Angular

By: Youssef RAFII

February 2024

Plan

First Day	Introduction
	Typescript
	Setup & installation
	Basics
	Directives
Second Day	Communication between Components
	Pipes
	Services & DI
Third Day	Routing
	Observables
	Forms
Fourth Day	Http Requests
	Authentication

First Day

- Introduction to Angular and its features
- Typescript
- Installation
- Basics
- Directives

GENERALE PERFORMANCE

Introduction to Angular and its features

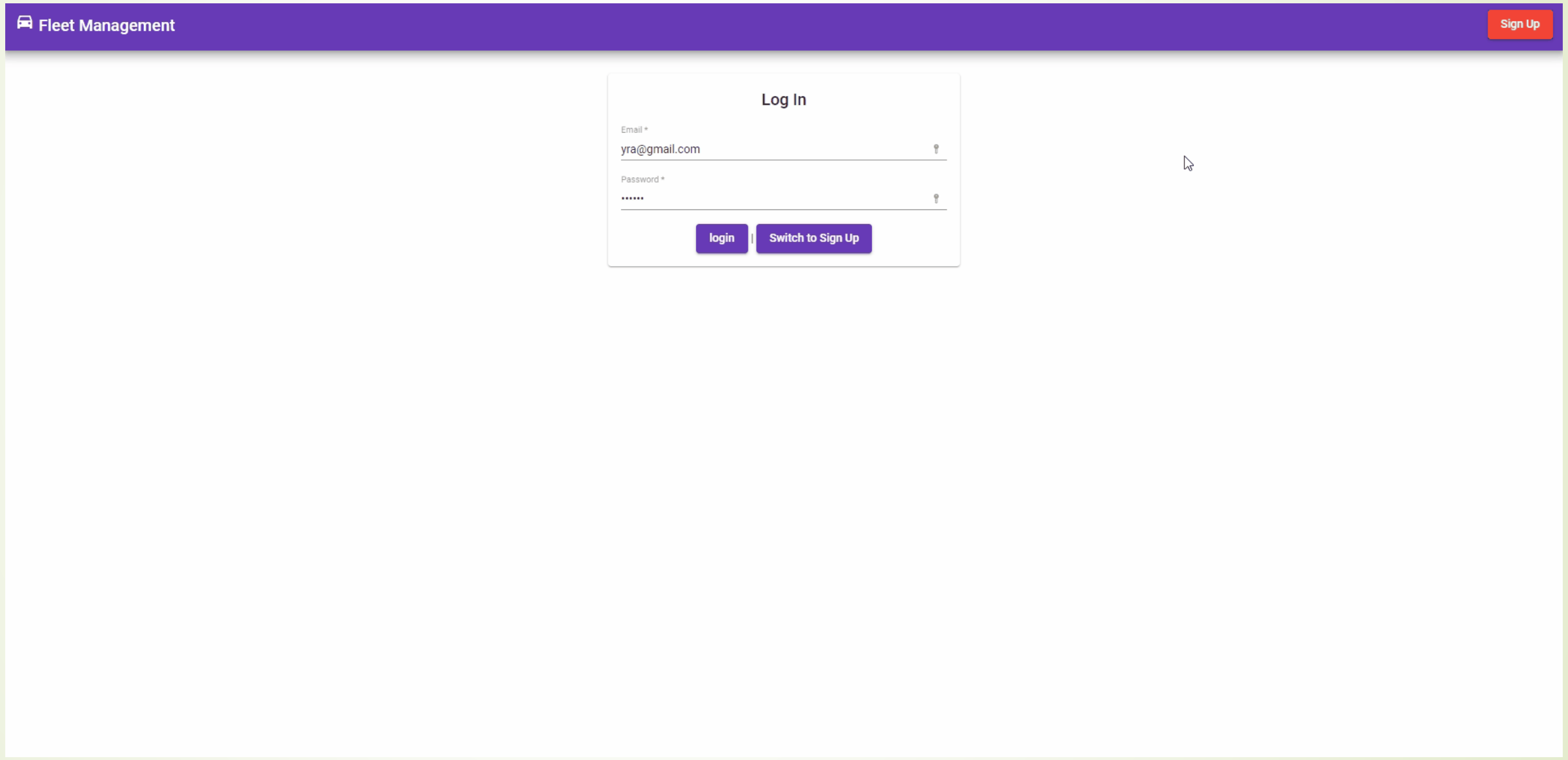
Angular: A powerful framework for building Single-Page Applications

- Angular is a powerful framework for building Single-Page Applications (SPAs)
- Angular uses a component-based architecture, which allows for the creation of reusable components
- Angular uses TypeScript, a statically-typed superset of JavaScript, which provides:
  - added type safety
  - better maintainability
- Angular has built-in support for data binding and event handling, making it easy to manage the flow of data between components

What are Single-Page Applications (SPAs)

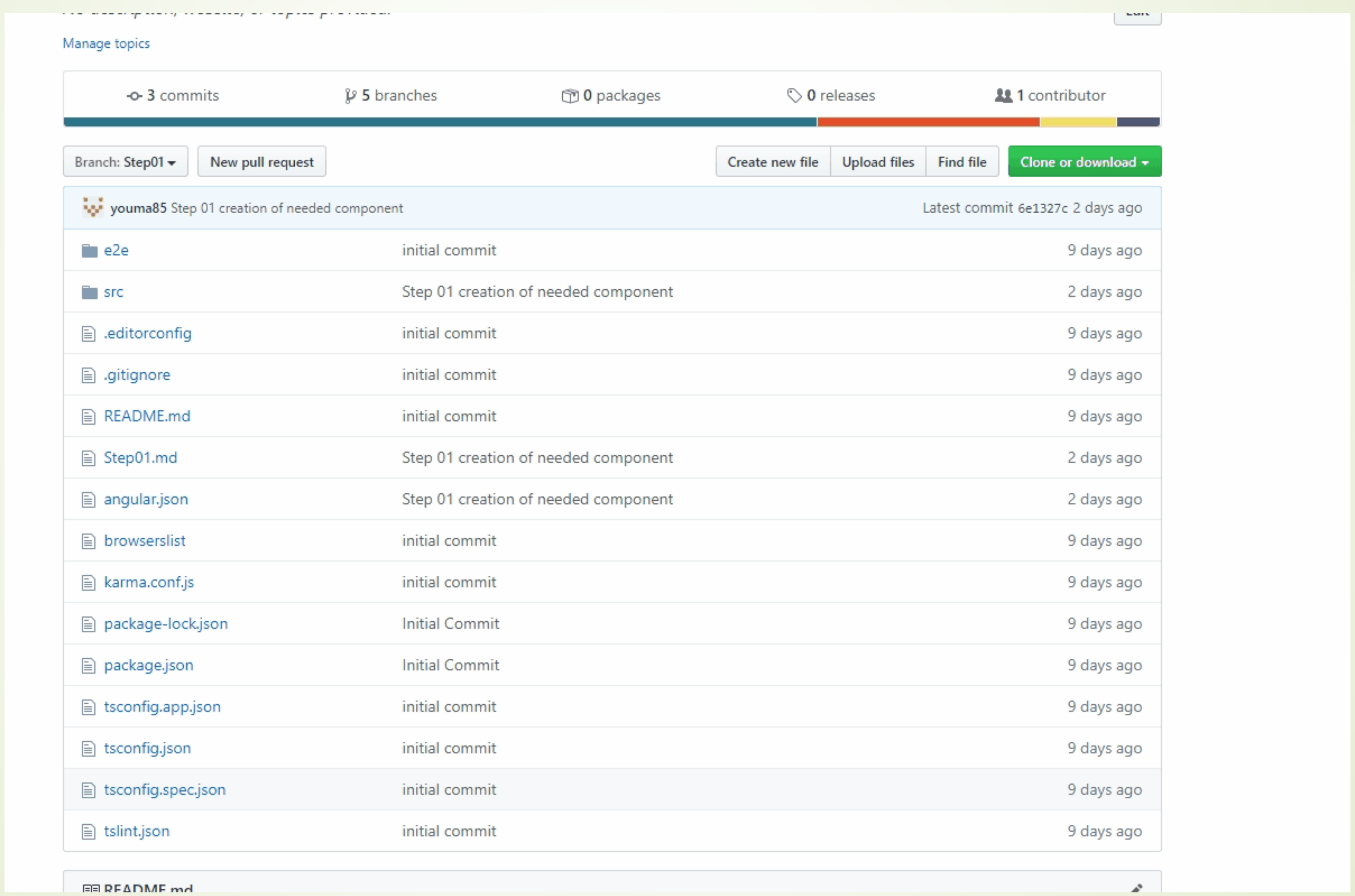
- Explanation of SPAs
  - SPAs are web applications that load a single HTML page and dynamically update that page as the user interacts with the app
  - SPAs provide a more seamless user experience, as they do not require the browser to refresh the page each time a new view is loaded
  - SPAs use JavaScript to dynamically update the content of the page, which results in faster load times and a more responsive user interface
- How Angular can be used to build SPAs
  - Angular provides a powerful set of tools for building SPAs, including its component-based architecture, data binding, and routing capabilities
  - Angular also includes a powerful set of features such as Dependency Injection, Directives, and Pipes, which make it easy to build complex and scalable applications

What will be done during this training



Email *
yra@gmail.com
Password *
••••••

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

https://github.com/youma85/FleetManagement

GENERALE PERFORMANCE

TypeScript

Introduction to TypeScript

- TypeScript is strongly typed programming language based on JavaScript that adds features such as classes, interfaces, and type annotations. It is developed and maintained by **Microsoft** and is increasingly used by Angular developers.

- 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.

Understanding data types in TypeScript

- TypeScript includes a number of built-in data types such as number, string, boolean, and others. These types can be used to define the type of variables and function arguments.

```
let name: string = "John";
let age: number = 30;
let isStudent: boolean = false;
```

Declaring variables with specific data types

```
let name: string = "John";
console.log(typeof name); // Output: string
```

Using the typeof operator to check the data type of a variable

```
let value: number | string = "John";
value = 30;
```

Using union types to define multiple possible types

```
let names: string[] = ["John", "Mike", "Sara"];
let ages: Array<number> = [25, 30, 35];
```

Working with arrays

Understanding data types in TypeScript

```
let person: [string, number] = ["John", 30];
person[0] = "Mike";
person[1] = 35;
```

Working with tuples

```
let value: any = "John";
value = 30;
value = { name: "John", age: 30 };
```

Using **any** Type

```
function printName(name: string): void {
  console.log(name);
}
```

Using the **void** type

It's important to note that **any** and **void** should be used with caution, as they can make your code less type-safe.

The **any** type allows for any value to be assigned to a variable

The **void** type is used for functions that do not return a value.

