Hoje a definição das nossas classes vivem no escopo global. Além disso, para que o desenvolvedor saiba quais são as classes disponíveis pelo autocomplete do TypeScript ele precisa saber pelo menos parte do nome. Contudo, TypeScript oferece o conceito de namespace. Podemos agrupar classes dentro de um mesmo namepace e acessá-las através dele.

No caso, vamos envolver todas as classes dentro de alurabank/app/views no namespace Views:

namespace Views {

export abstract class View<T> {

protected \_elemento: JQuery;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.html(this.template(model));

}

abstract template(model: T): string;

}

}

Veja que além do namespace, é necessário adicionarmos a instrução export para que a classe esteja disponível.

Agora, vamos alterar MensagemView e NegociacaoView adicionando-os no mesmo namespace. Agora, para estendermos da classe View precisaremos usar a sintaxe Views.View:

namespace Views {

export class MensagemView extends Views.View<string> {

template(model: string): string {

return `<p class="alert alert-info">${model}</p>`;

}

}

}

namespace Views {

export class NegociacoesView extends Views.View<Negociacoes> {

template(model: Negociacoes): string {

return `

<table class="table table-hover table-bordered">

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

${model.paraArray().map(negociacao =>

`

<tr>

<td>${negociacao.data.getDate()}/${negociacao.data.getMonth() +1}/${negociacao.data.getFullYear()}</td>

<td>${negociacao.quantidade}</td>

<td>${negociacao.valor}</td>

<td>${negociacao.volume}</td>

<tr>

`

).join('')}

</tbody>

<tfoot>

</tfoot>

</table>

`;

}

}

}

Agora, em NegociacaoController:

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new Views.NegociacoesView('#negociacoesView');

private \_mensagemView = new Views.MensagemView('#mensagemView');

// código posterior omitido

Excelente, mas essa solução não resolve o problema de termos que nos preocupar com a ordem de importação dos scripts em index.html, um dos tendões de Aquiles do mundo JavaScript. É por isso que o TypeScript aceita também o sistema de módulos do ES2015. Será ele que utilizaremos, em vez do sistema de namespace.

A sintaxe de módulos do ES2015 considera cada script um módulo e através das instruções import e export importamos e exportamos artefatos respectivamente.

Adequando todos os arquivos para o sistema de módulos do ES2015:

export abstract class View<T> {

protected \_elemento: JQuery;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.html(this.template(model));

}

abstract template(model: T): string;

}

import { View } from './View';

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

export class NegociacoesView extends View<Negociacoes> {

template(model: Negociacoes): string {

return `

<table class="table table-hover table-bordered">

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

${model.paraArray().map(negociacao =>

`

<tr>

<td>${negociacao.data.getDate()}/${negociacao.data.getMonth() +1}/${negociacao.data.getFullYear()}</td>

<td>${negociacao.quantidade}</td>

<td>${negociacao.valor}</td>

<td>${negociacao.volume}</td>

<tr>

`

).join('')}

</tbody>

<tfoot>

</tfoot>

</table>

`;

}

}

import { View } from './View';

export class MensagemView extends View<string> {

template(model: string): string {

return `<p class="alert alert-info">${model}</p>`;

}

}

import { Negociacao } from './Negociacao';

export class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return [].concat(this.\_negociacoes);

}

}

export class Negociacao {

constructor(private \_data: Date, private \_quantidade: number, private \_valor: number) {}

get data() {

return this.\_data;

}

get quantidade() {

return this.\_quantidade;

}

get valor() {

return this.\_valor;

}

get volume() {

return this.\_quantidade \* this.\_valor;

}

}

import { NegociacoesView } from '../views/NegociacoesView';

import { MensagemView } from '../views/MensagemView';

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

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

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new NegociacoesView('#negociacoesView');

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.val().replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.val()),

parseFloat(this.\_inputValor.val())

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

}

}

import { NegociacaoController } from './controllers/NegociacaoController';

const controller = new NegociacaoController();

$('.form').submit(controller.adiciona.bind(controller));

Isso ainda não é suficiente, precisamos utilizar um loader, uma biblioteca que seja capaz de carregar o módulo app.js e a partir deles carregar todos os demais módulos.

