```
<a mat-button routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact:true}">Home</a>
<a mat-button routerLink="/drivers" routerLinkActive="active">Drivers</a>
<a mat-button routerLink="/vehicles" routerLinkActive="active">Vehicles</a>
```

[routerLinkActiveOptions]="{exact:true}"

Do not activate the routerLinkActive only if the url is exactly ‘/’

Child(Nested) Route

A child route is like any other route, in that it needs both a path and a component.

The one difference is that you place child routes in a children array within the parent route

```
{path: ‘drivers', component: DriverComponent, children: [
   {path: '', component: DriverListComponent},
   {path: 'new', component: CreateDriverComponent},
   {path: ':id', component: UpdateDriverComponent}
 ]},
```

Using childe route, we can load other routes inside an existing route.

Must add <router-outlet></router-outlet> in the Html of the Driver component

Passing Parameters to Routes

Configuration:

```
...
{path: ':id', component: DriverDetailsComponent}
...
```

Calling the route:

```
<a [routerLink]="['/drivers',5]" ></a>
```

Or programmatically

```
this.router.navigate(['/ drivers', idDriver]);
```

Fetching Route Parameters

```
import {ActivatedRoute} from "@angular/router";

constructor(private route: ActivatedRoute) {
  this.id = this.route.snapshot.params['id'];
}
```

Fetching Route Parameters Reactively

```
constructor(private route: ActivatedRoute) {

 this.id = this.route.snapshot.params['id'];

  this.route.params
    .subscribe(
      (parametres:Params)=>{
    this.id = parametres['id'];
  });
}
```

Redirecting and Wildcard Routes

```
{path:'not-found',component:PageNotFoundComponent},
{path:'**', redirectTo:'/not-found’}
```

The path:'**' must be the last route in the appRoutes

Guards

"Guards" are used to control access to a "route" (e.g. authorization) or departure from a "route".

Configuration:

The "Guards" are added in "Routing" configuration:

```
{path: ‘drivers', component: DriverComponent,canActivate:[IsSignedInGuard]}
{path: ‘vehicles', component: VehicleComponent,canDeactivate:[IsNotDirtyGuard]}
```

CanActivate

The canActivate is called when the URL changes to the route and matches the route with the Guard.
This type of guards is commonly used to:

- Limit route access to specific users
- Ensure prerequisites are met

An activation guard is a service that implements the CanActivate interface.

```
@Injectable({
  providedIn: 'root'
})
export class IsSignedInGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.authService.isAuthenticated()? true: this.router.navigate(['/login']);
  }
}
```

CanActivateChild

This guard type is similar to the canActivate, except that it is called when activating a route child and not a route itselft.

An activation Child guard is a service that implements the CanActivateChilde interface.

```
@Injectable({
  providedIn: 'root'
})
export class AdminGuard implements CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.authService.isAdmin()? true: this.router.navigate(['/home']);
  }
}
```

```
{path: 'drivers',
  component: DriverComponent,
  canActivateChild: [AdminGuard],
  children: [
  ...
  ]},
```

CanDeactivate

A deactivation guard is a service that implements the CanDeactivate interface.
This service must implement the canDeactivate method.
This method is called whenever the user wishes to leave the route. it must then return a value of type boolean or Promise <boolean> or Observable <boolean> indicating whether access to the "route" is authorized or not.

```
@Injectable({
 providedIn: 'root'
})
export class IsNotDirtyGuard implements CanDeactivate<DriverComponent> {
 canDeactivate(component: ProfileViewComponent,
               currentRoute: ActivatedRouteSnapshot,
               currentState: RouterStateSnapshot,
               nextState?: RouterStateSnapshot){
          if(component.driverForm.dirty){
           return true;
          } else {
           alert('You have unsaved changes');
           return false;
          }
 }
}
```

Lazy Loading

Lazy loading allows to split the code at module level.

When the application starts, Angular loads all the modules and all the entities of the application.

If the application is very large then the startup will be slower.

To avoid these scalability problems, Angular allows modules to be loaded on demand (i.e. "Lazy Loading") in order to ease the initial load of the application.

Lazy Loading configuration

The configuration of "Lazy Loading" is done in the "Routing" configuration.

The AppRoutingModule "Routing" module can delegate the "Routing" management of a part of the application to another module. This "Lazy Loaded" module will therefore be loaded asynchronously when visiting the "routes" for which it is responsible.

```
{
  path: 'vehicles',
  loadChildren: () => import('./vehicle/vehicle.module').then(m => m.VehicleModule),
}
```

This configuration delegates the "Routing" of all the /vehicles/... part of the application to the Vehicle Module module.

GENERALE PERFORMANCE

Observables

Observables

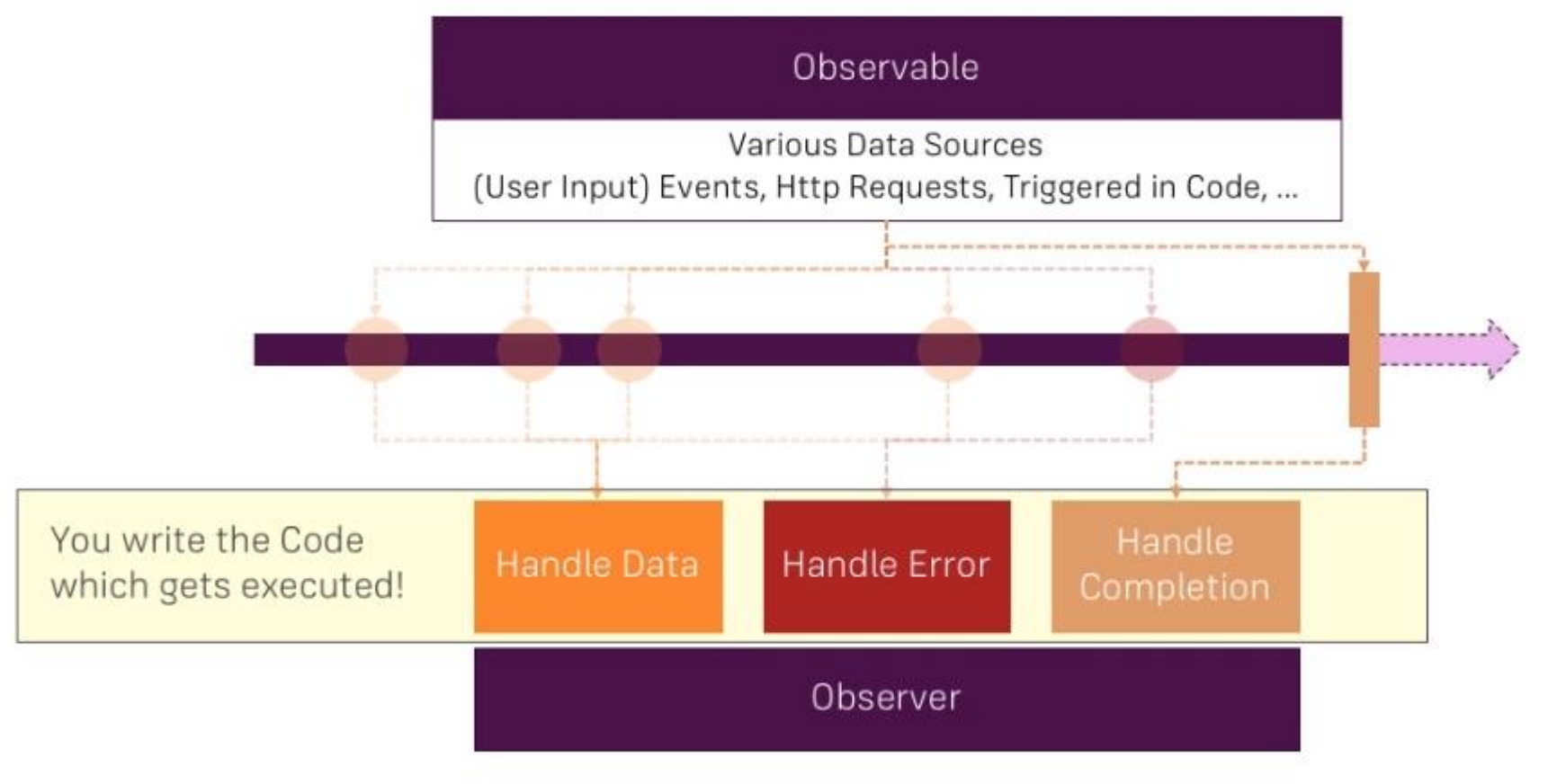
An Observable is an object allowing to make the “link” between publishers and subscribers.

