Tecnologias e Programação Web - Teste 2

José Mendes 107188

2023/2024

1 Angular Framework

1.1 Typescript

Começa da mesma sintaxe e semântica que o JavaScript. Consegue usar código JavaScript existente, incorporar bibliotecas JavaScript bastante populares e o seu código compilado pode ser chamado a partir de JavaScript.

Typescript compila para código JavaScript simples, legível e **compatível com qualquer navegador, Node.js e qualquer engine que suporte ECMAScript 3** (ou superior).

A melhor parte do Typescript é o uso de **Types** em JavaScript. Mesmo sendo opcional. Desta forma permite práticas JavaScript como:

- **Static Checking** - deteção de erros em tempo de compilação
- **Code Refactoring** - alterar o código sem alterar o comportamento

Os tipos deixam definir interfaces entre componentes de software, dando mais controlo no comportamento de bibliotecas JavaScript. O Typescript também permite **type inference** para verificação estática de código.

1.1.1 Declaração de Variáveis

```
var x = 1; // deprecated because its flaws

let y = 2; // new and strong declaration

const z = 3; // constants

let arr = [10, 20];
let [a, b] = arr; // array destructuring

Let o = { a: 1, b: 'hello', c: 10 }
Let {a, c} = o; // object destructuring
```

1.1.2 Tipos Básicos

```
// boolean
let isfull: boolean = true;

// number
let dec: number = 10.5; // decimal
let hex: number = 0xFF; // hexadecimal
let oct: number = 0o373 // octal
let bin: number = 0b1010 // binary

// string
let name: string = "John";
name = 'Jones';
let s: string = 'Your name is: ${name}';

// array
let list: number[] = [10, 20, 30];
let list: Array<number> = [10, 20, 30];

// tuple
let t: [Boolean, number];
t = [false, 100]; // ok
t = [100, false]; // error

// enum
enum Color {red, green, blue}
let c: Color = Color.green;

// any
let what: any = 100;
what = true; // ok

// void (nothing, the opposite of any)
let nothing: void = null; // or undefined, normally used for return type of a function

// null and undefined
// null is the absence of a value in a variable
// undefined is the absence of definition of a variable

// never
// type of values that never occur
// usually used as function type for never ending functions

// type assertions (cast like)
let avalue: any = "this is a string value";
let slength: number = (<string>avalue).length;
```

1.1.3 Funções

Tal como em JavaScript, as funções podem ser criadas tanto como com nome ou anónimas, e tipadas.

```
function add(x: number, y: number): number {
  return x + y;
  }
let sum = add(1, 2);
// or:
let sum = function(x: number, y: number): number {
  return x + y;
};
```

Definir o tipo das funções:

- Inclui 2 partes, parametros e tipo de retorno
- Na função seguinte, tem de ter 2 parametros do tipo number e retorna um number

```
let sum: (a: number, b: number) => number =
  function(x: number, y: number): number {
    return x + y;
  };
```

Pode ter parametros **default**:

```
function myname(fname: string, lname: 'Burton') {
  return fname + " " + lname;
}
```

Parametros **opcionais**:

```
function myname(fname: string, lname?: string) {
  if (lname)
    return fname + " " + lname;
  else
    return fname;
}
```

Funções **Fat Arrow**:

```
let sum = (a, b) => a + b;
s = sum(10, 20);

// or

let data = [1, 2, 3];
let s = 0;
data.forEach((x) => s += x);
```

1.1.4 Interfaces

Um dos principios base do Typescript é que a verificação de tipo se concentra na forma que os valores têm.

São uma forma poderosa de definir contratos dentro do código, bem como contratos com código exterior.

```
interface Hello {
  mesg: string;
  name: string;
}

function printHello(obj: Hello)
{
  console.log(obj.mesg + " " + obj.name + "!!!");
}

let myObj = {idobj: 10, mesg: "Hello myfriend", name: "John",
           fullname: "John Simons"};
printHello(myObj); // Hello myfriend John!!!
```

1.1.5 Classes

O Typescript permite aos desenvolvedores usar técnicas de programação orientada a objetos, baseada em classes, para programar os seus web scripts.