Vamos alterar o arquivo tsconfig.json e indicar para o TypeScript que ele deve usar o sistema de módulos do System.js:

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js",

"noEmitOnError": true,

"noImplicitAny": true,

"removeComments": true,

"module": "system"

},

"include": [

"app/ts/\*\*/\*"

]

}

Agora, quando nossos arquivos são compilados, eles possuem essa estranha estrutura:

System.register(["../views/NegociacoesView", "../views/MensagemView", "../models/Negociacoes", "../models/Negociacao"], function (exports\_1, context\_1) {

"use strict";

var \_\_moduleName = context\_1 && context\_1.id;

var NegociacoesView\_1, MensagemView\_1, Negociacoes\_1, Negociacao\_1, NegociacaoController;

return {

// código posterior omitido

Por fim, vamos importar o loader utilizá-lo para carregar js/app/js. É a partir dele que os demais serão carregados.

<div id="negociacoesView"></div>

<script src="lib/jquery.min.js"></script>

<script src="lib/system.js"></script>

<script>

System.defaultJSExtensions = true;

System.import('js/app.js').catch(err => console.error(err));

</script>

Contudo, isso não é suficiente. Loaders usam XMLHttpRequest, ou seja, realizam requisições Ajax para baixar os módulos e para isso precisamos de um servidor que disponibiliza nossa aplicação para o browser.

Como vimos, precisamos servir nossa aplicação através de um servidor web. Utilizaremos o [lite-server](https://github.com/johnpapa/lite-server). Além dele servir a pasta alurabanl/app para nós, ele ainda suporta livereloading através do BrowserSync que traz embutido. Isso é perfeito, pois toda vez que os arquivos .ts forem modificados e os arquivos .js gerados nosso navegador automaticamente será recarregado.

Dentro da pasta alurabank, vamos instalar o lite-server`:

npm install lite-server@2.3.0 --save-dev

Agora, em alurabank/package.json vamos adicionar a chamada do servidor através do script "server":

{

"name": "alurabank",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"compile": "tsc",

"start": "tsc -w",

"server": "lite-server --baseDir=app"

},

"author": "",

"license": "ISC",

"devDependencies": {

"@types/jquery": "^2.0.42",

"lite-server": "^2.3.0",

"typescript": "^2.3.2"

}

}

Quando executamos no terminal npm run server nosso servidor subirá e podemos acessar nossa aplicação através de localhost:3000. No entanto, é necessário abrir outro terminal e dentro de alurabank executar o comando npm start para que o compilador em tempo real do TypeScript continue rodando.

No entanto, abrir dois termais não é nada prática. Que tal se pudermos executar os dois processos paralelamente?

**Rodando scripts paralelamente com o módulo concurrently**

Vamos instalar o módulo [concurrently](https://www.npmjs.com/package/concurrently). Ele nos permitirá rodar os dois scripts que criamos em paralelo nas plataformas Windows, MAC e Linux.

Dentro da pasta alurabank vamos executar o comando:

npm install concurrently@3.4.0 --save-dev

Agora, vamos renomear o script "start" para "watch" e adicionar novamente o script "start" que chamará o módulo concurrently:

{

"name": "alurabank",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"compile": "tsc",

"watch": "tsc -w",

"server": "lite-server --baseDir=app",

"start": "concurrently \"npm run watch\" \"npm run server\""

},

"author": "",

"license": "ISC",

"devDependencies": {

"@types/jquery": "^2.0.42",

"concurrently": "^3.4.0",

"lite-server": "^2.3.0",

"typescript": "^2.3.2"

}

}

Excelente, agora, no terminal, basta executarmos o comando npm start para termos os dois serviços rodando em paralelo em um único terminal.

npm start

Podemos simplificar bastante a importação dos módulos através da estratégia Barrel(barril). Nela, um módulo importa e exporta todos os artefatos de uma pasta permitindo assim que apenas o barril seja importado na cláusula from.

Vejamos:

// app/ts/views/index.ts

export \* from './View';

export \* from './MensagemView';

export \* from './NegociacoesView';

// app/ts/models/index.ts

export \* from './Negociacao';

export \* from './Negociacoes';

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

// código posterior omitido

import { View } from './View';

import { Negociacoes } from '../models/index';

export class NegociacoesView extends View<Negociacoes> {

// código posterior omitido

O loader utilizado nos obriga a escrever index no final da importação do módulo. Projetos em Angular e Ionic que fazem uso do Webpack podem omiti-lo na importação. O importante é sabermos desta forma prática de organizar nossos módulos.