Understanding functions in TypeScript

- Functions are a way to group a set of statements together to perform a specific task. TypeScript supports several types of functions, including function declarations, function expressions, and arrow functions.

```
// Function Declarations
function add(a: number, b: number): number {
  return a + b;
}

// Function Expressions
let multiply = function(a: number, b: number): number {
  return a * b;
}

// Arrow Functions
let divide = (a: number, b: number): number => a / b;
```

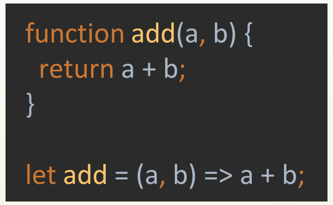
```
// Optional and Default Parameters
function printName(firstName: string, lastName?: string): void {
  console.log(firstName + (lastName ? ' ' + lastName : ''));
}
function printAge(age: number = 30): void {
  console.log(age);
}
```

```
// Rest Parameters
function addAll(...nums: number[]): number {
  let result = 0;
  for (const num of nums) {
    result += num;
  }
  return result;
}
```

```
// Anonymous Functions
let myFunc = function() {
  console.log("Hello World!");
}
```

Arrow Function

- 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.
- Here is a comparison of the same function written using the traditional function declaration and arrow function:



```
function add(a, b) {
  return a + b;
}

let add = (a, b) => a + b;
```

Working with classes and interfaces

- Classes in TypeScript are similar to classes in other object-oriented programming languages. They provide a blueprint for creating objects and can have properties, methods, and constructor.
- Interfaces are a way to define the structure of an object in TypeScript. They describe the shape of the object and can be used to enforce a contract on the object.

```
class Employee {
  name: string;
  age: number;
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
  getName() {
    return this.name;
  }
  setName(name: string) {
    this.name = name;
  }
}
```

Defining a class

```
let emp = new Employee("John", 30);
emp.setName("Mike");
console.log(emp.getName());
```

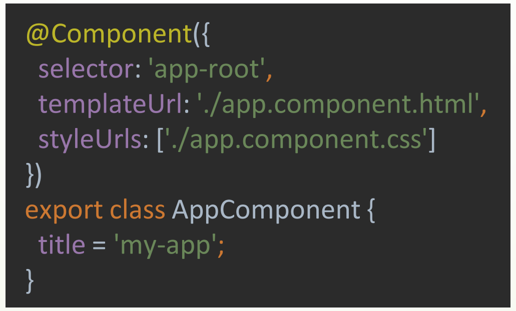
creating an object

```
interface IEmployee {
  name: string;
  age: number;
  getName(): string;
}
class Employee implements IEmployee {
  name: string;
  age: number;
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
  getName() {
    return this.name;
  }
}
```

Implementing interfaces

Working with decorators

- Decorators in TypeScript are a way to add metadata to a class, property, method, or parameter. Decorators are implemented as functions that are invoked with a specific set of arguments, depending on the type of decorator being applied. Decorators can be used to add functionality to a class, such as logging, data binding, and more.



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

let vs var vs const

var	let	const
Function-scoped (global scope)	Block-scoped	Block-scoped

Reassignable
 	
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);
}
	Reassignable
	
	Variable declared with let keyword can be updated but not re-declared.
	
	function varGreeter(){
	  let a = 10;
	  a= 30; // OK
	  let a = 20; //SyntaxError: //Identifier 'a' has already been declared
	  console.log(a);
	}
		Not reassignable

Hoisted to the top of the function or global scope:
{
  console.log(c); // undefined: Due to hoisting
  var c = 2;
}
	Not Hoisted:
	{
	  console.log(b); // ReferenceError: b is not defined
	  let b = 3;
	}
		Not Hoisted

GENERALE PERFORMANCE

Setup & Installation

Setting up your Angular Development Environment

- A proper development environment is essential for building and testing Angular applications
- It includes the installation of required tools such as Node.js and the Angular CLI
- It also includes the configuration of a code editor or integrated development environment (IDE)
- A proper development environment will make it easier to write, debug, and test your code.

Installing Node.js

- Node.js is a JavaScript runtime that allows you to run JavaScript code on your computer
- Node.js is required to run the Angular CLI, which is a command-line tool for creating and managing Angular projects
- Instructions for installing Node.js
  - Download the appropriate version of Node.js for your operating system from the official website: https://nodejs.org/en/download/current/
  - Verify that Node.js has been installed by running the command "node -v" in your terminal

Installing Angular CLI

- The Angular CLI (Command Line Interface) is a command-line tool for creating and managing Angular projects
- It allows you to create new projects, generate new components and services, and manage the dependencies of your project.
- Instructions for installing Angular CLI
  - Open your terminal and run the command: "**npm install -g @angular/cli**"
  - Verify that the Angular CLI has been installed by running the command: "**ng version**"
  - You can now use the Angular CLI to create new projects, generate new components and services, and manage the dependencies of your project.

- 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 version

Angular CLI Commands

- Here are some of the most commonly used commands:
  - ng new: Creates a new Angular project with a default file structure and configuration.
  - ng serve: Runs the application in development mode, and starts a development server that listens on a localhost port.
  - ng build: Builds the application for production.
  - ng test: Runs unit tests for the application.
  - ng e2e: Runs end-to-end tests for the application
  - ng generate or ng g: Generates new code artifacts for the application such as components, services, and pipes.
  - ng add: Installs and configures additional libraries and dependencies for the application.
  - ng update: Updates the version of Angular and its associated packages in the application.
  - ng version: Displays the version of Angular CLI and its associated packages.
  - ng deploy: Deploys the application to a specified platform such as GitHub Pages, Firebase, or Azure.
  - ng doc: Open the official documentation of Angular in the browser.

Setting up a code editor or IDE

- Some popular code editors for Angular development:
  - Visual Studio Code
  - WebStorm
  - Sublime Text
  - Atom

- You can use any editor or IDE that you are comfortable with, as long as it has good support for TypeScript and Angular.

GENERALE PERFORMANCE

Creating an Angular project

Creating your first Angular project

- Angular projects are created using the Angular CLI
- The CLI allows you to quickly create a new project with a basic file structure and configuration
- You can then add components, services, and other features to the project as needed

Creating a new project

- To create a new Angular project, you will use the "**ng new**" command in the Angular CLI
- This command will create a new directory with the specified name and generate a basic file structure and configuration
- Instructions for creating a new project
  - Open your terminal and navigate to the location where you want to create the new project
  - Run the command "ng new projectName"
  - The CLI will generate a new directory with the specified name and create the basic file structure and configuration for the new project.

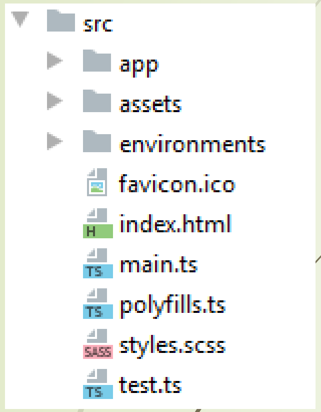
Running the project

- Instructions for running the project
  - Open your terminal and navigate to the root directory of the project
  - Run the command "**ng serve**" to start the development server
  - Open a web browser and navigate to "http://localhost:4200" to see the running project
    - You can modify the default port with the option --port: **ng serve --port 5555**

Angular 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 src Folder


src
app
assets
environments
favicon.ico
index.html
main.ts
polyfills.ts
styles.scss
test.ts

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	The main HTML file for the application.
main.ts	The application's entry point.
polyfills.ts	Few lines of code which make your application compatible for different browsers
styles.css	The global CSS file for the application.
test.ts	Unit test entry point, not part of application

GENERALE PERFORMANCE

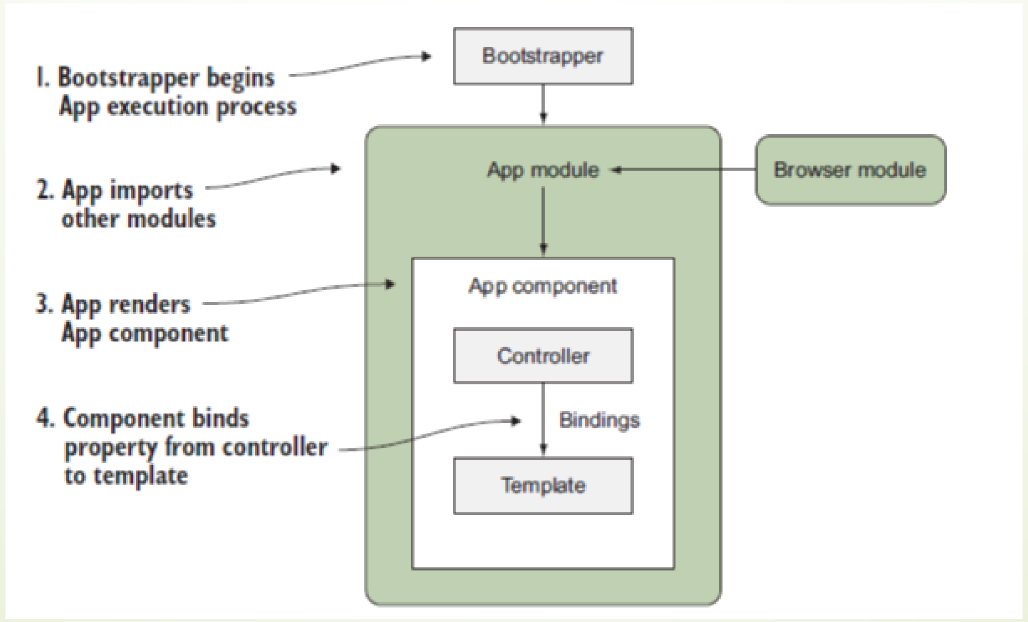
Basics

GENERALE PERFORMANCE

Bootstraping an Angular application

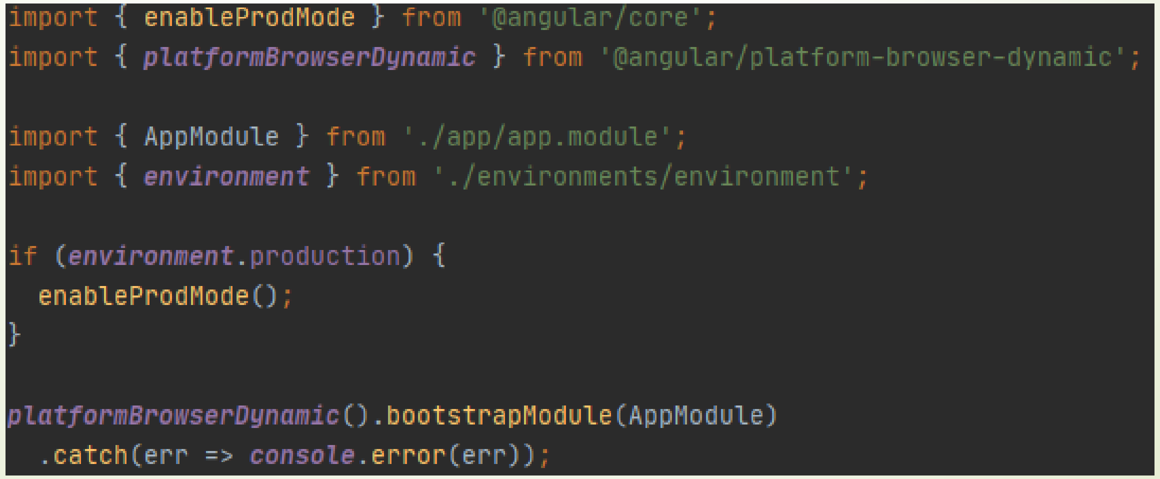
Overview of the bootstrap process

- Bootstrapping refers to the process of starting up an application and initializing it with the necessary resources and configurations.
- Angular bootstraps an application by loading the main.ts file, which then uses the platformBrowserDynamic function to initialize the application and make it ready to run.


1. Bootstrapper begins App execution process
2. App imports other modules
3. App renders App component
4. Component binds property from controller to template
Bootstrapper
App module
Browser module
App component
Controller
Bindings
Template

The main.ts file

- The main.ts file is the entry point of the Angular application.
- It's responsible for bootstrapping the root module of the application.
- In this file, angular will check if the application is running in a production environment. If it is, then it calls the "enableProdMode" function, which enables the production mode of the application.
- Then it calls the "bootstrapModule" function from the "platformBrowserDynamic" object, passing in the root module of the application, "**AppModule**", as an argument. This function is used to start the application.
- If any error occurs during the bootstrap process, it will be caught by the catch block and logged to the console.



```
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));
```

The AppModule

- The AppModule is the root module of your application and it is defined in the app.module.ts file. It is responsible for bootstrapping the application and providing the necessary services and components for the application to run.
- In the AppModule, you will also find the imports of the BrowserModule and the CommonModule.
  - The **BrowserModule** is necessary for running the application in the browser and provides services such as the Renderer and the Browser
  - The **CommonModule** provides common directives such as ngFor and ngIf.
- It's important to note that the AppModule is only loaded once when the application starts, and it serves as the entry point for the rest of the application. So It should only import and provide the services and components that are needed throughout the entire application.