An Observable is basically a function that can return a stream of values to an observer over time.

An observable is a function that creates an observer and attaches it to the source where values are expected, for example, clicks, mouse events from a dom element or an Http request, etc.

Observables will emit events which will be intercepted by observers.

Observables



Observers

It is an object with next(), error() and complete() methods, that will get called when there is interaction to the with the observable

A handler for receiving observable notifications implements the Observer interface.

NOTIFICATION TYPE	DESCRIPTION
next	Required. A handler for each delivered value. Called zero or more times after execution starts.
error	Optional. A handler for an error notification. An error halts execution of the observable instance.
complete	Optional. A handler for the execution-complete notification.

Subscribe to an observable

In the example below we will create an observable using the interval function which produces a self-incrementing value on a regular basis.

```
import { interval } from 'rxjs';

const data$ = interval(1000);
```

As long as you don't subscribe to the "observable", nothing happens because this observable is "lazy".

The common point between all "observables" is the subscribe method which allows you to subscribe to an Observable and be notified of new values, errors or the end of the "stream".

next:
error:
complete:

```
const subscription =data$.subscribe({
   value => console.log(value),
   err => console.error(err),
   () => console.log('DONE!')
});
```

```
const subscription =data$.subscribe({
   (value: number) => { console.log(value)} ,
   err => console.error(err),
   () => console.log('DONE!')
});
```

Unsubscription

The subscribe method returns an object of type Subscription.

This object is mainly used to unsubscribe from an Observable via its unsubscribe method.

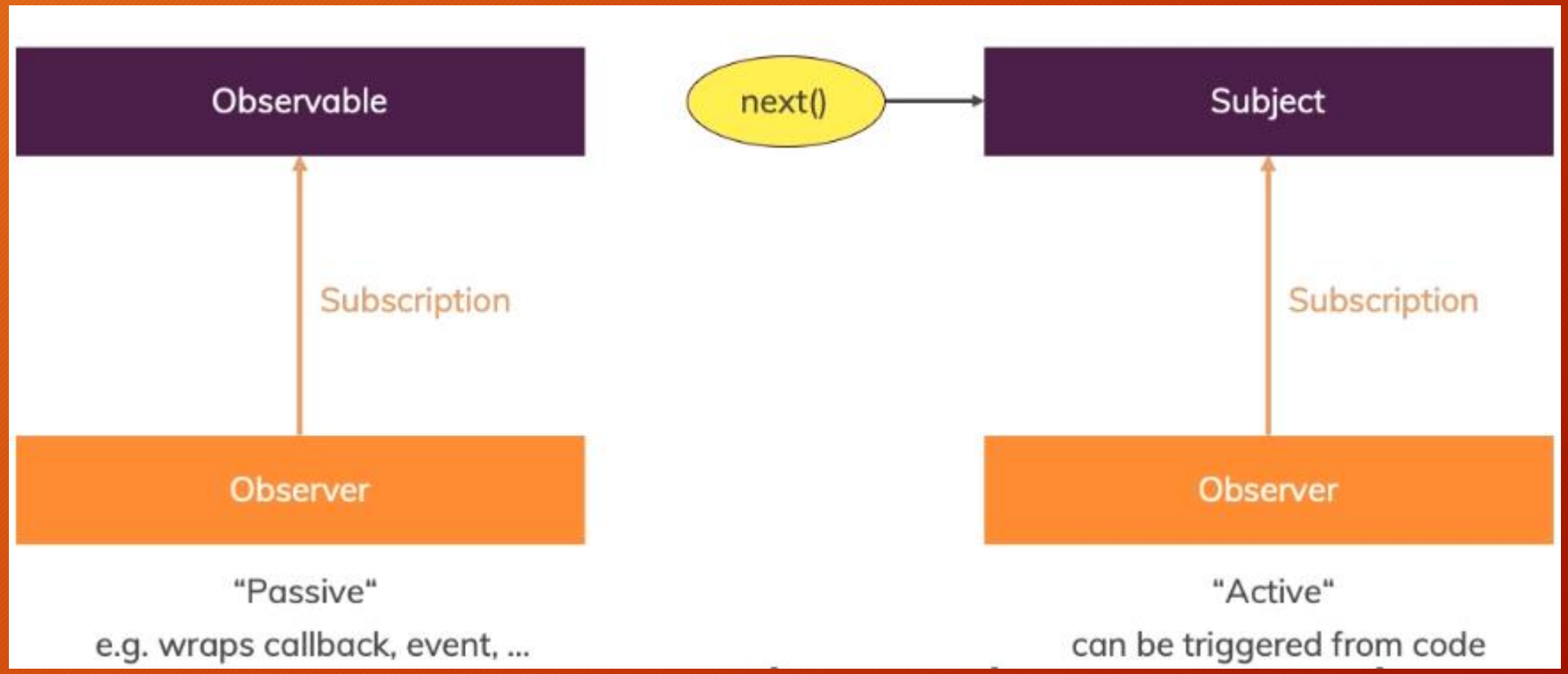
```
subscription.unsubscribe ();
```

The unsubscribe method allows you to:

- unsubscribe the "callbacks": next, error and complete;
- destroy the Observable (interrupt the processing carried out by the Observable)
- possibly free the memory because by unregistering the "callbacks" .