Nossa classe Negociacao possui atributos privados, pois não faz sentido alterarmos uma instância de Negociacao depois de criada. Vamos recapitular:

export class Negociacao {

constructor(private \_data: Date, private \_quantidade: number, private \_valor: number) {}

get data() {

return this.\_data;

}

get quantidade() {

return this.\_quantidade;

}

get valor() {

return this.\_valor;

}

get volume() {

return this.\_quantidade \* this.\_valor;

}

}

Contudo, o TypeScript possui um atalho para declaração de propriedades somente leitura. Para isso, basta usarmos o modificador readonly.

export class Negociacao {

constructor(readonly data: Date, readonly quantidade: number, readonly valor: number) {}

get volume() {

return this.quantidade \* this.valor;

}

}

Veja que não foi necessário criar os getters para que pudéssemos acessar as propriedades que antes eram privadas. Agora, qualquer atribuição feita às propriedades resultarão em erro de compilação.

Vamos realizar uma pequena alteração na classe View. Hoje, interpretamos nosso template sem nos preocuparmos em escapá-lo. Nada impede de alguém colocar a tag script dentro do nosso template. Para evitar que isso aconteça, vamos implementar uma simples lógica de escape que remove a tag script e todo seu conteúdo dos templates das nossas views, mas deixaremos que o desenvolvedor escolha se deseja ou não realizar essa operação:

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escapar: boolean;

constructor(seletor: string, escapar: boolean) {

this.\_elemento = $(seletor);

this.\_escapar = escapar;

}

update(model: T) {

let template = this.template(model);

if(this.\_escapar)

template = template.replace(/<script>[\s\S]\*?<\/script>/, '');

this.\_elemento.html(template);

}

abstract template(model: T): string;

}

No construtor() de View recebemos um novo parâmetro do tipo boolean que guardamos na propriedade da classe escape. Consultamos a propriedade no método update. Perfeito, mas isso fará com que NegociacaoController deixe de compilar, pois somos obrigados a passar o valor true oufalse:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

// erro de compilação

private \_negociacoesView = new NegociacoesView('#negociacoesView');

// erro de compilação

private \_mensagemView = new MensagemView('#mensagemView');

Não queremos obrigar aos desenvolvedores terem que passar false quando não desejarem usar essa funcionalidade, queremos simplesmente que o parâmetro seja ignorado.

Podemos conseguir isso indicando que o parâmetro escapar é opcional, adicionando ? ao final de sua declaração:

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escape: boolean;

// tornando o parâmetro opcional!

constructor(seletor: string, escapar?: boolean) {

this.\_elemento = $(seletor);

this.\_escapar = escapar;

}

// código posterior omitido

Excelente! Nosso código continua compilando. Um ponto importante é que o valor de escapar quando não for passado, será undefined. Os tipos null e undefined podem ser atribuídos para outros tipos como boolean.

Muitas vezes, atribuímos null e undefined à variáveis para realizarmos alguma espécie de controle. Mas esses tipos podem causar problemas em runtime em nosso código se não tivermos cuidado com eles:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new NegociacoesView('#negociacoesView');

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

let nome: string = '';

nome = null; // algo permitido

}

Contudo, o TypeScript possui o modo strickNullChecks. Neste modo, null e undefined não fazem parte do domínio dos tipos e só podem ser atribuídos a eles mesmos. Com a exceção de undefined, que pode ser atribuído a void. Isso pode ser interessante para evitarmos valores nulos e indefinidos em nosso projeto.

Vamos ativá-lo em tsconfig.json:

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js",

"noEmitOnError": true,

"noImplicitAny": true,

"removeComments": true,

"module": "system",

"strictNullChecks": true

},

"include": [

"app/ts/\*\*/\*"

]

}

Agora, nosso código deixará de compilar.

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new NegociacoesView('#negociacoesView');

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

let nome: string = '';

// erro de compilação

nome = null;

}

// código anterior omitido

Type 'null' is not assignable to type 'string'.

Vamos remover o teste que fizemos em nosso código e voltar nossa atenção para o arquivo Views.ts, que deixou de compilar após a ativação desse novo tipo de checagem:

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escape: boolean;

constructor(seletor: string, escapar?: boolean) {

console.log(escape);

this.\_elemento = $(seletor);

// erro de compilação

this.\_escapar = escapar;

}

// código posterior omitido

Type 'boolean | undefined' is not assignable to type 'boolean'.