GENERALE PERFORMANCE

ANGULAR

App module

```
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 other modules that are used in the app.

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.

The Root Component

- The root component is typically the first component that gets loaded by the application.
- It is defined in the AppModule and is the parent of all other components in the application.
- It is responsible for bootstrapping the application and providing a container for other components to be rendered within.

```
@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

The index.html

- The index.html is the main HTML file of the application, and it is the starting point of the application.
- It is the first file that is loaded when the application is run, and it contains the root element of the application.
- It loads all the necessary scripts and stylesheets.
- It also includes the <app-root> element, which is the element that corresponds to the root component of the application.

GENERALE PERFORMANCE

Modules

Introduction to modules

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

- Modules in Angular are a way to organize and structure an application. They are used to group together related components, directives, pipes and services.
- Modules help to make the application more modular and maintainable by encapsulating functionality and making it easier to reuse code.
- Modules help to organize the application by grouping related functionality together. This makes it easier to understand and navigate the application, as well as making it easier to test and maintain.

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

Creating a Module

- This can be done by running the command:

  "**ng generate module [module name]**"

- In the "module.ts" file, you will find the **@NgModule** decorator, which is used to configure the module.
- The @NgModule decorator is a function that takes an object literal as its argument. This object literal contains the properties that configure the module.
- The properties that are commonly used in the @NgModule decorator include:
  - **declarations**: an array of components, directives, and pipes that belong to this module.
  - **imports**: an array of modules whose exported classes are needed by component templates declared in this module.
  - **exports**: an array of the components, directives, and pipes that should be available to modules that import this module.
  - **providers**: an array of dependency injection providers.

Types of Modules in Angular

Root module	• Is the **<u>main module</u>** in an Angular application. It acts as the entry point for the application and is responsible for bootstrapping the app by providing the necessary components, directives, and services for the app to run.
Feature modules	• They together related functionality for a **<u>specific feature</u>** or section of an application. • They help to organize and structure the application by encapsulating the components, directives, pipes, and services that are related to a specific feature in one place.
Shared modules	• They contain components, directives, pipes, and services that are **<u>used across multiple parts of the application</u>**. • By placing these items in a shared module, they can be easily imported and reused in other parts of the application, reducing code duplication and improving maintainability.
Core modules	• Provide **<u>services and singleton objects</u>** that are used throughout the application. • These services and objects are often used for application-wide functionality such as logging, error handling, or data storage.
Third-party modules	• These are modules that are created by **<u>external libraries</u>** and are used to add functionality to the application. Examples include libraries for handling HTTP requests, form validation, or animations.

GENERALE PERFORMANCE

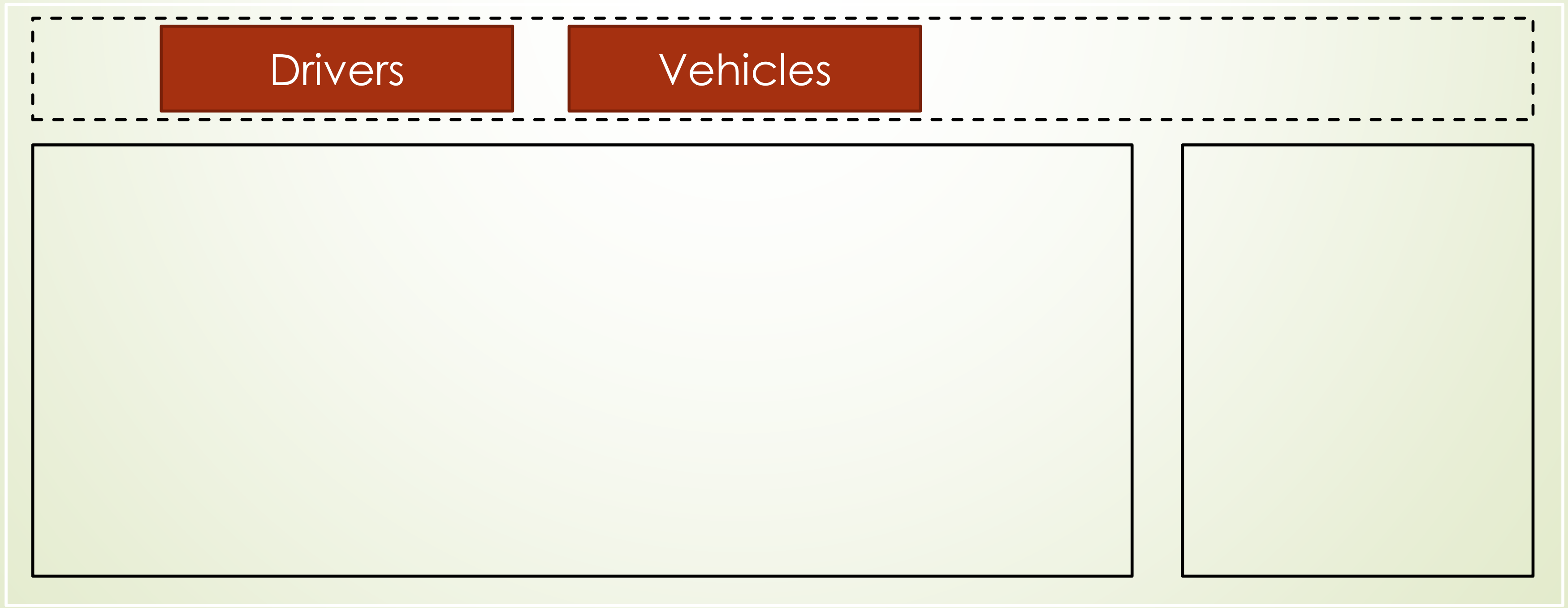
Understanding the Component Architecture

Introduction to Component Architecture

- In Angular, the application is divided into small, independent, and reusable components
- Each component has a specific responsibility, and they interact with each other to build the overall functionality of the application

What are Components?

- Components are the building blocks of an Angular application
- They are essentially classes that control a portion of the screen (a view)
- Each component has an associated template, which defines the layout and content of the view, and a class, which defines the behavior and logic of the view
- Components can also interact with other components and services to share data and functionality



Creating a Component

- Components are created using the Angular CLI
- The CLI will generate a new component, including a class, a template, and a CSS file
- The component class is decorated with the @Component decorator, which defines the metadata for the component,
- Run the command: "**ng generate component componentName**" ,
- The CLI will generate a new directory with the specified name, containing the class, template, and CSS files for the new component, and add the component to the declarations array of the module

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

}
```

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

Understanding the @component decorator

- The @Component decorator is an important part of creating a component in Angular. It is used to define the metadata of a component, such as its selector, template, styles, and other properties. This decorator takes an object as its argument, and the properties of that object define the behavior of the component:
  - selector: The CSS selector used to match elements in the template and create an instance of the component
  - template: The template that defines the view for the component
  - templateUrl: The location of the template file
  - styles: An array of styles to be applied to the template
  - styleUrls: An array of URLs for style files

From Angular 14:

- standalone: Mark the component as standalone

What are Standalone Components?

- Standalone Components are Angular components that can be declared and used independently without the need to be declared in an NgModule. This means you can have components, directives, and pipes that are self-contained and can be imported directly where needed.

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

// Define the component
@Component({
  selector: 'app-greeting',
  template: `<p>Hello, {{ name }}!</p>`,
  standalone: true // Mark the component as standalone
})
export class GreetingComponent {
  @Input() name: string;
}
```

```
// Usage in another component
import { Component } from '@angular/core';
import { GreetingComponent } from './greeting.component';

@Component({
  selector: 'app-root',
  template: `<app-greeting name="World"></app-greeting>`,
  imports: [GreetingComponent] // Import the standalone
component
})
export class AppComponent {}
```

Ways of selector

- In Angular, a component's selector can be defined in several ways:
  - **Element selector “app-root”:** This type of selector is defined as an HTML element, such as <app-root>.
  - **Attribute selector “[app-root]”:** This type of selector is defined as an HTML attribute, such as <div app-root>. The component will be rendered as an HTML element with this attribute.
  - **Class selector “.app-root”:** This type of selector is defined as a CSS class, such as <div class="app-root">.
  - **Identifier selector “#app-root”:** This type of selector is defined as a unique identifier such as <div #app-root>.

GENERALE PERFORMANCE

Understanding the template syntax

Introduction to template syntax

- Angular, templates are used to define the layout and content of a component's view
- Templates use a special syntax that is based on HTML, but also includes additional features for binding data, handling events, and more

Interpolation

- Interpolation is the process of binding a component's property to the template
- It is denoted by double curly braces {{ }}
- The property is evaluated and its value is inserted into the template

```
{{driver.firstName}}
```

- In this example, the firstName property of the driver is evaluated and its value is inserted into the template

Property binding

- Property binding is used to bind a component's property to a DOM element property
- It is denoted by square brackets []
- It allows you to set the value of a DOM element property to a component property

```
<input [value]="name">

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

- In this examples, the disabled property of the button is bound to the result of the function isValid() also the value of the input element is bound to the name property of the component

Event binding

- Event binding is used to bind a DOM event to a component's method
- It is denoted by parentheses ()
- When the event occurs, the associated method is executed

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

```
onClick(){
  …
}
```

- In this example, the onClick() method of the component is executed when the button is clicked.

Two-way data binding

- Two-way data binding is used to bind a component property to a DOM element and vice versa
- It is denoted by the combination of square brackets [] and parentheses ()
- When the property changes, the DOM element is updated, and when the DOM element changes, the property is updated
- With two way databinding we combine property and event binding.
- You need to enable the ngModel directive by adding the FormsModule to the imports[] array in the AppModule.

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

- In this example, the ngModel directive is used to bind the name property of the component to the value of the input element, and vice versa.

GENERALE PERFORMANCE

Directives

Introduction to Directives

- Directives are a way to add behavior to elements in the DOM. They allow you to extend the HTML syntax to create custom elements and attributes.
- They are used to change the appearance and behavior of elements in the DOM.
- Different types of directives
  - Structural Directives: They change the structure of the DOM by adding, removing or replacing elements. They are prefixed with * (e.g. *ngIf, *ngFor).
  - Attribute Directives: They change the appearance or behavior of an element. They do not add or remove elements from the DOM. They are not prefixed with * (e.g. ngClass, ngStyle).

Structural directives: *ngIf

- The *ngIf directive is used to conditionally add or remove an element from the DOM.
- It takes a boolean expression as an input and adds the element to the DOM if the expression is true, and removes it if the expression is false.

```
<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>
```

Structural directives: *ngFor

- The *ngFor directive is used to iterate over a collection of data and generate a template for each item.
- It takes an iteration variable and an of or in clause, and is used in conjunction with an HTML template.

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

Structural directives: *ngSwitch

- The *ngSwitch directive is used to conditionally apply a CSS class or styles to an element, based on the value of an expression.
- It works similarly to a switch statement in JavaScript, where the expression is matched to a case and the corresponding template is used.