Compila para código JavaScript que funciona em qualquer browser moderno e plataformas, sem ter de esperar pela próxima versão do JavaScript.

```
class Welcome {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  say() {
    return 'Hello ${this.name}! You're Welcome!!!';
  }
}

let w = new Welcome("Robert");
console.log(w.say());
```

1.1.6 Classes - Herança

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  say() {
    return 'Hello ${this.name}!';
  }
}

class Employee extends Person {
  id: number;
  constructor(name: string, id: number) {
    super(name);
    this.id = id;
  }
  say() {
    return super.say() + ' Your id is: ${this.id}';
  }
}

let e = new Employee("Robert", 100);
console.log(e.say());
```

1.1.7 Classes - Overloading

```
class Animal {
  name: string;
  constructor(n: string){
    this.name = n;
  }
  talk() {
    return this.name + " is talking: ";
  }
}
class Dog extends Animal {
  constructor(name: string){
    super(name);
  }
  talk() {
    return super.talk() + 'Woof! Woof!';
  }
}
let dog = new Dog('Ben');
let ss = dog.talk();
console.dog(ss);
```

1.1.8 Modules

- **Modules** são executados dentro de seu próprio escopo, não no escopo global;
- Variáveis, dunções, classes, etc, declaradas em um módulo não são visíveis fora do módulo;
- **Modules** são declarativos - relações entre módulos são especificadas em termos de **imports** e **exports** ao nível do ficheiro;
- Em Typescript, qualquer ficheiro contendo um top-level **import** ou **export** é considerado um módulo.
- Qualquer ficheiro sem um top-level **import** ou **export** é tratado como um script com conteúdos disponíveis no escopo global.

```
// File: Validation.ts
export interface StringValidator {
  isAcceptable(s: string): boolean;
}
// File: ZipCodeValidator.ts
import { StringValidator } from "./Validation";
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

1.2 Angular

A **Angular Framework** é uma aplicação de desenvolvimento web baseada em linuagem **TypeScript**.

Faz a transição de MVC (Model-View-Controller), usado em AngularJS, para uma abordagem **Components-Based Web Development**.

O **Angular** permite construir aplicações web através de componentes, que são blocos de construção UI, fáceis de testar e reutilizar.

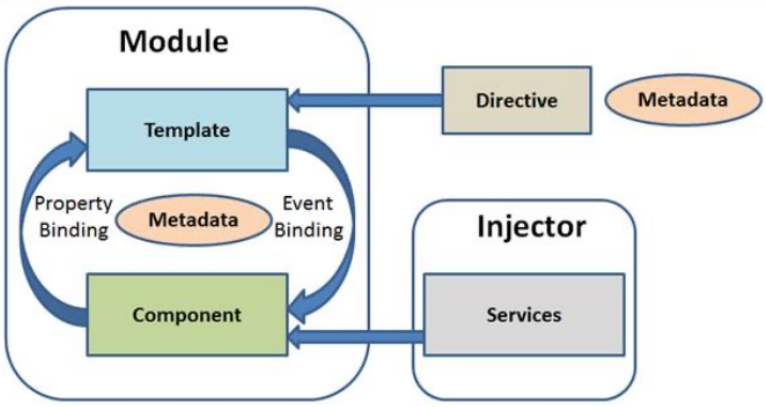
Estes componentes encapsulam bem todo o estilo e a função necessária para uma determinada feature, criando elementos HTML personalizados.

A ideia é construir componentes declarativos, completmente encapsulados. Estes descrevem as próprias views, e podem ser facilmente packaged e distribuidos para outros developers.

A aplicação consiste de um componente **root** que contém um conjunto de componentes para cada elemento UI, screen e route.

Esta árvore de componentes é a base de qualquer aplicação Angular.