Subject

A Subject is a special kind of Observable



Subject

A Subject is both an observable AND an observer. We can therefore subscribe to it, but also send it values with next:

```
const subject = new Subject<number>();

subject.subscribe((number) => {
  console.log( number);
});

subject.next(1);  // send value
subject.next(2);  // send another
```

A subscriber will only get published values that were emitted *after* the subscription.

BehaviourSubject

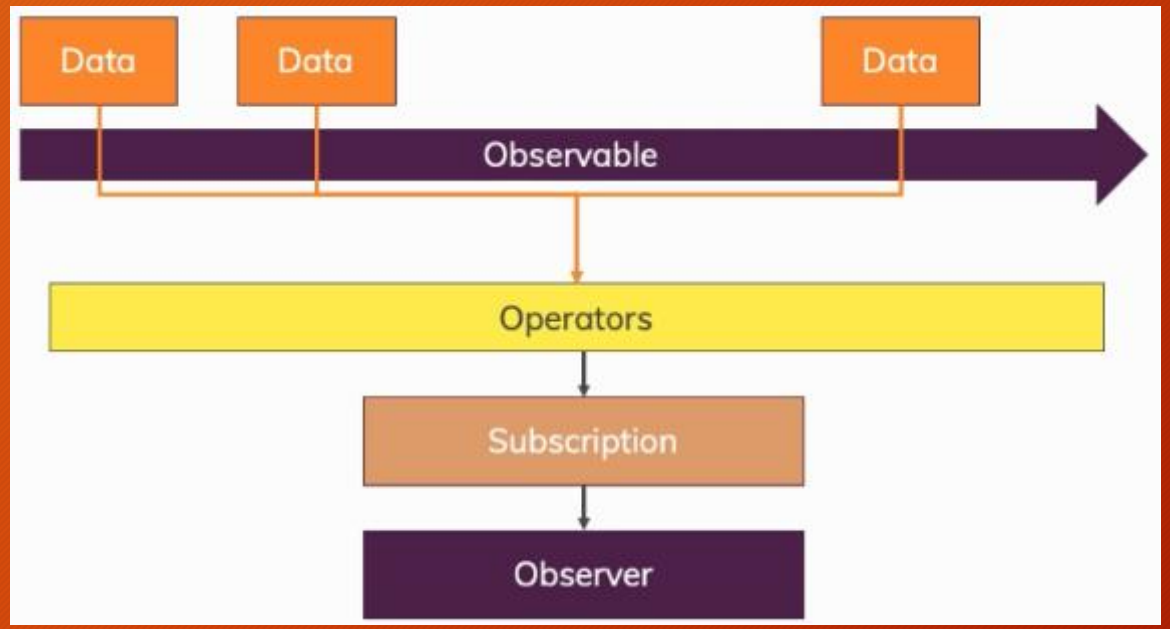
A Behaviour Subject is like a Subject, the difference is that behaviour subject also gives subscribers immediate access to the previously emitted value, even if they haven't subscribed at the point of time that value was emitted.

So we can fetch data, even if it was emitted before we subscribe to the observable.

The BehaviorSubject is initialized with an initial value.

Operators

An operator makes it possible to define an Observable from another by applying some transformations to it. Operators are functions that can be applied to an Observable via the pipe method.



map

The map operator allows you to create a new Observable from the original Observable by transforming each of its values.

```
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//add 10 to each value
const example = source.pipe(map(val => val + 10));
//output: 11,12,13,14,15
const subscribe = example.subscribe(val => console.log(val));
```

```
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

const subscribe = from([1, 2, 3, 4, 5])
                        .pipe(map(val => val + 10));
                        .subscribe(val => console.log(val));
```

ExhaustMap

ExhaustMap, as well as other **Map operators, will substitute value on the source stream with a stream of values, returned by inner function.

It waits for the first observable to complete, and replace it with the inner observable returned insead the function of exhaustMap.

Example:
We have a login page with a login button, where we *map* each click to an login ajax request.

If the user clicks more than once on the login button, it will cause multiple calls to the server.

So we can use exhaustMap to temporarily “disable” the mapping while the first http request is still on the go — this makes sure you never call the server while the current request is running.

filter

The filter operator allows you to keep only the elements for which the predicate function returns true.

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//filter out non-even numbers
const example = source.pipe(filter(num => num % 2 === 0));
//output: "Even number: 2", "Even number: 4"
const subscribe = example.subscribe(val => console.log(`Even number: ${val}`));
```

Take

Emits only the first specified number of values emitted by the source Observable.

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

//emit 1,2,3,4,5
const source = of(1, 2, 3, 4, 5);
//take the first emitted value then complete
const example = source.pipe(take(2));
//output: 1 2
const subscribe = example.subscribe(val => console.log(val));
```

Async pipe

The async pipe in angular will subscribe to an Observable and return the latest value it has emitted.

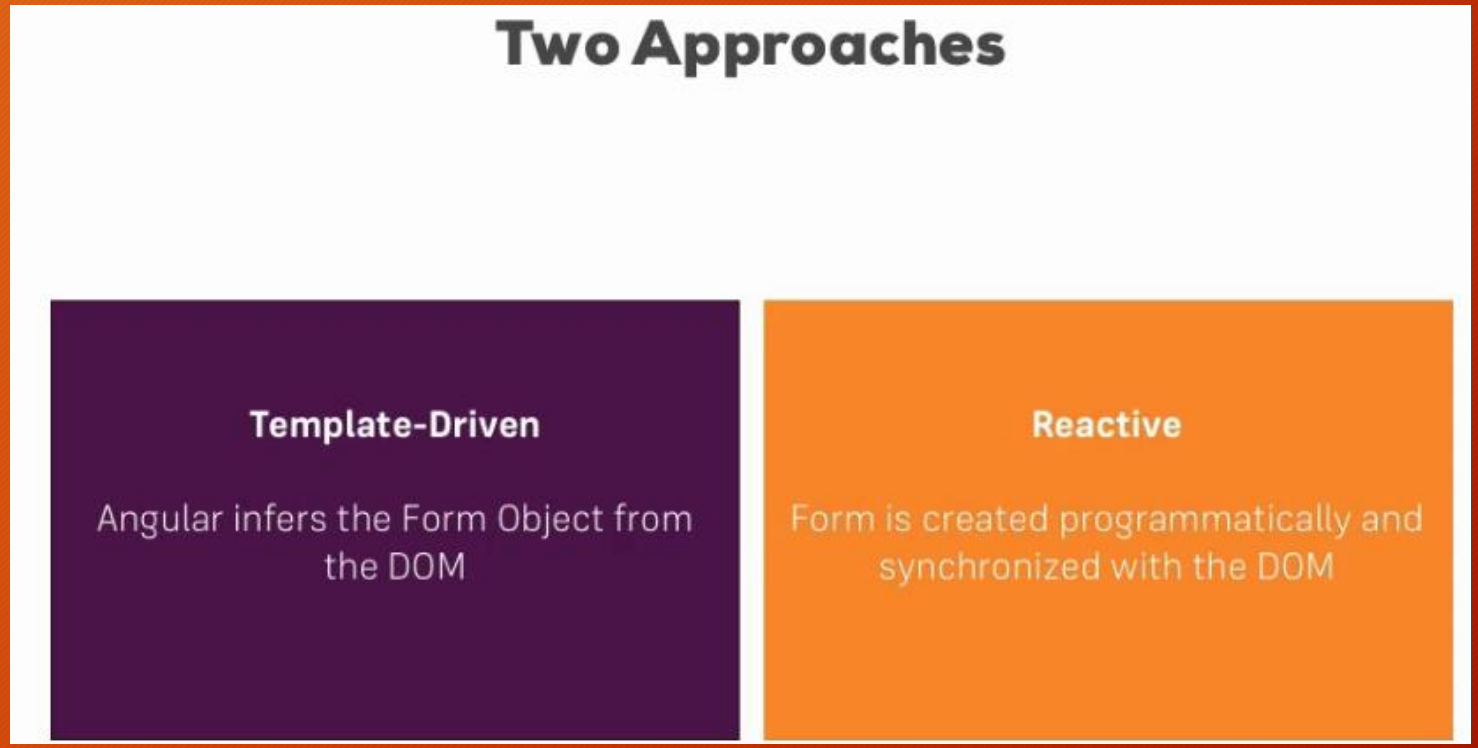
When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

```
@Component({
        selector: 'async -pipe-observable',
        template: '<div><code>observable|async</code>: Time: {{ time | async }}</div>'
})

export class AsyncPipeObservableComponent {

        time = new Observable<string>((observer: Observer<string>) => {
                setInterval(() => observer.next(new Date().toString()), 1000);
        });

}
```