Type 'undefined' is not assignable to type 'boolean'.

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escape: boolean;

// tirou o tipo opcional que recebia undefined caso não fosse passado para um valor padrão, no caso false

constructor(seletor: string, escapar: boolean = false) {

console.log(escape);

this.\_elemento = $(seletor);

this.\_escapar = escapar;

}

Mas ainda há um problema com outra classe, a classe Negociacoes:

// app/ts/models/Negociacoes.ts

// código anterior omitido

paraArray(): Negociacao[] {

// erro de compilação

return [].concat(this.\_negociacoes);

}

// código posterior omitido

argument of type 'Negociacao[]' is not assignable to parameter of type 'never[]'.

Type 'Negociacao' is not assignable to type 'never'.

Ele é resolvido indicando o tipo do [] com a sintaxe as:

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

O compilador do TypeScript possui uma série de configurações e uma delas é a strictNullChecks. Neste modo, null e undefined não fazem parte do domínio dos tipos e só podem ser atribuídos a eles mesmos. Com a exceção de undefined que pode ser atribuído a void. Isso pode ser interessante para evitarmos valores nulos e indefinidos em nosso projeto.

Que tal vermos mais um exemplo no qual ele pode nos ajudar a detectar erros em nosso código?

Vamos buscar em um HTML hipotético um elemento com ID cartao\_1e a partir dele acessar seu elemento pai através de parentNode:

const elCartao: HTMLDivElement = <HTMLDivElement> document.querySelector('#cartao\_1');

let elPai = elCartao.parentElement;

Excelente. Mas o mesmo código não compilará com strictNullChecks com a seguinte alteração:

const elCartao: HTMLDivElement = <HTMLDivElement> document.querySelector('#cartao\_1');

let elPaiDoPai = elCartao.parentElement.parentElement; // [ts] Object is possibly 'null'

Veja que o aviso do compilador TypeScript faz todo sentido, porque pode ser que o elemento pai do pai do elemento elCartao não exista e, se não existir, o valor de parentNode será null.

Dessa forma, o programador terá que lidar antecipadamente com essa situação em seu código:

const elCartao: HTMLDivElement = <HTMLDivElement> document.querySelector('#cartao\_1');

let elPaiDoPai;

if(elCartao.parentElement) {

elPaiDoPai = elCartao.parentElement.parentElement;

}

console.log(elPaiDoPai);

Temos a seguinte função:

function minhaFuncao(flag: boolean) {

let valor = null;

if(flag) return null;

return true;

}

let x = minhaFuncao(false);

O código compila, pois minhaFuncao pode retornar null ou um boolean dependendo do valor do parâmetro flag passado. O TypeScript também consegue inferir os dois tipos retornados pela função.

Mas se quisermos tipar o retorno de minhaFuncao como boolean? Teremos um erro de compilação:

function minhaFuncao(flag: boolean): boolean {

let valor = null;

// erro aqui!

// Type 'null' is not assignable to type 'boolean'.

if(flag) return null;

return true;

}

let x = minhaFuncao(false);

Com strictNullChecks ativado não podemos retornar null para uma função que explicitamente indicamos que retorna boolean. Se a opção do compilador não estivesse ativa, funcionaria pois null pode ser atribuído a qualquer tipo do TypeScript. Mas será que conseguimos fazer esse código compilar sem termos que desativar esta strictNullChecks no compilador?

Podemos indicar que a função pode devolver mais de um tipo, no caso ela devolverá boolean ou null:

// deixarmos explícitos que a função pode retornar boolean ou null

function minhaFuncao(flag: boolean): boolean | null{

let valor = null;

if(flag) return null;

return true;

}

let x = minhaFuncao(false);

Agora, como explicitamos que seu retorno pode ser também null, nosso código passará pelo strictNullChecks. Curiosamente, linguagens como a Golang permitem uma função ou método ter mais de um tipo de retorno.

TypeScript possui um tipo curioso, o tipo never. Este tipo é aplicável à métodos ou funções que por algum motivo, planejado ou não, podem não terminar sua execução de seu bloco.

Exemplos clássicos são os de métodos que caem em um loop infinito ou de métodos que **sempre** retornam exceções. Exceções fazem com que o método não execute até o fim.