1.3 Angular - Arquitetura



1.3.1 Modules

- É um bloco de códifo com o objetivo de realizar uma única tarefa simples.
- Pode ser exportado em forma de classe.
- Aplicações Angular podem ter vários modules, no entanto, têm de ter pelo menos 1.
- Cada aplicação Angular tem um **root module**, que pode ter vários **feature modules**.

Um módulo Angular é uma classe com o decorador **@NgModule**. Decoradores, fornecem uma forma de adicionar anotações e sintaxe meta-programática para declaração de classes e membros.

@NgModule pega num único objeto de metadados e as suas propriedades para descrever o módulo.

- Following are the important properties of @NgModule:
  - exports – it's the subset of declarations which would be used in the component template of other module.
  - imports - imports other modules
  - providers - It is a creator of services. They can be accessible in all the parts of the application.
  - bootstrap - The root module has to set the bootstrap property. It is used to host all other views.
  - declarations - It declare the view class that belong to current module. There are three types of view classes supported by Angular: components, directives, and pipes.

1.4 Angular - Components

Um componente é uma combinação de uma classe, que contém a lógica base para uma página, e um template associado que trata da view.

A lógica da aplicação é escrita dentro da classe que é usada pela view. A classe interage com a View através de métodos e propriedades da sua API.

O Angular cria e dá update dos componentes conforme necessário, e destrói os que já não são usados, enquanto o utilizador navega pela aplicação.

1.5 Angular - Metadados

Os metadados é a forma como o Angular processa uma classe, um método ou uma propriedade. Em Typescript, os metadados são representados por decoradores. Por exemplo, o decorador **@Component** é usado para definir um componente.

1.6 Angular - Template

Um template é a view de um componente e diz ao Angular como renderizar o componente. É semelhante ao HTML normal.

1.7 Angular - Data Binding

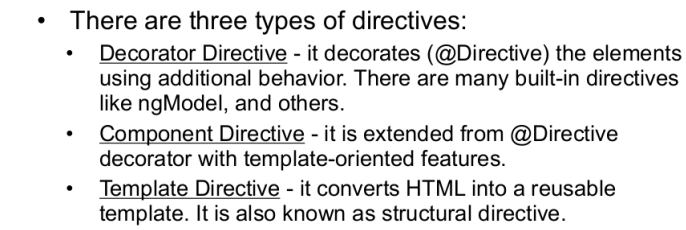
O **Data Binding** é uma feature poderosa do desenvolvimento de software. É a conexão entre a View e a lógica de negócio da aplicação.

Há 4 tipos de **Data Binding** suportados pelo Angular:

- Interpolation - used to display the component value within HTML.
- Property Binding - It passes the property's value of a parent component to its child's property.
- Event Binding – used to fire an event when we click on a component's name, or some change occurs in an input component.
- Two-way Binding – it combines event and property binding in single notation by using ngModel directive, in order to have automatic change between the data model and the view.

1.8 Angular - Diretivas

As diretivas extendem o HTML com atributos. Estes marcadores nos elementos do DOM têm um comportamento especial e dizem ao Angular HTML compiler para anexar.



1.9 Angular - Service

- Um **Service** é uma categoria vasta que engloba qualquer valor, função ou feature que a aplicação necessite.
- Um serviço é tipicamente uma classe com um propósito bem definido.
- O Angular distingue componentes de serviços para aumentar a modularidade e a reusabilidade.
- Tipicos exemplos de serviços são: logging service, data service, message bus, etc.

1.10 Angular - Dependency Injection

- O **Dependency Injection** é um padrão de design de software em quais os objetos são passados como dependências.
- Ajuda-nos a remover as dependências hard-coded e torna as dependências configuráveis.
- Usar **Dependency Injection** pode tornar componentes mais maintainable, reusable, and testable.
- DI está conectado à Angular Framework e usado em toda a aplicação, para fornecer novos componentes com os serviços ou outras coisas que necessitam.

1.11 Angular - Routing

O browser é um modelo familiar para navegação de uma aplicação. Introduzir um URL na barra de endereço para ir para uma página, clicar em links para ir para uma página nova, usar os butões de voltar e de avançar para navegar.