```
<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>
```

Attribute directives: ngClass

- It allows you to dynamically set CSS classes on an element.
- The basic syntax is: [ngClass]="{'classname':condition}" where classname is the CSS class and condition is a boolean expression.

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;
}
```

Attribute directives: ngStyle

- It allows you to dynamically set inline styles on an element.
- The basic syntax is: [ngStyle]="{property:'value'}" where property is the CSS property and value is a string or expression.

with a Object:

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

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

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

GENERALE PERFORMANCE

ANGULAR

Directives: Custom directives

- Custom directives are a way to extend the functionality of HTML elements by creating your own directive that can be used throughout the application.
- You can create the directive using this command: **ng g d directive_name**

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

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

}
```

```
@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

GENERALE PERFORMANCE

ANGULAR

Directives Listener

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

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

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

GENERALE PERFORMANCE

ANGULAR

Using HostBinding to Bind to Host Properties

Use the @HostBinding decorator to bind properties of the directive host element.

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

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

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

@Input

- Use the @Input decorator to define inputs for the directive.

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string;
  @HostBinding('style.backgroundColor') backgroundColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.backgroundColor = this.highlightColor;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.backgroundColor = null;
  }
}
```

```
<p appHighlight [highlightColor]="'yellow'">This text will be highlighted in yellow when hovered over.</p>
```

Second Day

- Communication between Components
- Pipes
- Services & DI

GENERALE PERFORMANCE

Communication between component

Overview

- In Angular, component binding allows for communication between a parent component and its child component(s).
- This can be done through various ways such as:
  - input/output properties
  - services
  - event emitters.
- Each method allows for data to flow in a specific direction:
  - Input properties allow data to flow from the parent component to the child component.
  - Output properties allow data to flow from the child component to the parent component.
  - Services allow for bidirectional communication between components.
  - Event emitters allow for custom events to be emitted from a child component and caught by the parent component.

Input Properties

- Input properties allow data to flow from a parent component to a child component.
- By default, component property cannot 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().

```html
<app-child [name]="NameInTheParent"></app-child>
```

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

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

@Component({
  selector: 'app-child',
  template: `
    <p>{{ name }}</p>
  `
})
export class ChildComponent {
  @Input() name: string;
}
```

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 Properties

- 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>
```

GENERALE PERFORMANCE

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 (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 {
  drivers: Driver[] =  [
  ];

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

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

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

Local Reference

- A local reference is a way to access an element or a directive in the template of a component.
- It allows you to assign a name to an element or directive and then use that name to access its properties and methods.
- Local references are typically created using the # symbol in the template.

Creating a local reference in the template:

```
<input #nameInput type="text">
```

Accessing the local reference:

```
@ViewChild('nameInput') nameInput: ElementRef;
```

Using the local reference:

```
console.log(this.nameInput.nativeElement.value);
```

Communication between component using ViewChild

- ViewChild is a decorator that allows a component to access a child component or a DOM element.
- It can be used to access a child component or a DOM element by using its template reference variable.

```
// parent.component.ts
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-parent',
  template: `
    <app-child #childRef></app-child>
    <button (click)="changeChildValue()">Change Child Value</button>
  `
})
export class ParentComponent {
  @ViewChild('childRef') child: ChildComponent;

  changeChildValue() {
    this.child.value = 'new value';
    this.child.changeValue();
  }
}
```

```
// child.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <p>{{ value }}</p>
  `
})
export class ChildComponent {
  value = 'original value';

  changeValue() {
    this.value = 'changed value';
  }
}
```

GENERALE PERFORMANCE

Pipes

Overview of Pipes

- Pipes are a way to transform data in Angular templates
- Pipes can be used to format data, such as converting a date to a specific format or to transform data, such as converting a number to currency.
- Angular provides a set of built-in pipes, such as date, currency, and uppercase.
- 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 (|).
- Custom pipes can also be created for specific use cases.
- Angular has several native "pipes": https://angular.io/api?type=pipe

GENERALE PERFORMANCE

ANGULAR

Built-in Pipes

- DatePipe : used to format date values. It provides a simple way to display dates in a variety of formats, such as short date format, long date format, time format, etc.
- UpperCasePipe : converts a string to uppercase.
- LowerCasePipe : transforms a string to all lowercase letters.
- CurrencyPipe : formats a number as currency
- DecimalPipe : formats a number as a decimal using the appropriate number of decimal places and a thousands separator.
- PercentPipe : format a number as a percentage.
- JsonPipe : convert an object into a JSON string
- SlicePipe : create a new array or string that contains a subset of the elements from the original array or string.

Using Pipes with Parameters

- Pipes can also accept parameters to customize their behavior.
- The syntax for using pipes with parameters is:

  {{ value | pipeName:parameter1:parameter2 }}.

  The colon (:) is used to separate the pipe name from the parameters.

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

```
{{ price | currency:'EUR'}}
```

Chaining Pipes

- Multiple pipes can be applied to a value by chaining them together.

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

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

Creating a Custom Pipe

- To create a custom pipe, use the "ng generate pipe" command.
- A custom pipe should have a name, and a method to transform the input data called "transform".
- The created class will implement the PipeTransform interface, and decorated with @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;
 }
}
```

```
{{ name | 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

Using services and dependency injection

Services

- Services are a way to share data and functionality across different components in an Angular application
- They provide a way to organize and reuse code, making it easier to maintain and test your application
- Services can be used to handle common functionality such as making HTTP requests, logging, and data storage

Creating a Service

- Creating a Service using the Angular CLI:
  - Run the command: ng generate service <service-name>
  - This will create a new service file in the src/app directory with the provided name and add it to the providers array in the app.module.ts file.

- Creating a Service Manually:
  - Create a new file in the src/app directory with the desired service name and the .service.ts extension.
  - Add the following code to the file:

```
@Injectable()
export class ServiceName {
  //service methods and properties
}
```

The @Injectable decorator is required to make the service available for dependency injection

  - Add the service to the <u>providers</u> array in the app.module.ts file

Services After 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() :

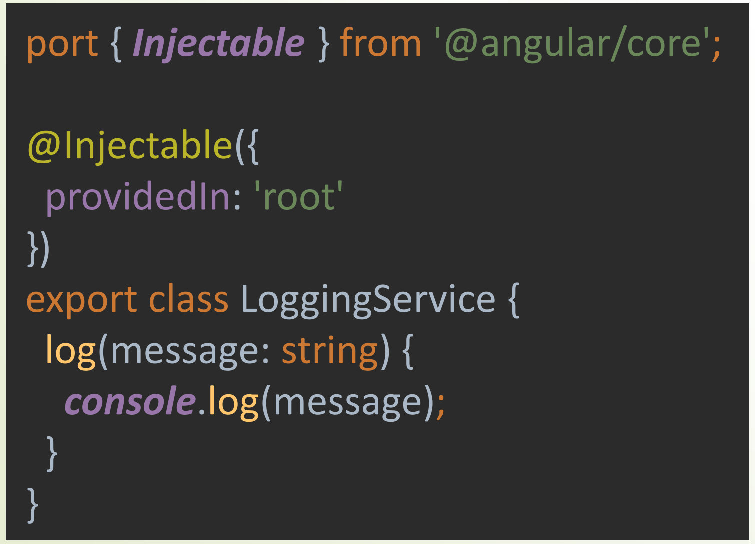
```
@Injectable({
  providedIn: 'root'
})
export class ServiceName {
}
```

Dependency injection

- Dependency injection is a design pattern that allows an application to automatically provide dependencies (or services) to other parts of the application.
- In Angular, dependency injection is used to provide services to components. The framework is responsible for managing the creation and lifetime of services and providing them to components that need them.
- When a component is created, its constructor is called, and the framework will automatically provide any services that the component requires. This allows the component to use the service without having to create it itself.

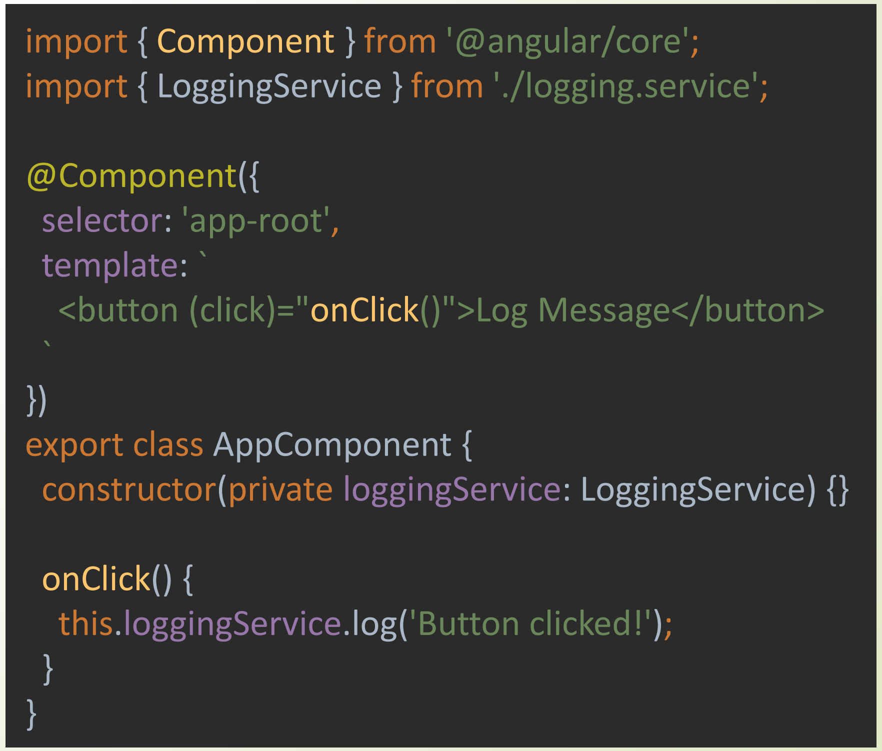
Use of DI

- In this example, Angular's Dependency Injection system is used to provide an instance of the "LoggingService" to the "AppComponent" class. The "loggingService" property of the "AppComponent" class is then used to call the "log()" method of the service.



```
port { Injectable } from '@angular/core';

@Injectable({
 providedIn: 'root'
})
export class LoggingService {
 log(message: string) {
  console.log(message);
 }
}
```



```
import { Component } from '@angular/core';
import { LoggingService } from './logging.service';

@Component({
 selector: 'app-root',
 template: `
  <button (click)="onClick()">Log Message</button>
 `
})
export class AppComponent {
 constructor(private loggingService: LoggingService) {}

 onClick() {
  this.loggingService.log('Button clicked!');
 }
}
```

Scopes 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

Routing

Introduction to Routing and Navigation

- **What is Angular Routing?**: In a single-page app (SPA), routing allows you to change what the user sees by showing or hiding portions of the display corresponding to specific components. Instead of fetching entirely new pages from the server, Angular's router interprets browser URLs as instructions to change the view within the app.

- **Why Use Angular Routing?**: Routing is essential for creating seamless navigation experiences. It enables users to move between different views (pages) within your Angular application without full page reloads.

Setting up the Router

- To use routing in an Angular application, you need to import the **RouterModule** and add it to the imports array of the @NgModule decorator
- It is recommended to place this configuration in a dedicated module AppRoutingModule imported by the "root module" AppModule.

```
import { Routes } from '@angular/router';

const appRoutes: Routes = [];

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

```
ng generate module app-routing --flat
```

Configuring Routes

- Routes are configured using an array of route configuration objects
- Each route configuration object has a path and a component properties
- The path property is a string that corresponds to the URL path
- The component property is the component that should be displayed when the path is navigated to

```
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 {
}
```

Configuring Routes since Angular 17

- In Angular 17, with the introduction of standalone components, the requirement of modules has been reduced. This change also impacts routing.
- Instead of setting up routing in a module, you can now configure it directly in your main.ts file. You can pass the router configuration in the bootstrapApplication method.

```
import {bootstrapApplication} from "@angular/platform-browser";
import {AppComponent} from "./app/app.component";
import {provideRouter} from "@angular/router";


const routes: Routes = [
  // other routes
];

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes)
  ]
});
```

Adding parameters to the routes

- You can add parameters to the routes to make them dynamic
- For example, to pass an id parameter in the URL, you can define the route as follows:

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

Router Outlets

- Router outlets are placeholder in the Angular template that the router dynamically inserts components into as the user navigates.
- They allow the dynamic rendering of different components based on the current URL, making it possible to manage multiple views within a single application.
- 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

- The RouterLink directive in Angular is a directive used to bind a clickable HTML element to a specific route in the application's router.