Não confundir o tipo never com o tipo void. O segundo, apesar de indicar que a função ou método nada retorna, indica que a função ou método executará até o fim, mesmo que não retorne nada.

Geralmente não usamos esse tipo em nosso código, mas ele pode aparecer como aviso do compilador. Quando aparecer, você já saberá que a execução do seu método nunca chegará até o fim, sendo um forte indicativo de um bug em seu código.

Vamos alterar nosso código para não permitir que negociações feitas no sábado ou no domingo sejam cadastradas.

// app-src/controllers/NegociacaoController.ts

adiciona(event: Event) {

event.preventDefault();

let data = new Date(this.\_inputData.val().replace(/-/g, ','));

if(data.getDay() == 0 || data.getDay() == 6) {

this.\_mensagemView.update('Somente negociações em dias úteis, por favor!');

return

}

const negociacao = new Negociacao(

data,

parseInt(this.\_inputQuantidade.val()),

parseFloat(this.\_inputValor.val())

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

}

No entanto, ver em nosso código algo como ==0 e ==6 não nos diz muita coisa. Poderíamos até ter criado uma variável como sabado ou domingo, contudo o TypeScript possui um recurso mais elegante para o problema que estamos vendo, as enumerations.

Dentro do mesmo arquivo que definimos nosso controller, vamos criar a enum DiaDaSemana:

enum DiaDaSemana {

Domingo,

Segunda,

Terca,

Quarta,

Quinta,

Sexta,

Sabado,

}

Por padrão, o valor de cada um começa de 0 e vai até 6, pois Sabado é o sétimo elemento.

Usando em nosso código temos:

adiciona(event: Event) {

event.preventDefault();

let data = new Date(this.\_inputData.val().replace(/-/g, ','));

if(data.getDay() == DiaDaSemana.Sabado || data.getDay() == DiaDaSemana.Domingo) {

this.\_mensagemView.update('Somente negociações em dias úteis, por favor!');

return

}

const negociacao = new Negociacao(

data,

parseInt(this.\_inputQuantidade.val()),

parseFloat(this.\_inputValor.val())

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

}

Podemos deixar o nosso código ainda mais claro, criando um método privado em nosso controller, chamado ehDiaUtil, para nos retornar true ou false se a data na negociação é um dia útil:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacao, Negociacoes } from '../models/index';

export class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new NegociacoesView('#negociacoesView');

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

}

adiciona(event: Event) {

event.preventDefault();

let data = new Date(this.\_inputData.val().replace(/-/g, ','));

if(!this.\_ehDiaUtil(data)) {

this.\_mensagemView.update('Somente negociações em dias úteis, por favor!');

return

}

const negociacao = new Negociacao(

data,

parseInt(this.\_inputQuantidade.val()),

parseFloat(this.\_inputValor.val())

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

}

private \_ehDiaUtil(data: Date) {

return data.getDay() != DiaDaSemana.Sabado && data.getDay() != DiaDaSemana.Domingo;

}

}

enum DiaDaSemana {

Domingo,

Segunda,

Terca,

Quarta,

Quinta,

Sexta,

Sabado

}

Excelente, melhoramos um pouco mais a legibilidade do nosso código.

Vejamos o seguinte exemplo de enum:

enum Tipo {

ESPECIAL,

PADRAO

}

Sabemos que o valor de Tipo.ESPECIAL e Tipo.PADRAO serão 0 e 1 respectivamente.

Agora, em nosso código fazemos:

let tipo: Tipo = Tipo.ESPECIAL;

É uma sintaxe totalmente válida. E agora?

let tipo: Tipo = 4;

O código anterior compila? Compila! Se você vem de outra linguagem como Java ou C#, este código não compilaria, porque só poderíamos utilizar os tipos Tipo.ESPECIAL e Tipo.PADRAO.

A questão é que a enum tem como subtipo number, por isso pode receber qualquer outro número que não esteja dentro do escopo da Enum.

Há até uma issue sobre esta situação

<https://github.com/Microsoft/TypeScript/issues/11772>

Não é bem um erro, mas um comportamento esperado de como o TypeScript lida com Enum.