O **Angular Router** é baseado nesse modelo:

- Consegue interpretar um URL como uma instrução para navegar para uma client-generated view.
- Podem se passar parametros opcionais, ajudando a view a mostrar os detalhes da informação.
- Podemos estabelecer links numa página para ajudar o utilizador a navegar, clicando nos links.
- Podemos navegar imperativamente quando o utilizador clica num botão ou seleciona de uma lista.
- E os logs de navegação mantêm o histórico do browser, permitindo usar os botões de voltar e de avançar.

1.11.1 Desenvolvimento

A melhor forma é dando load e configurando o router de forma separada e top-level module que é dedicado ao routing. Por convenção, este módulo fica em **"app.routes.ts"** no diretório **"src/app"**, que exporta **Routes**.

1.11.2 Adicionando Routes

- Adding Routes
  - Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar.
  - A typical Angular Route has two properties:
    - **path**: a string that matches the URL in the browser address bar.
    - **component**: the component that the router should create when navigating to this route.
  - Example – navigate to AuthorsComponent when the URL is something like localhost:4200/authors.
  - Import AuthorsComponent so you can reference it in a Route. Then define an array of routes with a single route to that component.

app.routes.ts

```ts
import { Routes } from '@angular/router';
import {AuthorsComponent} from "./authors/authors.component";

export const routes: Routes = [
  {path: 'authors', component: AuthorsComponent}
];
```

  - If needed, add RouterOutlet to “app.component.ts”, like this:

app.component.ts

```ts
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { AuthorsComponent} from "./authors/authors.component";

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, AuthorsComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title : string  = 'My Books';
}
```

  - If RouterOutlet available is imported, than Angular is ready to route all defined routes.
  - Add its selector to “app.component.html” like this:

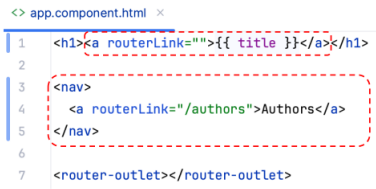
app.component.html

```html
<h1>{{ title }}</h1>

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

  - Now, you can already try the URL “http://localhost:4200/authors”

- You can also add navigations link to navigate between home and authors:



- Using routerLink attribute is a better way than href attribute. Try to use both to see the differences.

2 Django Framework

2.1 RESTful Web Services - Django REST Framework

2.1.1 Serviços Web

Os serviços web são serviços oferecidos por dispositivos eletrónicos para enviar dados para outros dispositivos eletrónicos, usando tecnologias web.

Normalmente, os dados são enviados em formato XML ou JSON.

Um de muitos propósitos de serviços web é para fornecer interoperabilidade e integração dos dados entre sistemas de informação heterogéneos.

2.1.2 REST Web Services

REST significa **REpresentational State Transfer**.

É um modelo de arquitetura para aplicações hypermedia, maioritariamente usado para implementação de serviços web, sendo leve, simples, escalável e sustentável.

Um serviço baseado nesta tecnologia designa-se por **RESTful Service**.

Serviços REST, não dependem de nenhum protocolo específico, mas o mais usado é o **HTTP** para transportar os dados.

Outro tipo de serviços web, são os **SOAP Web Services**, que são baseados no protocolo **SOAP**, que é muito formal, restrito e pesado, pelo que não são tão populares como os RESTful.

2.1.3 Django REST Framework (DRF)

O DRF é uma biblioteca python para criar serviços web RESTful integrados na Framework Django.

Fornece um conjunto de funções importantes para programar os seguintes serviços:

- A possibilidade de publicar uma dada API.
- Politicas de autenticação, usando **OAuth1** ou **OAuth2**.
- Serialização de dados de BDs através do ORM do Django.
- Pode usar views genéricas, caso não seja necessário personalizar as views.
- Atualmente é usado em grandes organizações como **Mozilla**, **Red Hat**, ...