GENERALE PERFORMANCE

Forms: Template driven

Forms


Two Approaches
Template-Driven
Angular infers the Form Object from the DOM
Reactive
Form is created programmatically and synchronized with the DOM

TD vs Reactive

Template Driven Forms Features	Reactive Forms Features
Easy to use	More flexible, but needs a lot of practice
Suitable for simple scenarios	Handles any complex scenarios
Two way data binding(using [(NgModel)] syntax)	No data binding is done
Minimal component code	More component code and less HTML markup
Automatic track of the form and its data(handled by Angular)	Adding elements dynamically

TD: Registering the Controls

First of all the module **FormsModule** must be imported in the app.module

To register a control we have to add NgModel and a name for this control

Create Form using HTML Tag

```
<form>
  <input type="text" id="username" class="form-control" ngModel name="username">
</form >
```

TD: Submitting and Using the Form

Use ngSubmit to submit the Form

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  ...
</form >
```

```
import { NgForm } from '@angular/forms';
...

onSubmit(form:NgForm){
  console.log(form);
}
```

TD: Adding Validation to check User Input

Use HTML5 validation to validate forms

```
<input type="text"
id="username"
class="form-control"
ngModel
name="username"
required minlength="4">
```

TD: Using form state

```
<button type="submit" [disabled]="!f.valid">Submit</button>
```

```
input.ng-invalid, select.ng-invalid {
  border: 1px solid red;
}
```

TD: Outputting Validation Error Messages

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

<span class="help-block" *ngIf="!email.valid && email.touched">
 Please enter a valid email
</span>
```

TD: Grouping Form Controls

```
<div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">
    <input type="text" ngModel name="username">
    <input type="email" ngModel name="email">
</div>

<p *ngIf="!userData.valid && userData.touched">user Data is invalid</p>
```

TD: Handling Radio Buttons

```
<div class="radio" *ngFor="let g of genders">
  <label>
    <input type="radio" name="gender" ngModel [value]="g">{{g}}
  </label>
</div>
```

```
genders=['male','female'];
```

TD: Using Form Data

```
user={
  username:'',
  email:'',
  secretQuestion:'',
  answer:'',
  gender:''
};

...
onSubmit(myForm:NgForm){
  this.user.username=myForm.value.userData.username;
  this.user.email=myForm.value.userData.email;
  this.user.gender=myForm.value.gender;
  this.user.secretQuestion=myForm.value.secret;
  this.user.answer=myForm.value.questionAnswer;
}
```

TD: Reset a form

Use reset method to reset the Form

```
this.myForm.reset();
```

Fourth Day

Forms: Reactive

Making Http Requests

Authentication

GENERALE PERFORMANCE

Forms: Reactive

Reactive: Adding a basic form control

There are three steps to using form controls.

1. Register the reactive forms module(**ReactiveFormsModule**) in your app. This module declares the reactive-form directives that you need to use reactive forms.

2. Generate a new FormControl instance and save it in the component.

3. Register the FormControl in the template.

Reactive: Creation of the form

```
signupFrom: FormGroup;

ngOnInit(){
 this.signupFrom= new FormGroup({
   'username': new FormControl(null),
   'email': new FormControl(null),
   'gender': new FormControl('male')
 });
}
```

```
signupFrom: FormGroup;

constructor(protected fb: FormBuilder) {}

ngOnInit(){
 this.signupFrom=  this.fb.group({
   'username':  [null],
   'email': [null],
   'gender': ['male']
 });
}
```

Reactive: Syncing HTML and Form

```
<form [formGroup]="signupFrom">
  <div class="form-group">
    <label for="username">Username</label>
    <input  type="text" id="username"  class="form-control"
            formControlName="username"
    >
  </div>
</form>
```

Reactive: submit the form

The FormGroup directive listens for the submit event emitted by the form element and emits an ngSubmit event that you can bind to a callback function.

```
<form [formGroup]="signupFrom" (ngSubmit)="onSubmit()">
  ...
  <button type="submit">Submit</button>
</form>
```

```
onSubmit(){
  console.log(this.signupFrom.value);
}
```

Reactive: Validation

```
import { Validators } from '@angular/forms';

...

this.signupFrom= new FormGroup({
  'username': new FormControl(null, Validators.required),
  'email': new FormControl(null, [Validators.required,Validators.email]),
  'gender': new FormControl('male')
});
```

Reactive: Getting Access to Controls

```
<span
*ngIf="!signupFrom.get('username').valid && signupFrom.get('username').touched"
class="help-block">
Please enter a valid username
</span>
```

```
input.ng-invalid.ng-touched {
  border: 1px solid red;
}
```

Reactive: Creating nested form groups(1/2)

```
this.signupFrom= new FormGroup({
  'gender': new FormControl('male'),
  'userData':new FormGroup({
    'username': new FormControl(null, Validators.required),
    'email': new FormControl(null, [Validators.required,Validators.email])
  })
});
```