Criamos uma solução de renderização da view, que sempre renderiza do zero a view. Por exemplo, a cada inclusão de uma nova negociação em nosso modelo Negociacoes, nós renderizamos a tabela do zero toda vez. Para o escopo da nossa aplicação, isso não nos trará problema algum, mas será que em uma aplicação maior a solução imperativa seria mais indicada, ou quem sabe a utilização de algum framework que seja especializado nisso?

Para que possamos responder a essa pergunta, é necessário realizar um teste de performance. Talvez, o mais simples de todos seja subtrair o tempo final da chamada do tempo do tempo inicial, para termos uma ideia em segundos de quanto o método demora para ser executado.

A própria especificação do JavaScript disponibiliza o objeto global performance para isso. Não precisamos nos preocupar em baixar seu arquivo tds, porque o TypeScript já traz em sua instalação todos os tds para a especificação JavaScript.

Vamos alterar o método adiciona() de NegociacaoController e realizar o teste:

adiciona(event: Event) {

const t1 = performance.now();

// código omitido

const t2 = performance.now();

console.log(`Tempo de execução do método adiciona(): ${(t2 - t1)/1000} segundos`);

}

Podemos ter um teste mais preciso, usando a mesma estratégia no método update(), da nossa classe View.

abstract class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

const t1 = performance.now();

this.\_elemento.innerHTML = this.template(model);

const t2 = performance.now();

console.log(`Tempo de execução do método adiciona(): ${(t2 - t1)/1000} segundos`);

}

abstract template(model: T): string;

}

O problema é que começamos a ter um código espalhado pela nossa aplicação. Se quisermos testar outros métodos, precisaremos fazer a mesma coisa. Por mais que o código seja isolado em uma função, teremos que lembrar de chamar a função no início e no final do método. Será que há uma solução para isso? Queremos poder isolar o código que teste a performance em um único lugar.

Podemos isolar a lógica do nosso teste de performance em um único lugar e aplicá-lo nos métodos que temos interesse sem que tenhamos que modificar sua implementação. Para tal, precisamos ativar em nosso compilador TypeScript a configuração experimentalDecorators. Quando true, permite utilizar decorators, estrutura que atenderá nossa finalidade.

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js",

"noEmitOnError": true,

"noImplicitAny": true,

"removeComments": true,

"module": "system",

"strictNullChecks": true,

"experimentalDecorators": true

},

"include": [

"app/ts/\*\*/\*"

]

}

Como criar um decorator? Vamos criar o arquivo app/ts/helpers/decorators/logarTempoDeExecucao.ts e nele exportamos uma função de mesmo nome:

export function logarTempoDeExecucao() {

}

Essa função indica o nome do nosso decorator, mas sua implementação deve estar na função retornada por logarTempoDeExecucao:

export function logarTempoDeExecucao() {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

}

}

A função retornada não recebe três parâmetros por acaso. O primeiro target é aquele que possui uma referência para o elemento cujo método foi decorado por logarTempoDeExecucao. O segundo parâmetro é uma string que nos retorna o nome do método decorado. Por fim, o descriptor nos dará acesso ao método que desejamos modificar sua execução, através de descriptor.value.

A ideia é a seguinte. Vamos guardar uma referência para o método original antes de substituí-lo com nosso código. É importante que no final o descriptor seja retornado, com as modificações que faremos:

export function logarTempoDeExecucao() {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

// aqui vamos substituir descriptor.value pela lógica do nosso decorator

return descriptor;

}

}

O valor de descriptor.value será function(...args: any[]). Isso se dá dessa forma, porque o método que estamos sobrescrevendo pode receber zero, um ou mais parâmetros de tipos que desconhecemos. Usamos ... para indicar um REST PARAMETER, algo que não é exclusivo do TypeScript, mas do ES2015:

export function logarTempoDeExecucao() {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

descriptor.value = function(...args: any[]) {

const retorno = metodoOriginal.apply(this, args);

return retorno;

}

return descriptor;

}

}

Fazemos metodoOriginal.apply(this, args) para invocar o método original, capturar seu resultado, caso exista e retorná-lo. Ainda não há a nossa lógica do teste de performance. Por enquanto vamos deixar assim. Não deve ocorrer nenhum erro e o comportamento da nossa aplicação deve continuar o mesmo. Mas para que possamos utilizá-lo, vamos exportá-lo através de um barril e importá-lo em View.