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

- This code creates a link to the “drivers" route in the application, and when the user clicks the link, they will be navigated to the associated component.
- 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.

GENERALE PERFORMANCE

ANGULAR

Navigating Programmatically

- Navigation is done using the router's navigate method

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


…


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

- The Router class in Angular is a service that provides the navigation and URL manipulation capabilities. It allows you to navigate to different parts of your application using the navigate method.

Styling Active Router Links

- The RouterLinkActive directive in Angular is used to add or remove CSS classes from an element when the associated router link is active.
- This directive is often used to visually indicate the current active route in the application

```
<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}:

This object is used to set properties such as exact which allows you to specify if the active link should only be active when the entire URL is an exact match

Nested Routes in Angular

- Nested routes in Angular refer to a mechanism of organizing routes within an application by nesting child routes inside parent routes.
- This helps to keep the structure of the application organized, making it easier to manage

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

- **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(1/3)

In Angular, the ActivatedRoute service is used to access the route parameters of the current route. There are two ways to access the route parameters:

1. using ActivatedRoute.snapshot.params
2. using ActivatedRoute.params observable.

Fetching Route Parameters(2/3)

ActivatedRoute.snapshot.param: used to get the route parameters synchronously.

This means that the route parameters are fetched immediately and the page will not be updated even if the route parameters change later.

It is useful when you only need to access the route parameters once, such as when you need to retrieve data from the server based on the route parameter.

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

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

Fetching Route Parameters(3/3): Reactively

ActivatedRoute.params observable is used to get the route parameters asynchronously.
This means that the route parameters are fetched when the observable is subscribed to, and the page will be updated if the route parameters change later.

It is useful when you need to continuously monitor the route parameters, such as when you need to update the page content based on the route parameter changes.

```
constructor(private route: ActivatedRoute) {

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

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

GENERALE PERFORMANCE

ANGULAR

Redirecting and Wildcard Routes

- A redirect route is a route configuration that redirects a URL to another URL.

- Wildcard routes, on the other hand, are used to match any URL that does not match any other defined routes. You can use wildcard routes to display a default page or a 404 page when a user tries to access an undefined URL. In Angular, you can use the path property with a wildcard value (e.g. **) to configure a wildcard route.

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

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

What are guards?

- In Angular, guards are a feature that can be used to control access to certain routes or functionality in an application. Guards are used to protect the application against unauthorized access or actions by users.
- Guards in Angular can be used to protect routes or functionality based on certain conditions or criteria. For example, a guard can check whether a user is authenticated before allowing them to access a certain route or feature. This is useful for preventing unauthorized access to sensitive parts of an application.
- Guards can also be used for authorization, where different types of users may have different levels of access to certain parts of the application. In this case, guards can check the user's role or permissions to determine whether they are allowed to access a particular route or feature.
- Angular contains different type of guards, for example: CanActivate, CanActivateChild or CanDeactivate

CanActivate

This guard is used to allow or prevent access to a particular route based on some conditions. It can be used to check if a user is authenticated or authorized to access a route.

For example, if you have a route that can only be accessed by an authenticated user, you can use the CanActivate guard to check if the user is authenticated before allowing them to access the route.

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

  canActivate(): boolean {
    if (!this.authService.isAuthenticated()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
}
```

```
{path: 'drivers',
  component: DriverComponent,
  canActivate: [AuthGuard]
}
```

CanActivateChild

This guard is used to protect child routes of a route. It works similar to the CanActivate guard, but it is applied to child routes.

For example, if you have a parent route that has child routes that can only be accessed by an authenticated user, you can use the CanActivateChild guard to check if the user is authenticated before allowing them to access the child routes.

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

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

```
{path: ‘drivers’,
  component: DriverComponent,
  canActivateChild: [AuthGuard],
  children: [
  …
  ]},
```

CanDeactivate

This guard is used to prevent a user from leaving a particular route. It can be used to check if the user has unsaved changes and warn them before leaving the route.

For example, if you have a form that the user has started filling but hasn't submitted yet, you can use the CanDeactivate guard to warn the user before leaving the form.

```
@Injectable({
  providedIn: 'root'
})
export class FormGuard implements CanDeactivate<DriverDetailsComponent> {
  canDeactivate(component: DriverDetailsComponent): boolean {
    if (component.hasUnsavedChanges()) {
      return confirm('Are you sure you want to leave this page? ');
    }
    return true;
  }
}
```

```
{
     path: ‘:id’,
     component: DriverDetailsComponent,
     canDeactivate:[FormGuard]
}
```

Since Angular 14 Functional Guards: CanActivate

In Angular v14.2, functional route guards were introduced as a new way to protect parts of Angular applications.

Functional route guards require just a single function, and can be written in a separate file, or created inline for a route if needed.

```
import { inject } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from './path-to-auth-service';

export function authGuard(): CanActivateFn  {
 return () => {
   const authService: AuthService = inject(AuthService);

   if (!authService.isAuthenticated()) {
     router.navigate(['/login']);
     return false;
   }
   return true;
 };
};
```

```
{path: 'drivers',
 component: DriverComponent,
 canActivate: [authGuard]
}
```

CanDeactivate functional guard

```
export function formGuard(component: DriverDetailsComponent): boolean {
  if (component.hasUnsavedChanges()) {
    return confirm('Are you sure you want to leave this page?');
  }
  return true;
}
```

```
{
     path: ‘:id’,
     component: DriverDetailsComponent,
     canDeactivate:[formGuard]
}
```

What is lazy loading?

- Lazy loading is a technique used in web development to optimize the performance of applications. It involves loading modules and components on-demand, rather than all at once when the application starts. By deferring the loading of certain parts of the application until they are actually needed, the initial load time and memory footprint of the application can be reduced.

- There are several benefits to using lazy loading in an Angular application, including:
  - <u>Faster initial load time:</u> By loading only the necessary components and modules when they are needed, the initial load time of the application can be greatly reduced.
  - <u>Improved memory usage:</u> Lazy loading helps to reduce the memory footprint of the application by only loading the necessary modules and components.
  - <u>Simplified application architecture:</u> By breaking an application into smaller, more manageable chunks, lazy loading can help to simplify the overall architecture of an application.

How to use lazy loading

- In Angular, lazy loading is achieved through the use of the **loadChildren** property in the Route configuration. This property specifies a path to a module that will be loaded lazily when the corresponding route is activated.
- The Root "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.

Preloading strategy (1/3)

In Angular, when setting up lazy-loaded routes, you can specify a **preloadingStrategy** to control how and when the lazy-loaded modules are loaded. Here are the common strategies provided by Angular:

1. NoPreloading (default):

Lazy-loaded modules are only loaded when the user navigates to their routes. No modules are preloaded.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
  // No preloading strategy needed, as it's the default behavior
})
export class AppRoutingModule { }
```

Preloading strategy (2/3)

2. PreloadAllModules:

After the application has loaded, Angular will preload all lazy-loaded modules in the background, so they are ready as soon as the user navigates to their routes.

```
import { PreloadAllModules } from '@angular/router';

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

Preloading strategy (3/3)

3. Custom Preloading Strategy:

You can implement a custom preloading strategy by implementing the PreloadingStrategy interface. This allows you to write complex logic to determine when specific modules should be preloaded, such as based on user roles, connection speed, or other custom metrics.

```
export class CustomPreloadingStrategyService implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      return load();
    } else {
      return of(null);
    }
  }
}
```

```
// app-routing.module.ts
@NgModule({
  imports: [
    RouterModule.forRoot(routes, {
      preloadingStrategy: CustomPreloadingStrategyService
    })
  ],
  exports: [RouterModule],
  providers: [CustomPreloadingStrategyService]
})
export class AppRoutingModule { }
```

- **route**: The route object that could be preloaded.
- **load**: A function that you can call to load the route's module.

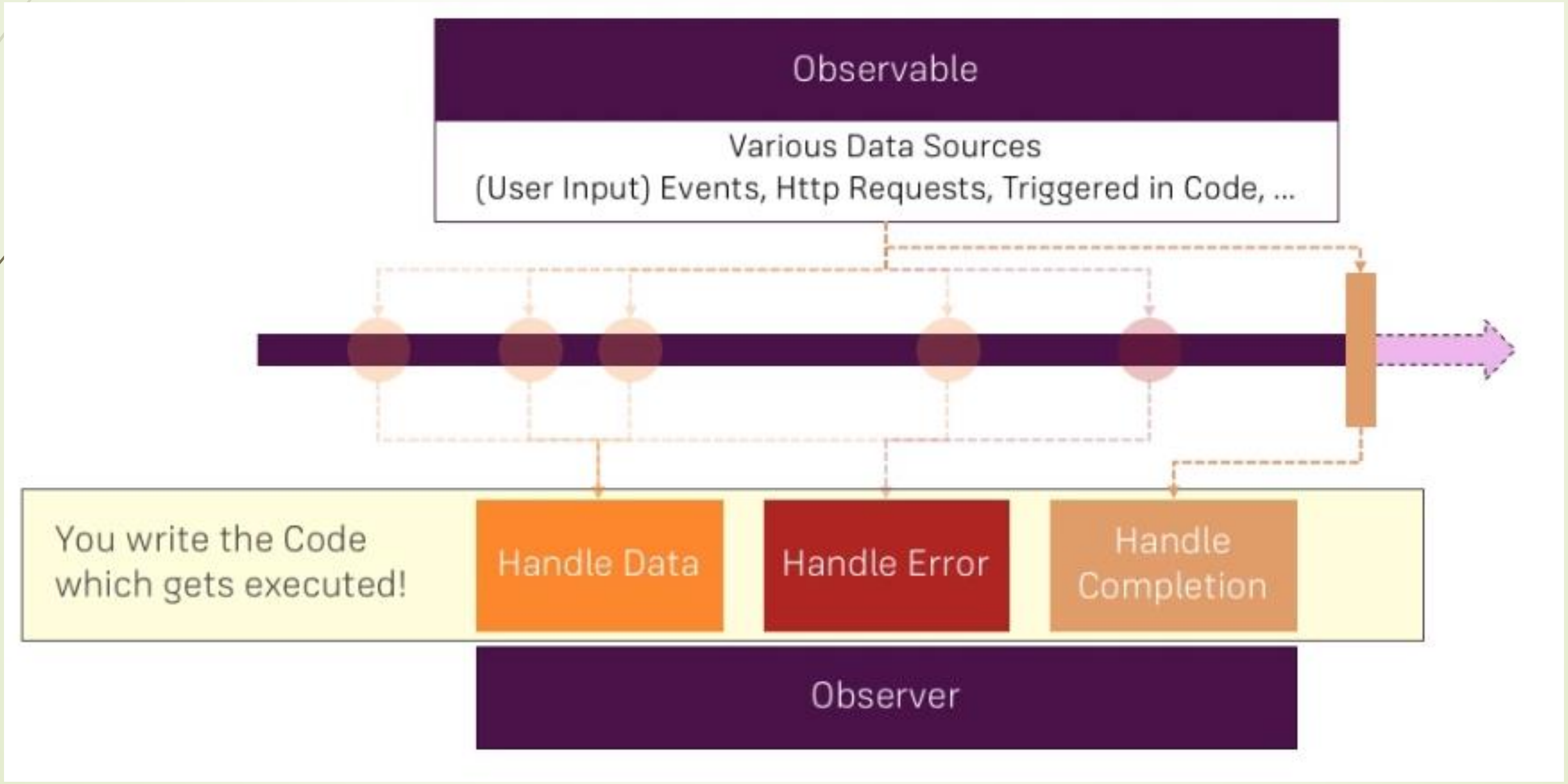
GENERALE PERFORMANCE

Observables

Observables in Angular

- Observables are essentially streams of data that can be observed and manipulated over time. In Angular, observables are used to handle asynchronous events such as HTTP requests, user input, and more.
- It can be thought of as a sequence of events that are generated by a source and sent to a subscriber.
- An observable can emit multiple values over time, and any number of observers can subscribe to the observable to receive these values.
- Some common examples of how observables are used in Angular applications include:
  - Handling HTTP requests: When making a request to an API endpoint, the response is typically asynchronous. You can use the Angular HttpClient service, which returns an observable, to handle these requests and process the response.
  - Handling user input: When a user interacts with a form or other input element, the changes are often emitted as events. You can use observables to handle these events and update the state of your application accordingly.
  - Real-time data: Observables can be used to model real-time data streams, such as stock prices. By subscribing to the stream, you can receive updates as the data changes over time.

Obesrvables


Observable
Various Data Sources
(User Input) Events, Http Requests, Triggered in Code, ...
You write the Code which gets executed!
Handle Data
Handle Error
Handle Completion
Observer

Observers

The Observer interface is an optional interface that can be used to implement custom logic for handling emitted values from an observable. The interface has three methods:

NOTIFICATION TYPE	DESCRIPTION
next(value: T)	Required. A handler for each delivered value. This method is called whenever an observable emits a new value. The value parameter represents the emitted value.
error(error: any)	Optional. A handler for an error notification. This method is called when an observable encounters an error. The error parameter represents the error object.
complete	Optional. This method is called when an observable completes and will not emit any more values.

Subscribing to Observables

- To subscribe to an observable, you can call the subscribe method of the observable instance and provide it with a function to handle emitted values. The provided function is called for each value emitted by the observable. Here is an example:

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

const observable = new Observable((observer) => {
  observer.next('hello');
  observer.next('world');
  observer.complete();
});
```

```
observable.subscribe({
  next: (value) => {
    console.log(value);
  },
  complete: () => {
    console.log('completed');
  },
  error: (error) => {
    console.error(error);
  },
});
```