Reactive: Creating nested form groups(2/2)

```
<form [formGroup]="signupFrom">
 <div class="form-group">
  <label for="username">Username</label>
  <input  type="text"  id="username"  formControlName="username"  class="form-control">
 </div>
  ...
 <div formGroupName="userData">
  <div class="form-group">
   <label for="username">Username</label>
   <input  type="text"  id="username"  formControlName="username"  class="form-control”>
  </div>
  <div class="form-group">
   <label for="email">email</label>
   <input  type="text"  id="email"  formControlName="email"  class="form-control">
  </div>
 </div>
</form>
```

Reactive: Creating dynamic forms

In order to declare controls dynamically we use FormArray

You can initialize a form array with any number of controls, from zero to many, by defining them in an array.

```
this.signupFrom= new FormGroup({
    ...,
  'hobbies':new FormArray([])
});
```

Reactive: Access the FormArray control

We can create a get method that return the controlls of the array

```
getControls() {
  return (<FormArray>this.signupForm.get('hobbies')).controls;
}
```

Define a method to dynamically insert an alias control into the alias's form array. The FormArray.push() method inserts the control as a new item in the array.

```
onAddHobby(){
  const control=new FormControl(null,Validators.required);
  (<FormArray>this.signupFrom.get('hobbies')).push(control);
}
```

Reactive:Display the form array in the template

```
<form [formGroup]="signupFrom">
...
 <div formArrayName="hobbies">
  <h4>Your hobbies</h4>

  <button class="btn btn-default" type="button" (click)="onAddHobby()">Add Hobby</button>

  <div class="form-group" *ngFor="let hobby of getControls(); let i=index">
   <input type="text" class="form-control" [formControlName]="i">
  </div>
 </div>
</form>
```

Updating parts of the data model

There are two ways to update the model value:

1. Use the setValue() method to set a new value for an individual control. The setValue() method strictly adheres to the structure of the form group and replaces the **entire** value for the control.

2. Use the patchValue() method to replace any properties defined in the object that have changed in the form model.

Creating a Custom Form Validator

```
import { AbstractControl, ValidationErrors } from '@angular/forms'

export function gte(control: AbstractControl): ValidationErrors | null {
  const v=+control.value;
  if (isNaN(v)) {
    return { 'gte': true, 'requiredValue': 10 }
  }

  if (v <= 10) {
    return { 'gte': true, 'requiredValue': 10 }
  }
  return null;
}
```

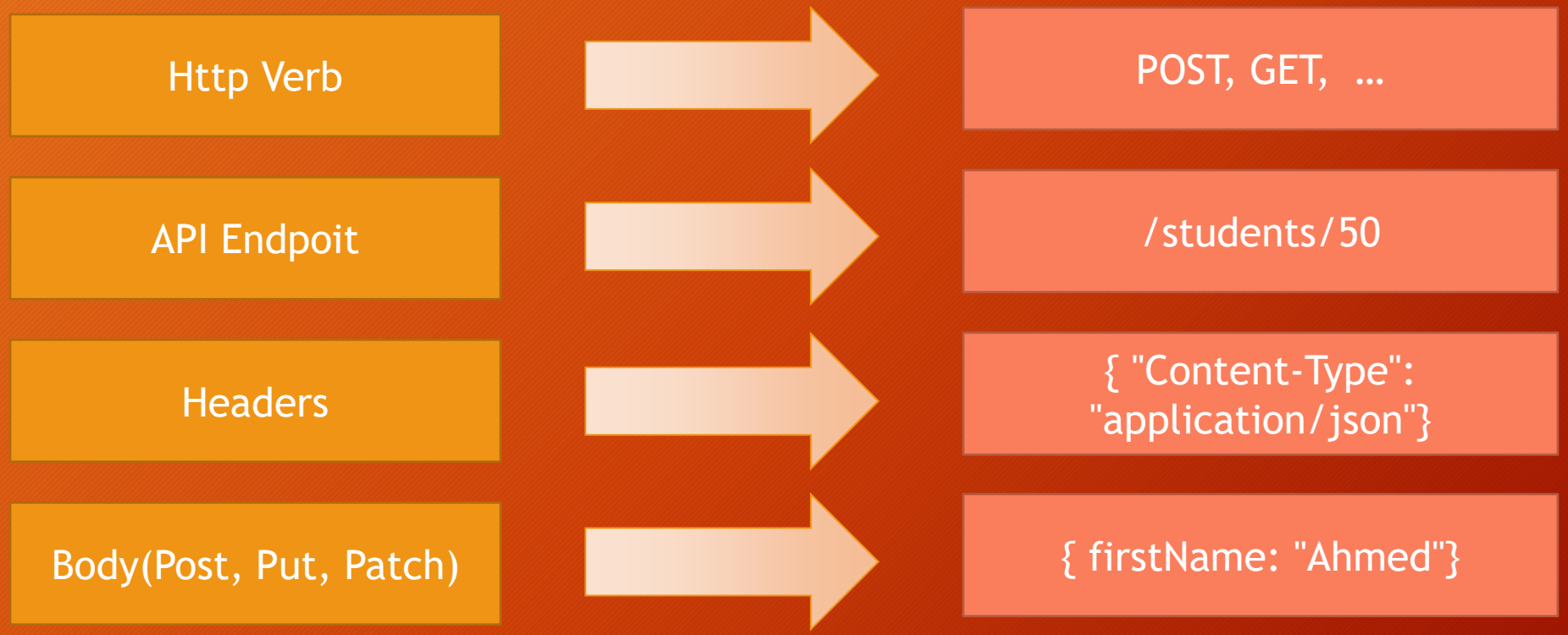
```
import { gte } from './gte.validator';
...
export class AppComponent {

  myForm = new FormGroup({
    numVal: new FormControl('', [gte]),
  })
}
```

```
<div>
  <label for="numVal">Number :</label>
  <input type="text" id="numVal" name="numVal" formControlName="numVal">
  <div *ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">
    <div *ngIf="numVal.errors.gte">
      The number should be greater than {{numVal.errors.requiredValue}}
    </div>
  </div>
</div>
```

GENERALE PERFORMANCE

HttpClient

Anatomy of HTTP Requests



Use of HttpClient

must import the module HttpClientModule in app.module

import { HttpClientModule } from '@angular/common/http'

HttpClient is an Angular service; we can therefore recover it with the Dependency Injection.

```
constructor(private httpClient: HttpClient) {
}
```

Sending a POST Request

```
export class StudentService {
      constructor(private http: HttpClient) {}

      addStudent (student: Student): Observable<Student> {
        return this.http.post<Student>(this.studentUrl, student);
      }
}
```

```
this.studentService.addStudent(newStudent)
  .subscribe(
    ...
  );
```

Sending a PUT Request

```
export class StudentService {
      constructor(private http: HttpClient) {}

      editStudent(student: Student): Observable<Student> {
        return this.http.put<Student>(`${this.studentUrl}\${student.id}`, student);
      }
}
```

```
this.studentService.editStudent(student)
  .subscribe(
              ...
  );
```