Primeiro, criando o barril

// app/ts/helpers/decorators/index.ts

export \* from './logarTempoDeExecucao';

Agora, importando e utilizando em View:

import { logarTempoDeExecucao } from '../helpers/decorators/index';

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escapar: boolean;

constructor(seletor: string, escapar: boolean = false) {

this.\_elemento = $(seletor);

this.\_escapar = escapar;

}

@logarTempoDeExecucao()

update(model: T) {

let template = this.template(model);

if(this.\_escapar)

template = template.replace(/<script>[\s\S]\*?<\/script>/g, '');

this.\_elemento.html(template);

}

abstract template(model: T): string;

}

Usamos decorator através de um @, seguido do nome do decorator, abrindo e fechando parênteses no final, justo, porque um decorator nada mais é do que uma função.

Recarregando nossa aplicação tudo continua funcionando. Agora precisamos escrever a lógica do teste de performance em nosso decorator.

Agora, só precisamos guardar o tempo antes da chamada do método original e logo depois da sua chamada, para no fim, realizarmos o cálculo do tempo gasto. Inclusive, vamos exibir no console os parâmetros recebidos pelo método, inclusive seu retorno:

export function logarTempoDeExecucao() {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

descriptor.value = function(...args: any[]) {

console.log('-----------------------')

console.log(`Parâmetros do método ${propertyKey}: ${JSON.stringify(args)}`);

const t1 = performance.now();

const resultado = metodoOriginal.apply(this, args);

console.log(`Resultado do método: ${JSON.stringify(resultado)}` )

const t2 = performance.now();

console.log(`${propertyKey} demorou ${t2 - t1} ms`);

console.log('-----------------------')

return resultado;

}

return descriptor;

}

}

Vamos analisar também o tempo de execução dos métodos adiciona e paraArray, porque os métodos recebem e retornam valores, respectivamente, e isso nos permitirá escrutinar seus valores:

import { Negociacao } from './Negociacao';

import { logarTempoDeExecucao } from '../helpers/decorators/index';

export class Negociacoes {

private \_negociacoes: Negociacao[] = [];

@logarTempoDeExecucao()

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

@logarTempoDeExecucao()

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

}

Excelente! Tudo funciona, mas que tal se pudermos dar para o usuário a possibilidade de mostrar o tempo gasto com a chamada do método em segundos, em vez de milissegundos? Podemos passar um parâmetro para nosso decorator e a partir desse parâmetro ajustar como ele deve se comportar:

export function logarTempoDeExecucao(emSegundos: boolean = false) {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

descriptor.value = function(...args: any[]) {

let divisor = 1;

let unidade = 'milisegundos'

if(emSegundos) {

divisor = 1000;

unidade = 'segundos';

}

console.log('-----------------------')

console.log(`Parâmetros do método ${propertyKey}: ${JSON.stringify(args)}`);

const t1 = performance.now();

const resultado = metodoOriginal.apply(this, args);

console.log(`Resultado do método: ${JSON.stringify(resultado)}` )

const t2 = performance.now();

console.log(`${propertyKey} demorou ${(t2 - t1)/divisor} ${unidade}`);

console.log('-----------------------')

return resultado;

}

return descriptor;

}

}

Se o parâmetro não for passado ou for passado false, será considerado milissegundos. Se o parâmetro for true, usaremos a segundos como unidade:

import { Negociacao } from './Negociacao';

import { logarTempoDeExecucao } from '../helpers/decorators/index';

export class Negociacoes {

private \_negociacoes: Negociacao[] = [];

@logarTempoDeExecucao(true)

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

@logarTempoDeExecucao(true)

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

}

import { logarTempoDeExecucao } from '../helpers/decorators/index';

export abstract class View<T> {

protected \_elemento: JQuery;

private \_escapar: boolean;

constructor(seletor: string, escapar: boolean = false) {

this.\_elemento = $(seletor);

this.\_escapar = escapar;

}

@logarTempoDeExecucao(true)

update(model: T) {

let template = this.template(model);

if(this.\_escapar)

template = template.replace(/<script>[\s\S]\*?<\/script>/g, '');

this.\_elemento.html(template);

}

abstract template(model: T): string;

}

Para não ficarmos poluindo nosso console, vamos deixar apenas o decorator no método update de View.