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

GENERALE PERFORMANCE

ANGULAR

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:

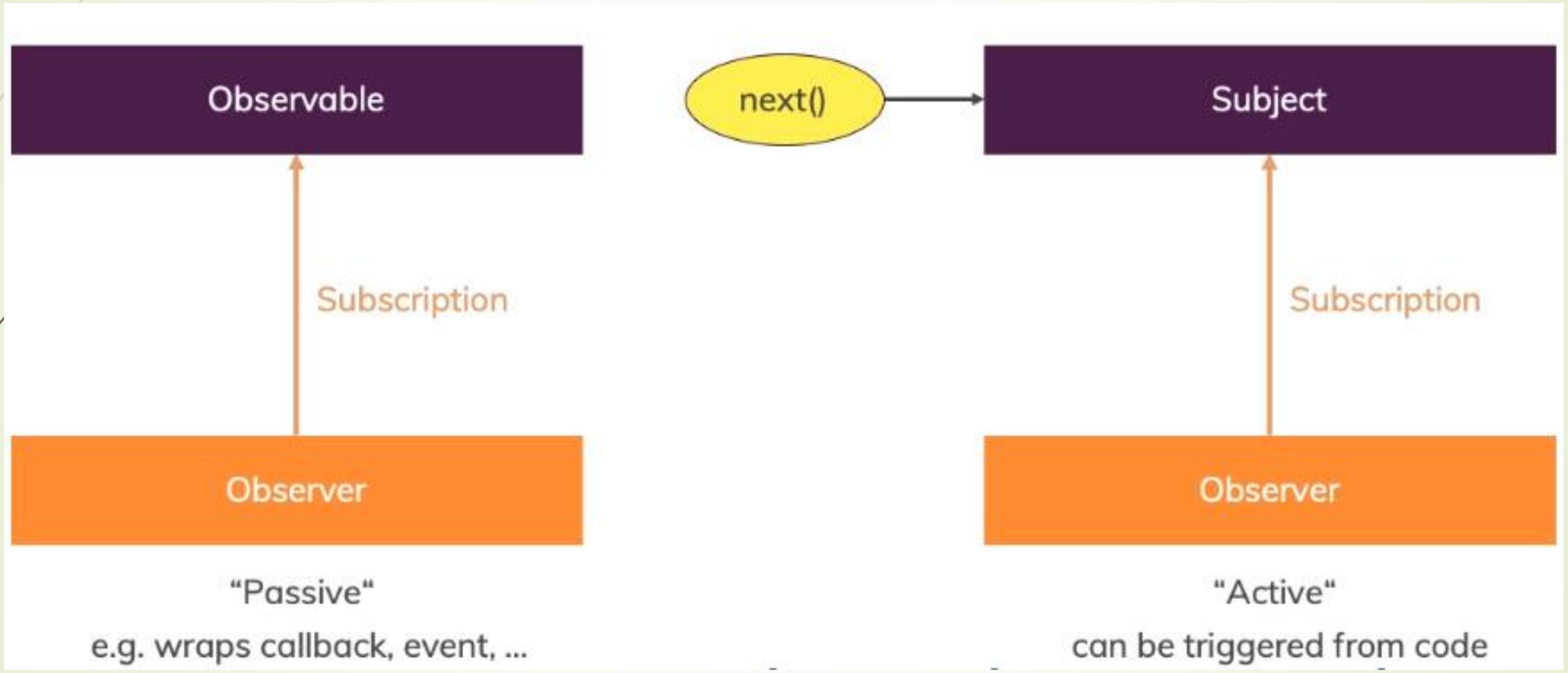
- unsubscribe the "callbacks": next, error and complete;
- destroy the Observable (interrupt the processing carried out by the Observable)
- possibly free the memory because by unregistering the "callbacks" .

Subject

- In RxJS, a Subject is a special type of observable that allows you to push values to it in addition to subscribing to it. This means that you can use a Subject to act as both an observer and an observable.

- Here are some key points about Subject:
  - You can use the **next()** method of a Subject to push new values to it. When you call next(), the Subject emits the value to all of its subscribers.
  - Subject does not have a fixed set of values like Observable. Instead, it is a stream of values that can change over time as new values are pushed to it.
  - Subject can act as both an observable and an observer. You can use Subject to subscribe to other observables, and then emit new values based on the values emitted by those observables.

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.

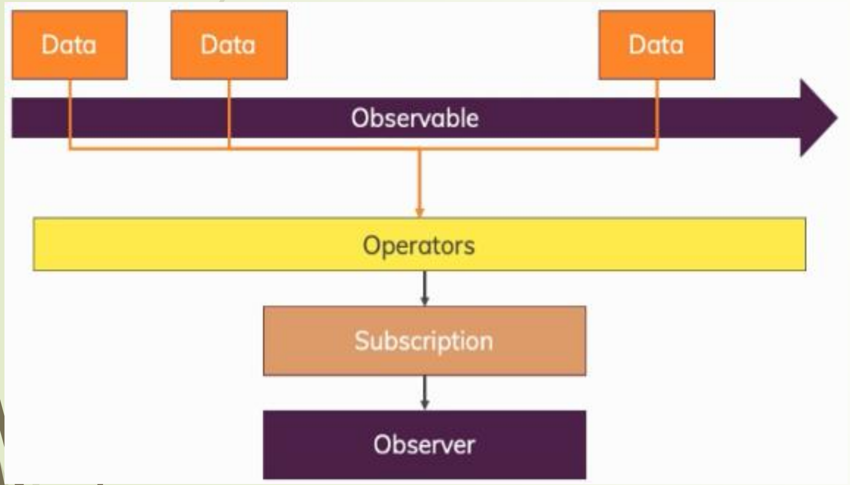
Subject vs Observable

Observables	Subjects
Immutable(values and state cannot be changed. This means that any operations performed on the observable, such as filtering or mapping, will create a new observable instance with the updated values or state.)	Mutable(the values and state of an observable can be changed after it has been created.)
Cannot be manually pushed new values	Can manually push new values
Have a cold execution strategy (i.e., the observable starts running anew for each subscriber)	Have a hot execution strategy (i.e., the subject is already running when a subscriber subscribes)
Used when you want to create a stream of data from an event or data source	Used when you want to create an observable that can be both an observer and an observable

BehaviourSubject

- BehaviorSubject is a variant of Subject that requires an initial value and emits the most recent value to all its subscribers. When a new subscriber subscribes to a BehaviorSubject, it immediately receives the most recent value emitted by the BehaviorSubject. Additionally, any subsequent subscribers also receive the most recent value, just like with Subject.
- The key difference between Subject and BehaviorSubject is that BehaviorSubject always has a current value, whereas Subject does not. If you want to guarantee that subscribers always receive a value, even if they subscribe after the value has been emitted, you should use BehaviorSubject.
- Subject is useful when you only need to emit values, while BehaviorSubject is more appropriate when you need to maintain a current state or want to ensure that new subscribers receive the most recent value.

Operators



- In RxJS, operators are functions that are used to manipulate the emitted values of an Observable. They allow you to transform, filter, or combine streams of data in various ways. There are two types of operators: pipeable operators and creation operators.
- Pipeable operators are functions that can be chained together using the pipe() method. They take an Observable as input and return a new Observable as output. Some examples of pipeable operators include map(), filter().
- Creation operators are functions that create Observables from scratch. Some examples of creation operators include of(), from(), and interval().

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.

GENERALE PERFORMANCE

ANGULAR

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));
```

GENERALE PERFORMANCE

ANGULAR

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);
        });

}
```

Error Handling

- In RxJS, errors can be propagated from an observable to its observer using the error method. This can happen due to various reasons such as network issues or incorrect data formatting.
- To handle errors in observables, we can use the catchError operator, which allows us to catch errors emitted by an observable and then return a new observable or throw a new error. Here is an example:

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

const observable = of('some data');

observable.pipe(
  catchError(error => {
    console.error('Error:', error);
    return of('default data');
  })
).subscribe(data => console.log(data));
```

- In this example, if an error occurs, the catchError operator catches it and returns a new observable with default data. The observer then receives the default data instead of the error.

Retry after an error

- The retry operator is used to retry the observable a certain number of times before throwing an error. Here is an example:

```
import { of } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';

const observable = of('some data');

observable.pipe(
  retry(3), // retry the observable 3 times
  catchError(error => {
    console.error('Error:', error);
    return of('default data');
  })
).subscribe(data => console.log(data));
```

- In this example, the retry operator retries the observable three times before giving up and throwing an error. If an error occurs during any of the retries, the catchError operator catches it and returns a new observable with default data.

GENERALE PERFORMANCE

Forms

Overview of forms in web applications

- Forms are a crucial part of most web applications as they allow users to input and submit data to be processed and stored by the application.
- Forms can be used for a variety of purposes, including user authentication, data entry, search, and more.
- Angular provides two approaches for working with forms: Reactive and Template-driven forms.

Template Driven VS Reactive

	Reactive Forms	Template Driven
Definition	A programmatic approach to building forms in Angular using reactive programming techniques.	A declarative approach to building forms in Angular using template syntax and directives.
Form Creation	Created programmatically in the component class using the FormBuilder service or by instantiating FormControl, FormGroup, and FormArray classes directly	Created in the template using form-specific directives and template syntax
Validation	Validation is done programmatically using the Validators class and reactive programming techniques	Validation is done using built-in Angular directives such as required and minlength
Form Complexity	More suitable for complex forms with dynamic form fields, conditional validation, and asynchronous validation	More suitable for simple forms with static form fields and simple validation

GENERALE PERFORMANCE

Forms: Template Driven

Building a Simple Form(1/2)

- To create a simple form using the template-driven approach, we need to follow these steps:

1. Import the **FormsModule** from the @angular/forms package in the app.module.ts file.
2. In the template, add a form element and its inputs. We can use Angular directives like **ngModel** to bind form inputs to component properties.
3. Add a submit button to the form. We can use the type attribute to specify the button as a submit button.
4. In the component class, add a method to handle form submissions. We can use the onSubmit event to call this method when the form is submitted.

Building a Simple Form(2/2)