Getting Data

```
export class StudentService {
      constructor(private http: HttpClient) {}

      getStudents(): Observable<Student> {
        return this.http.get<Student>(this.studentUrl);
      }
}
```

```
private fetchStudents() {
  this.studentService..getStudents()
    .subscribe(students => {
                this.students= students;
    });
}
```

Delete request

```
deleteStudent (id: number): Observable<{}> {
  const url = `${this.studentsUrl}/${id}`; // DELETE api/students/42
  return this.http.delete(url);
}
```

```
this.studentService .deleteStudent(hero.id).subscribe(...);
```

Handling Errors

```
private fetchStudents() {
  this.studentService..getStudents()
    .subscribe(
      students => {
              this.students= students;
    } , error => {
    this.error = error.message;
  });
}
```

```
<div class="alert alert-danger" *ngIf="error">
  <h1>An error has occured</h1>
     <p>{{error}}</p>
</div>
```

Setting Headers

```
return this.http
  .get<{ [key: string]: Post }>(
    url,
    {
      headers: new HttpHeaders({'Custom-header' : 'hello'})
    }
  )
```

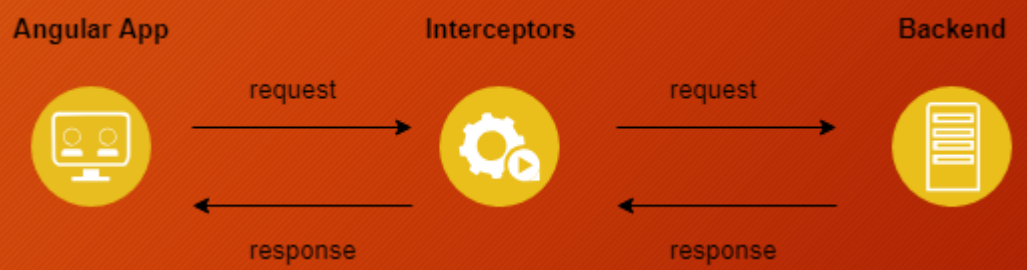
Adding Query Params

```
let searchParams = new HttpParams();
searchParams = searchParams.append('print', 'pretty');

return this.http
  .get<{ [key: string]: Post }>(
    url,
    {
      headers: new HttpHeaders({'Custom-header' : 'hello'}),
      params: searchParams
    }
  )
```

Introducing Interceptors

Interceptors are a unique type of Angular Service that we can implement.

Interceptors allow us to intercept incoming or outgoing HTTP requests using the HttpClient. By intercepting the HTTP request, we can modify or change the value of the request.



Use of Interceptors

First, create a service that implements HttpInterceptor

```
export class AuthInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const modifiedReq = req.clone({
      headers: req.headers.append('Auth', 'xyz')
    });
    return  next.handle(modifiedReq);
  }}
```

It has to be added to the list of all **HTTP_INTERCEPTORS**, which can be done that way in **app.module.ts**:

```
providers: [{provide : HTTP_INTERCEPTORS, useClass: AuthInterceptorService, multi : true}],
```

How we can use multiple interceptors?