```
@Component({
 selector: 'app-simple-form',
 template: `
  <form #simpleForm="ngForm" (ngSubmit)="onSubmit(simpleForm.value)">
   <label for="name">Name</label>
   <input type="text" id="name" name="name" ngModel>

   <label for="email">Email</label>
   <input type="email" id="email" name="email" ngModel>

   <button type="submit">Submit</button>
  </form>
 `,
})
export class SimpleFormComponent {
 onSubmit(formData: any) {
  console.log(formData);
 }
}
```

TD: Registering the Controls

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

```
<label for="name">Name</label>
<input type="text" id="name" name="name" ngModel>
```

- It is important to assign a unique name attribute to each form control. This allows Angular to associate the control with the corresponding ngModel directive, which is responsible for data binding between the control and the component's data model.

Form Validation

- In the Template Driven approach to building forms in Angular, form validation can be achieved using both built-in and custom validation directives.

Built-in Validation Directives

- Angular comes with several built-in validation directives that can be used in template-driven forms, including:
  - required: checks if the field is empty
  - minlength: checks the minimum length of the input value
  - maxlength: checks the maximum length of the input value
  - pattern: checks if the input value matches a specific pattern

```
<input type="text" name="firstName" ngModel required>
```

Custom Validation Directives

- Custom validation directives can also be created in Angular.
- To create a custom validation directive, you need to define a class that implements the Validator interface.

```
import { Directive } from '@angular/core';
import { NG_VALIDATORS, Validator, AbstractControl } from '@angular/forms';

@Directive({
  selector: '[alphabetsOnly]',
  providers: [{ provide: NG_VALIDATORS, useExisting: AlphabetsOnlyDirective, multi: true }]
})
export class AlphabetsOnlyDirective implements Validator {
  validate(control: AbstractControl): { [key: string]: any } | null {
    const alphabetsOnlyRegex = /^[a-zA-Z]+$/;
    const valid = alphabetsOnlyRegex.test(control.value);
    return valid ? null : { alphabetsOnly: { value: control.value } };
  }
}
```

```
<input
          type="text"
          name="firstName"
          ngModel
          alphabetsOnly>
```

Outputting Validation Error Messages

- To display validation errors, you can use the ngIf directive to show and hide error messages based on the validity of the input. Here is an example of showing an error message when the input field is empty:

```
<input type="text" name="firstName" ngModel required>
<div *ngIf="firstName.errors?.required && firstName.touched">First Name is required</div>
```

Form Submission

- In Template Driven Forms, handling form submission involves capturing the data entered by the user and performing any necessary actions, such as sending the data to a server or updating a local database.

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

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

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

Grouping Form Controls in Template Driven Forms

- Template Driven Forms in Angular allow for grouping form controls together. This can be helpful when working with forms that have multiple fields that are related to each other.

```
<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'];
```

GENERALE PERFORMANCE

ANGULAR

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();
```

Forms: Reactive

Creating a Reactive Form (1/2)

- Steps to set up a Reactive Forms approach in Angular:

1. Import the ReactiveFormsModule module from @angular/forms in your app.module.ts file.
2. In your component.ts file, create a FormGroup object to represent your form. This can be done in the constructor method of the component.
3. Create a new FormControl object for each form field that you want to include in your form.
4. In your HTML template, use the formGroup directive to bind the form group object to the form element.
5. Use the **formControlName** directive to bind each form control to its corresponding form field in the HTML template.
6. Add a button to the form in the HTML template, and bind the (click) event to a method in your component that will handle the form submission.

Creating a Reactive Form (2/2)

```
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']
 });
}
```

Syncing HTML and Form

- You can bind the form controls to the template using the formGroup and formControlName directives.
- The formControlName directive is used to bind a single FormControl to an input element. It is added as an attribute to the input element, and its value is set to the name of the FormControl.

```
<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);
}
```

Form Validation in Reactive Forms

- You can also use built-in validation functions from the Validators class, such as required or min and max for numeric values, minLength and maxLength for strings, and pattern for custom validation patterns.

```
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')
});
```

Custom Validators in Reactive Forms

```
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>
```

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>
```

Creating dynamic forms (1/3)

- In Reactive Forms, a Form Array is used to manage an array of Form Controls. It is useful when you want to allow the user to add or remove multiple inputs dynamically. For example, you might have a form where the user can add multiple email addresses or phone numbers.

```
myForm: FormGroup;

constructor(private fb: FormBuilder) {
 this.myForm = this.fb.group({
  emails: this.fb.array([ this.fb.control('', Validators.email) ])
 });
}
```

Creating dynamic forms (2/3)

- Add some needed functions to handle the creation of dynamic arrays

```
get emailControls() {
  return (this.myForm.get('emails') as FormArray).controls;
}

addEmail() {
  const control = this.fb.control('', Validators.email);
  (this.myForm.get('emails') as FormArray).push(control);
}

removeEmail(index: number) {
  (this.myForm.get('emails') as FormArray).removeAt(index);
}
```

Creating dynamic forms (3/3)

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
 <div formArrayName="emails">
  <div *ngFor="let email of emailControls; let i = index">
   <input type="email" [formControlName]="i">
   <button (click)="removeEmail(i)">Remove</button>
  </div>
  <button (click)="addEmail()">Add Email</button>
 </div>
 <button type="submit">Submit</button>
</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.

TypedForm from angular 14

- Typed forms, introduced in Angular 14, enhance the Reactive Forms module by adding strong typing. This feature allows developers to specify the data type of form controls, form groups, and form arrays using TypeScript's generic types.
- The FormControl, FormGroup, and FormArray classes became generic, enabling better type checking and editor support.
- The editor can use these types to provide more accurate suggestions, error checking, and refactoring capabilities.
- The introduction of UntypedFormControl also aids in the gradual migration of existing projects to typed forms, allowing for incremental updates without breaking changes.

```
loginForm: FormGroup<LoginForm> = new FormGroup({
  username: new FormControl<string>(''),
  password: new FormControl<string>('')
});
```

```
loginForm = new UntypedFormGroup({
  username: new UntypedFormControl(''),
  password: new UntypedFormControl('')
});
```

Typed **FormControl** Class

- In Angular 14, the FormControl class was updated to be a generic class. This update allows developers to specify the type of value the form control will work with, using the type parameter <TValue>.

```
const nameControl = new FormControl<string>('');
```

- By specifying <string>, we're indicating that nameControl is intended to work with string values. This brings stronger type checking to form controls.

- **Importantly, if you don't specify a type (<TValue>), the FormControl will function as an untyped control (any type).**

Fourth Day

- HTTP Requests
- Authentication

GENERALE PERFORMANCE

Making http Requests

Anatomy of Http Request

- Http Requests in REST are composed of several components, including URL, Method, Headers, and Body.

Http Verb	The Method defines the type of operation being performed on the resource: POST, GET, …
API Endpoint	The address of the resource being requested: /students/50
Headers	Contains additional information about the request: { "Content-Type": "application/json"}
Body(Post, Put, Patch)	contains the data being sent to the server as part of the request: { firstName: "Ahmed"}

Introduction to the HTTP module in Angular

- The HTTP module in Angular is used to make HTTP requests from your Angular application to a server.
- It provides a simple and straightforward way to perform common HTTP operations such as GET, POST, PUT, DELETE, and others.

Importing and Configuring the HTTP module

- To use the HTTP module in your Angular application, you need to import it in the module where you want to use it.
- To configure the HTTP module, you need to add it to the imports array of your module.

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

Making HTTP Requests with the HTTP module

- To make an HTTP request using the HTTP module, you need to inject the HttpClient service in your component.

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

- You can use the HttpClient.get(), HttpClient.post(), HttpClient.put(), and HttpClient.delete() methods to perform different HTTP operations.
- You can pass the URL and any parameters to these methods to make the request.

Handling HTTP Responses with the HTTP module

- When you make an HTTP request using the HTTP module, you receive a response from the server.
- The response is an Observable that you need to subscribe to in order to handle the response data.
- You can use the subscribe() method on the response to get the data and handle errors.

```
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(
		…
 );
```

Making GET Requests with the HTTP module

- The GET method is a common HTTP request method used to retrieve data from a server.
- It's used in RESTful APIs to request information from a specific resource.
- The GET request is read-only and does not modify the resource.
- Use the http.get method to make the GET request to a specific API endpoint.

- The http.get method returns an Observable, which can be subscribed to using the subscribe method.

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

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

- Within the subscribe method, access the response data using the response.body property.

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

Making POST Requests in Angular with HTTP

- The POST method is used to submit data to a server for further processing.
- Unlike the GET method, which is used to retrieve data from a server, the POST method sends data to the server for storage or processing.
- RESTful APIs use the POST method to create new resources on the server.
- The request body should contain the data that is to be stored on the server, such as a new user registration, new order, etc.

```
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(
    …
  );
```

Making PUT Requests in Angular with HTTP

- The PUT method is used to update a resource on the server, identified by its URL.
- It replaces the entire resource with a new representation sent in the request body.
- Call the put() method on the HttpClient instance and pass in the URL and the data to be updated as arguments

```
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(
                ...
  );
```

Using the DELETE method in Angular with the HTTP module

- The DELETE is used to delete resources on the server.
- This method sends a request to the server indicating that a specific resource should be removed. The URI of the resource is specified in the request.
- To make a DELETE request in Angular, you can use the HttpClient module and its delete() method.

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

```
this.studentService.deleteStudent(student.id).subscribe(…);
```

Error Handling in HTTP Requests

- HTTP requests are an integral part of any web application, and it is crucial to handle errors that may occur during the request-response cycle. Understanding and anticipating potential errors can help ensure a better user experience and minimize the impact of potential issues.

```
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 in HTTP Requests

- The HTTP headers are key-value pairs sent along with an HTTP request or response. They provide additional information about the request or response, such as the format of the body, authentication credentials, or the preferred language.
- In Angular, you can set headers for an HTTP request using the headers property of the HttpRequest object or by using the HttpHeaders class.

```
import { HttpHeaders, HttpClient } from '@angular/common/http';

const headers = new HttpHeaders({
  'Content-Type': 'application/json',
  'Authorization': 'my-auth-token'
});

this.http.get('/api/data', { headers: headers })
  .subscribe(data => {
    console.log(data);
  });
```

Using Query Params in Angular HTTP Requests

- Query params are a way to pass additional information to the server as part of the HTTP request. They are appended to the end of the URL in the form of key-value pairs.
- To set query params in Angular, you can create a new instance of HttpParams and append each key-value pair using the append() method.