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true },
  { provide: HTTP_INTERCEPTORS, useClass: MySecondInterceptor, multi: true }],
```

The interceptors will be **called in the order** in which they were provided.

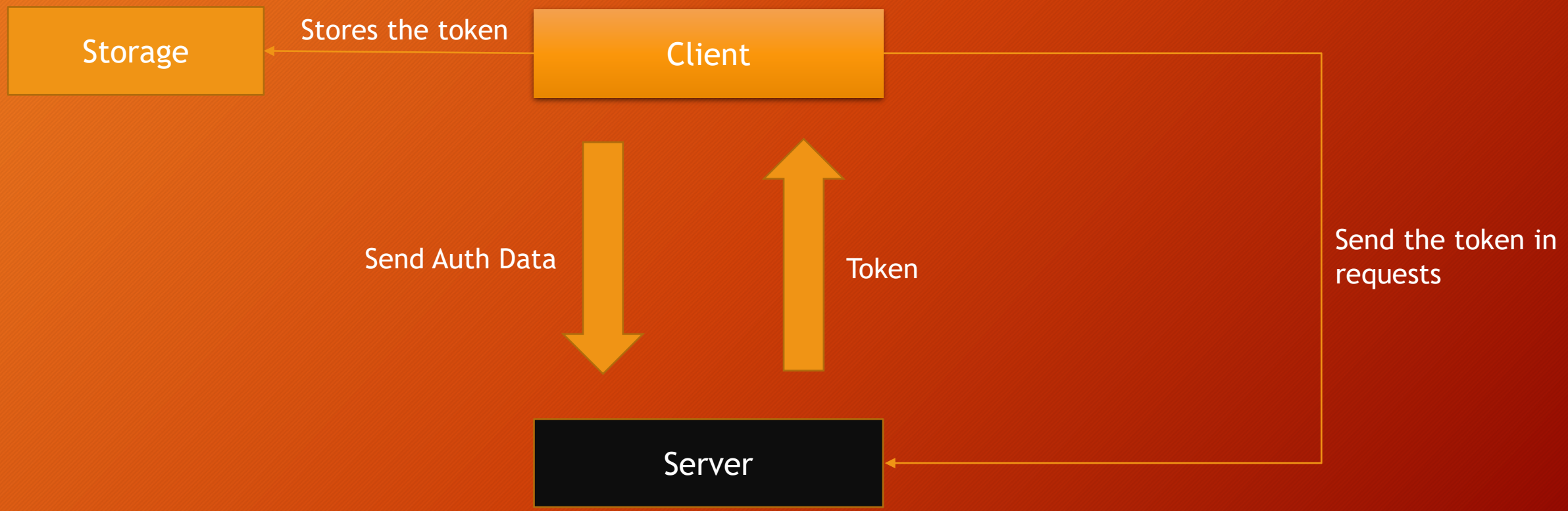
Using multi: true tells Angular that the provider is a multi provider.

With multi providers, we can provide multiple values for a single token in DI.

GENERALE PERFORMANCE

Authentication

How authentication works



QUESTIONS

SESSION END

THANK YOU
for
your
ATTENTION

GENERALE PERFORMANCE

ANNEX

Source code

Source code can be found in this github Link:

https://github.com/youma85/FleetManagement

GENERALE PERFORMANCE

Versions

Introduction & Versions

2

- May 2016
- Complete Re-Write of AngularJs
- Architecture is component based
- Supports ES6 and TS 1 to 1.8

Introduction & Versions

3

- Skipping V3 to avoid a confusion due to the misalignment of the router package's version

Introduction & Versions

4

- Released on March 23, 2017.
- Angular 2 compatible
- Lot of performance improvement
- Added supports for email validation pattern
- Supports TS 2.1 & 2.2
- Instead of writing 2 ngIf, else block is introduced.
- Introduced **HttpClient**, a smaller, easier to use, and more powerful library for making HTTP Requests.

Introduction & Versions

5

- Released on November 1, 2017
- **@angular/http** is replaced with **@angular/common/http** library
- Add supports for Number, Date and Currency pipes
- Build Optimizer and improvements of Material Desing.
- Build optimizations
- Improvements of the perfs
- Supports TS 2.3

Introduction & Versions

6

- Released on May 4, 2018
- No major breaking changes
- I18N introduced (No requirement to build one application by locale)
- Angular CLI Changes: Two new commands have been introduced
  - ng update <package>
  - ng add <package>

Introduction & Versions

7

- Released on October 18, 2018
- Various improvements in the performance
- TypeScript 3.1 Support
- Angular CLI prompt user, to help him to discover the in-built SCSS support or routing.

Introduction & Versions

8

- Released on May 28, 2019
- Internal changes
- @angular/http is no longer supported, use @Angular/common/http instead
- ViewChild changed temporary
- Supports TS 3.4
- Use a new compiler(IVY)

Introduction & Versions

9

- was released on February 6, 2020
- Smaller builds
- Internal changes
- Speed and performance
- Faster testing
- Improved CSS class and style binding
- Improved Debugging.
- ViewChild returned as before

Introduction & Versions

10

- was released on June 24, 2020
- Internal changes
- Keeping Up to Date with the Ecosystem(TypeScript 3.9, TSLint v6)
- made several new deprecations and removals from Angular.

Introduction & Versions

11

- was released on Nov 11, 2020
- Faster Builds
- The only IE version now still being supported is 11
- Improved Logging and Reporting
- Supports TS 4.0

Introduction & Versions

12

- was released on May 12, 2021
- Adding Nullish Coalescing operator(??)
- Improvements in styling
- Deprecating support for IE11
- Adding some improvements
- Supports TS 4.2

Introduction & Versions

13

- Release on November 3, 2021
- IE11 is no longer supported.
- Angular supports both RxJS 6 and 7
- Supports TS 4.4
- Tests Enhancements
- CLI Enhancements

Introduction & Versions

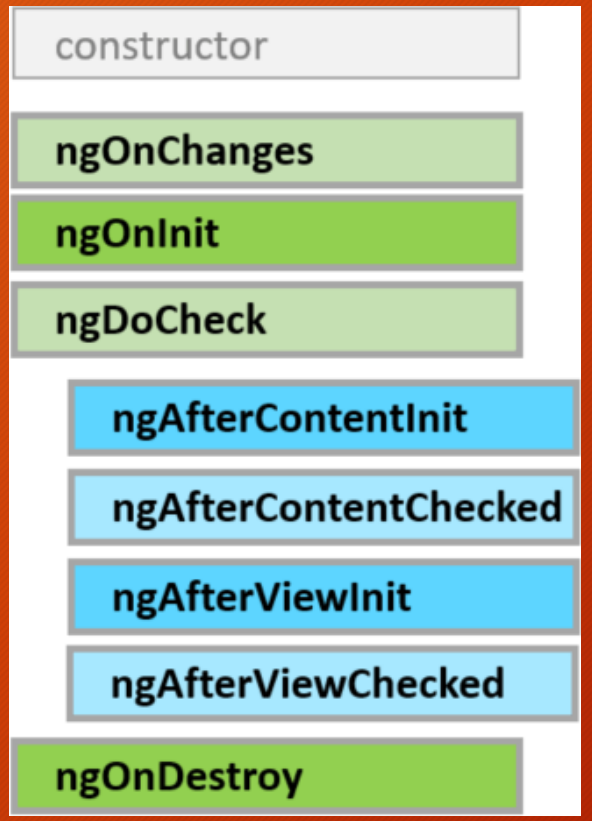
14

- Released on 2nd June 2022
- **Typed Angular Forms**
- **Angular CLI Enhancements**

GENERALE PERFORMANCE

LifeCycle Hooks

Lifecycle Hooks



ngOnChanges()

This is the very first lifecycle hook, it is called right after your class gets initialized and the component is created the ngOnChanges() is called.

Because Angular counts that very first class initialization as a data property change. So the hook that gets called once a data property change occurs is ngOnChanges().

This hook is basically called after the constructor is called and any other time there is a property change inside your component.

To use this hook we must implement the OnChanges interface.

ngOnInit()

This is the second lifecycle hook called by Angular, it is called right after the very first ngOnChanges hook is called. It is only called once, it initializes the component, sets and displays component input properties.

It is the most important lifecycle hook in Angular as it signals the activation of the created component. For the fact that this hook is called only once, it is therefore great for fetching data from external sources like servers and APIs.

To use this hook we must implement the OnInit interface.

ngDoCheck()

This is the third Angular lifecycle hook that gets called on a component.

It is called during every change detection run, Angular has an internal system that goes around the component processes every so often looking for changes that the compiler cannot detect on its own.

To use this hook we must implement the DoCheck interface.

ngAfterContentInit()

This is the fourth lifecycle hook Angular calls after a component has been initialized.

This hook is called only once immediately after the first ngDoCheck hook is called, it is a kind of ngDoCheck but for content projected into the component view with ng-content.

To use this hook we must implement the AfterContentInit interface.

ngAfterContentChecked()

This is the fifth lifecycle hook Angular calls after a component has been initialized.

It is called after the content projected into a component view is initialized, after the ngAfterContentInit hook and every subsequent ngDoCheck hook is called.

To use this hook we must implement the AfterContentChecked interface.

ngAfterViewInit()

This is the sixth lifecycle hook Angular calls after a component has been initialized.

It is called only once after the very first ngAfterContentChecked hook is called. It is called after Angular initializes component views and the subsequent child views under each component, this will have to include the views displayed through content projection too and that is why it is called after the ngAfterContentChecked hook.

To use this hook we must implement the AfterViewInit interface.

ngAfterViewChecked()

This is the seventh lifecycle hook Angular calls after a component has been initialized.

It is called after Angular checks the component views and the subsequent child views under each component for changes, this includes the views displayed through content projection too. It is called after the ngAfterViewInit hook and every subsequent ngAfterContentChecked hook.

To use this hook we must implement the AfterViewChecked interface.

Testing Fundamental Concepts

Angular supports two types of testing

- Unit Tests
- End To End Tests

Unit tests are written in spec.ts files.

The e2e file test are stored in an e2e folder, and end with e2e-spec.ts.

Testing Framwork in Angular

Angular supports two main framework:

- `Jasmine/Karma: Used for unit tests`
- `Protractor: Used for E2E tests`

These framworks are configured in the following config file:

- `Karma: karma.config.js`
- `Protractor: e2e/protractor.conf.js`

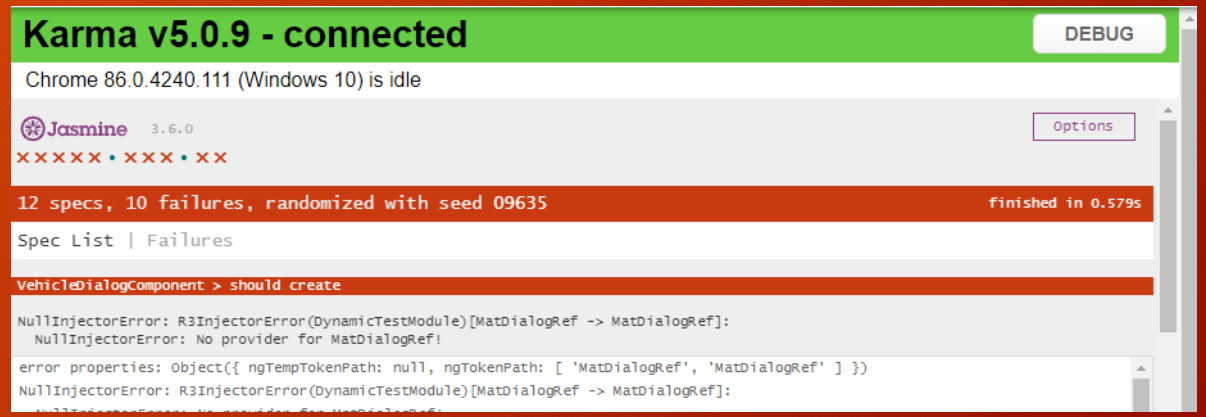
Angular Unit Testing

Unit Testing is a type of testing where the focus of the test is to test a particular piece of the application.

The test can be for components, pipes, directives, services,…

Tests are written in Jasmine framework.

To test we run the command: ng test



This comman will:

- Compile code
- Start a karma server and gave it a port number
- Execute the unit tests.
- Open a new window of a browser with a report of tests.

Angular E2E Testing

E2E tests means automating the application’s workflow for a functionality e2e.

To test we run the command: ng e2e

Jasmine(1/3)

Jasmine is an extensible framework dedicated to tests on "browser" and on NodeJS.

It includes everything you need for:

- define test suites (describe and it functions),
- implement assertions of all kinds (expect function),
- etc

Jasmine (2/3)

For example if we wanted to test this function:

```
function helloWorld() {
  return 'Hello world!';
}
```

We would write a Jasmine test spec like so:

```
describe('Hello world', () => { (1)
  it('says hello', () => { (2)
    expect(helloWorld()) (3)
        .toEqual('Hello world!'); (4)
  });
});
```

1. **describe(string, function):** function defines what we call a Test Suite, a collection of individual Test Specs.
2. **The it(string, function):** function defines an individual Test Spec, this contains one or more Test Expectations.
3. **The expect(actual):** expression is what we call an Expectation. In conjunction with a Matcher it describes an expected piece of behaviour in the application.
4. **The matcher(expected):** expression is what we call a Matcher. It does a boolean comparison with the expected value passed in vs. the actual value passed to the expect function, if they are false the spec fails.

Jasmine (3/3): Setup and Teardown

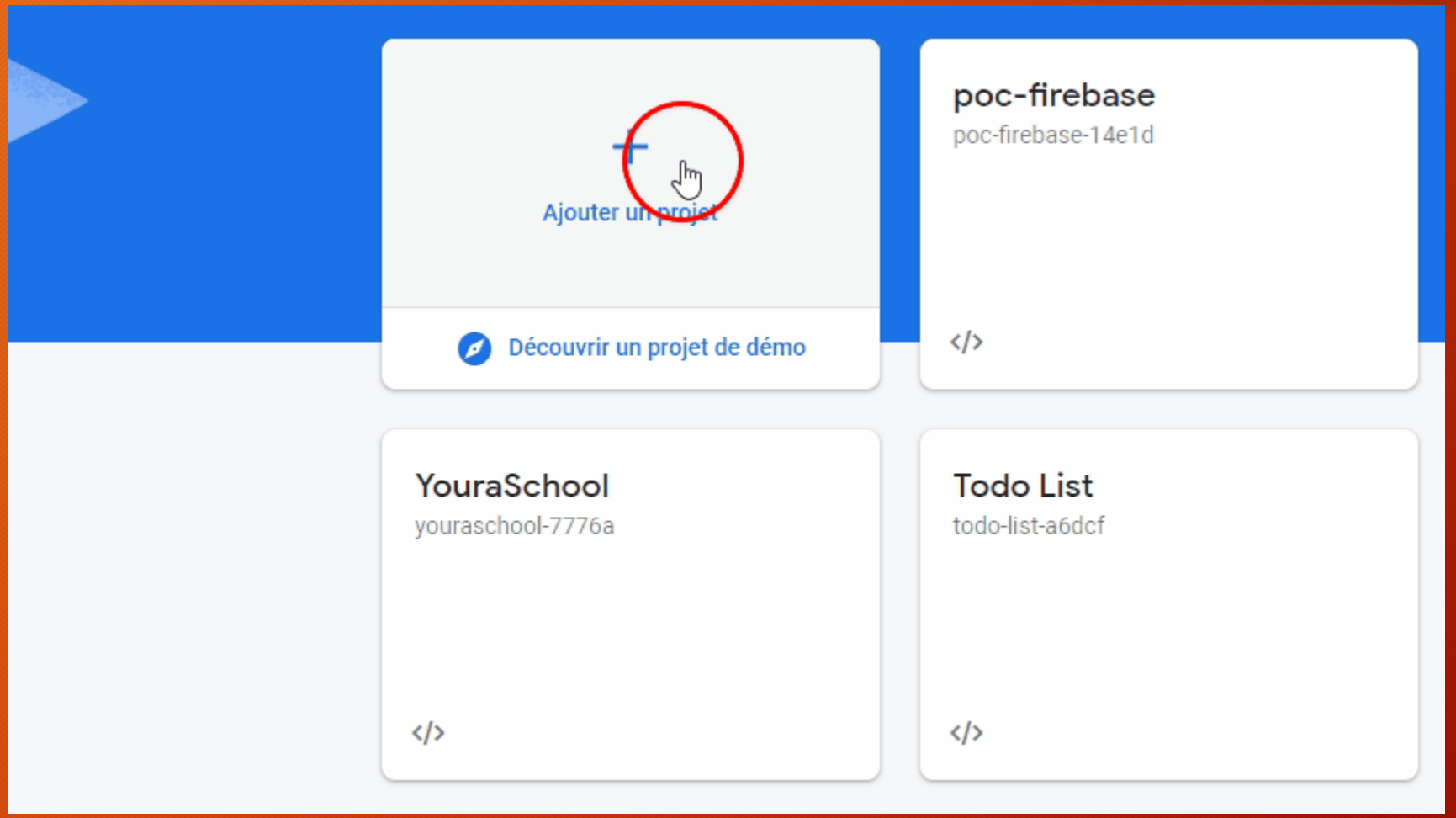
beforeAll:
This function is called once, before all the specs in a test suite (describe function) are run.

afterAll:
This function is called once after all the specs in a test suite are finished.

beforeEach:
This function is called before each test specification (it function) is run.

afterEach:
This function is called after each test specification is run.

Create a firebase project



Create an application