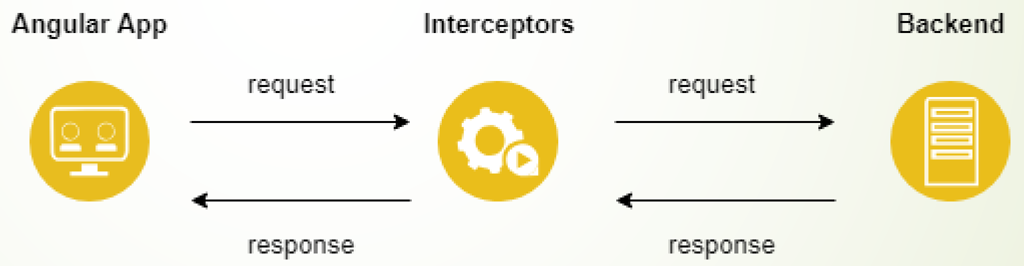
```
let params = new HttpParams();
params = params.append('param1', 'value1');
params = params.append('param2', 'value2');

this.http.get(url, { params: params });
```

- Once the query params are set using the HttpParams class, they are automatically included in the HTTP request as part of the URL.

HTTP Interceptors in Angular

- HTTP interceptors are a way to modify or inspect HTTP requests or responses globally in Angular.
- Common use cases of interceptors include adding headers, modifying request payloads, or catching and handling errors.



Use of Interceptors

- In Angular, HTTP interceptors are created by implementing the HttpInterceptor interface and registering them in the application's module using the HTTP_INTERCEPTORS provider.

```
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);
 }
}
```

- Register the interceptor in the application's module by adding it to the HTTP_INTERCEPTORS provider list.

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

GENERALE PERFORMANCE

ANGULAR

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

Introduction to authentication

- Authentication is the process of verifying the identity of a user or a system before allowing access to protected resources.

- Types of authentication: There are various methods of authentication such as username and password, tokens, certificates, biometrics, etc.

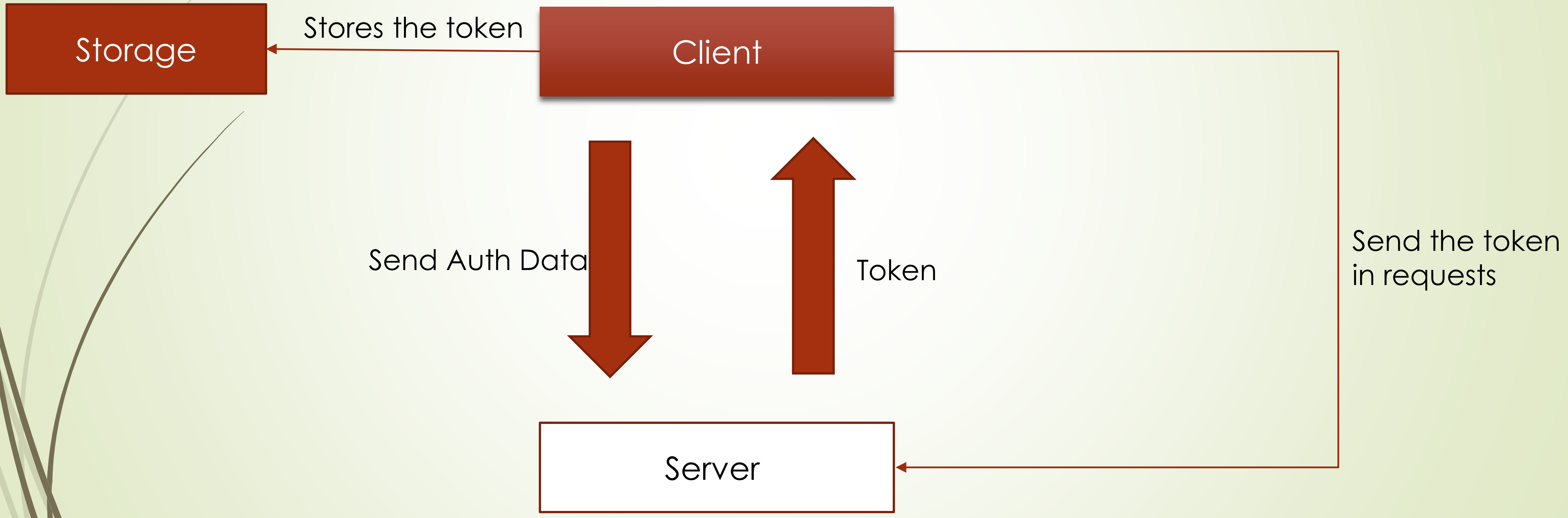
Token-based authentication

- Token-based authentication is a method of authentication where the client exchanges a username and password for a unique token.
- This token is then sent with each subsequent request to the server to authenticate the client. The token is typically generated and signed by the server, and can be used to verify the client's identity, store user information, and enforce authorization.
- The token-based authentication method provides a secure and scalable way to authenticate clients, and is widely used in modern web applications.

GENERALE PERFORMANCE

ANGULAR

How authentication works



Token Storage

- Tokens can be stored in local storage or as a cookie in the client's browser to maintain the user's authentication across different requests.
- Storing tokens in local storage provides a way to persist the authentication information even after the user closes the browser. This can make it convenient for users as they won't need to re-enter their credentials each time they visit the site.

Token-based authentication workflow in Angular(1/3)

- In Angular, the workflow of token-based authentication can be implemented using HTTP interceptors or by creating custom services that handle the token management.
- The token can be stored in local storage or as a cookie, depending on the requirements of the application.
- When a user logs out, the token can be removed from the client's storage to invalidate the user's authorization.

Token-based authentication workflow in Angular(2/3)

- the client first sends a request to the server with the login credentials
- The server verifies the credentials and, if they are correct, generates a token and sends it back to the client.
- The client then stores the token locally, usually in local storage or as a cookie.

```
// Send a POST request to the server with the login credentials
login(credentials) {
  return this.http.post('/api/login', credentials);
}

// Store the token locally after a successful login
this.login({ username: 'user', password: 'pass' })
  .subscribe(response => {
    localStorage.setItem('token', response.token);
  });
```

Token-based authentication workflow in Angular(3/3)

- For subsequent requests to the server, the client includes the token in the request headers.
- The server uses the token to verify the identity of the client and grant access to the requested resources.

```
// Include the token in subsequent requests
this.http.get('/api/protected-resource', {
  headers: {
    Authorization: `Bearer ${localStorage.getItem('token')}`
  }
})
  .subscribe(data => {
    console.log(data);
  });
```

GENERALE PERFORMANCE

THANK YOU
for your
ATTENTION

GENERALE PERFORMANCE

ANNEX

Versions

GENERALE PERFORMANCE

ANGULAR

Introduction & Versions

2

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

GENERALE PERFORMANCE

ANGULAR

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.

GENERALE PERFORMANCE

ANGULAR

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>

GENERALE PERFORMANCE

ANGULAR

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.

GENERALE PERFORMANCE

ANGULAR

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)

GENERALE PERFORMANCE

ANGULAR

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

GENERALE PERFORMANCE

ANGULAR

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

GENERALE PERFORMANCE

ANGULAR

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
- ❖**Strictly Typed Form**
- ❖**Angular CLI Enhancements**
- ❖**Stand-Alone Components**
- ❖Supports TS 4.7

Introduction & Versions

15

- ❖Released on 16 November 2022
- ❖**Better performance**
- ❖**New Angular Material components**
- ❖**New documentation**
- ❖Supports TS 4.8

GENERALE PERFORMANCE

ANGULAR

Introduction & Versions

16

❖Released on 03 May 2023
❖**Better debugging tools**
❖**New Angular Material components**
❖**Updated documentation**

GENERALE PERFORMANCE

ANGULAR

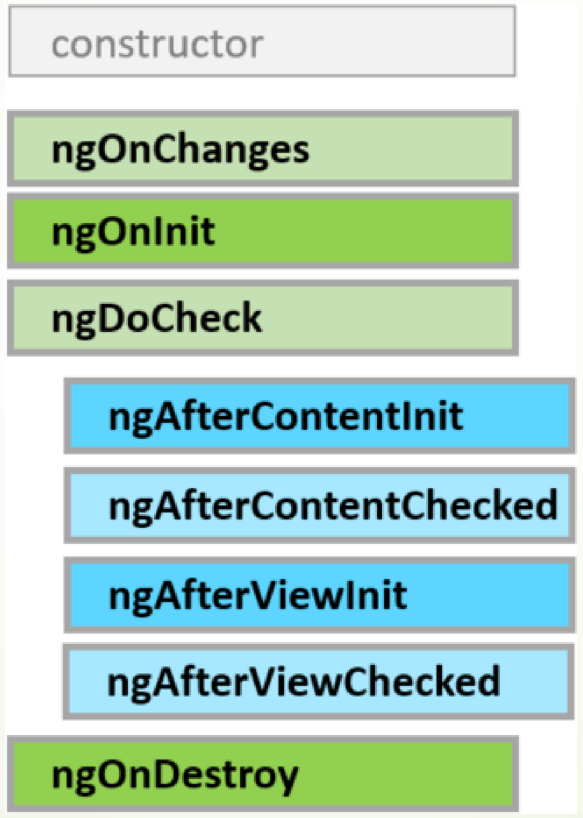
Introduction & Versions

17

- ❖Released on 03 May 2023
- ❖ **New Control Flow Syntax**
- ❖ **Deferrable Views**
- ❖ **Standalone Components as a Default**

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.

GENERALE PERFORMANCE

ANGULAR

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.

GENERALE PERFORMANCE

ANGULAR

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.

GENERALE PERFORMANCE

ANGULAR

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

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.`

GENERALE PERFORMANCE

ANGULAR

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`

GENERALE PERFORMANCE

ANGULAR

Angular Unit Testing

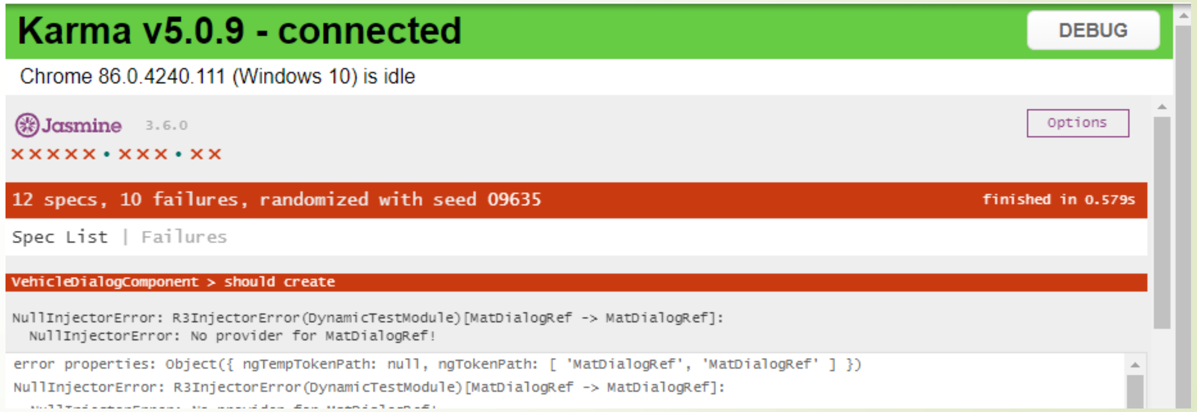
Unit Testing is a type of testing where the focus of the test is to test a particular piece of the application.

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.



```
12 specs, 10 failures, randomized with seed 09635                finished in 0.579s

VehicleDialogComponent > should create

NullInjectorError: R3InjectorError(DynamicTestModule)[MatDialogRef -> MatDialogRef]:
  NullInjectorError: No provider for MatDialogRef!
error properties: Object({ ngTempTokenPath: null, ngTokenPath: [ 'MatDialogRef', 'MatDialogRef' ] })
NullInjectorError: R3InjectorError(DynamicTestModule)[MatDialogRef -> MatDialogRef]:
```

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

GENERALE PERFORMANCE

ANGULAR

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.