Nossa aplicação é funcional, mas podemos organizar ainda melhor nosso código. Percebam que no constructor() de NegociacaoController procuramos os elementos do DOM para que possamos extrair seus valores ao adicionarmos uma negociação. Essa busca é feita assim que nossa classe é instanciada. Mas se o usuário abre a aplicação e desiste de adicionar negociações? Nós teríamos varrido o DOM desnecessariamente. Além disso, se nossa aplicação crescer e tiver que interagir com mais elementos, já estaríamos buscando todos de uma vez. Podemos melhorar isso com auxílio de decorators de propriedades.

Antes de criarmos nosso decorator, queremos algo assim:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

@domInject('#data')

private \_inputData: JQuery;

@domInject('#quantidade')

private \_inputQuantidade: JQuery;

@domInject('#valor')

private \_inputValor: JQuery;

Nosso código não compila, claro, mas a ideia é injetarmos os elementos do DOM diretamente na propriedade da classe. Além disso, usaremos a estratégia de *lazy loading*. Por debaixo dos panos, vamos substituir cada propriedade por um getter. Sendo um getter, podemos escrever um bloco de código que ainda assim para o JavaScript ele será considerado uma propriedade. Nesse bloco de código, só buscaremos o elemento do DOM quando o getter for acessado pela primeira vez. Novos acessos retornarão o mesmo elemento!

// app/ts/helpers/decorators/domInject.ts

export function domInject(seletor: string) {

return function(target: any, key: string) {

let elemento: JQuery;

const getter = function() {

if(!elemento) {

console.log(`buscando ${seletor} para injetar em ${key}`);

elemento = $(seletor);

}

return elemento;

}

}

}

Criamos uma função que será nosso getter, mas como faremos a substituição da propriedade alvo do decorator pelo getter que criamos? Faremos isso com auxílio de Object.defineProperty:

// app/ts/helpers/decorators/domInject.ts

export function domInject(seletor: string) {

return function(target: any, key: string) {

let elemento: JQuery;

const getter = function() {

if(!elemento) {

console.log(`buscando ${seletor} para injetar em ${key}`);

elemento = $(seletor);

}

return elemento;

}

Object.defineProperty(target, key, {

get: getter

});

}

}

Não podemos nos esquecer de exportar o decorator através de app/ts/helpers/decorators/index.ts.

Por fim, vamos importá-lo em NegociacaoController e utilizá-los nas propriedades da classe, não esquecendo de remover a busca manual dos elementos do seu constructor:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacoes, Negociacao } from '../models/index';

import { domInject } from '../helpers/decorators/index';

export class NegociacaoController {

@domInject('#data')

private \_inputData: JQuery;

@domInject('#quantidade')

private \_inputQuantidade: JQuery;

@domInject('#valor')

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

private \_negociacoesView = new NegociacoesView('#negociacoesView');

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

// removeu a busca manual dos elementos

this.\_negociacoesView.update(this.\_negociacoes);

}

// código posterior omitido

Quando recarregamos nossa página, nada é exibido no console, perfeito, isso significa que nossos decorators não entraram em ação. Agora, quando adicionarmos uma nova negociação, os elementos serão buscados do DOM uma única vez e reutilizados sem serem buscados novamente. Criamos um decorator que realiza injeção de elementos do DOM com o padrão lazy loading.

Em TypeScript também podemos criar decoradores de classes. Um decorador de classe nos dá acesso ao constructor da class que estamos decorando. Vejamos um exemplo:

export function meuDecoratorDeClasse() {

return function(constructor: any) {

// guarda o constructor original, pois iremos definir um novo

const original = constructor;

// cria um novo constructor. Como ele pode receber nenhum ou mais parâmetros, usamos ...args: any[]

const novo: any = function (...args: any[]) {

console.log("Criando uma instância com New: " + original.name);

// cria a instância da classe quando for chamado

return new original(...args);

}

// importante! O prototype do novo constructor deve ser o mesmo do original

novo.prototype = original.prototype;

// retorna o novo constructor

return novo;

}

}

Nosso decorator exibirá apenas uma mensagem no console indicando que chamará o constructor da classe.

// código anterior omitido

@meuDecoratorDeClasse()

export class NegociacaoController {

// código omitido

}

Este é outro tipo de decorator possível dentro da linguagem.

É muito comum aplicações acessarem API's de terceiros, com nossa aplicação não será diferente. Temos um servidor escrito em Node.js que disponibilizará a API que consumiremos e ele pode ser baixado [aqui](https://s3.amazonaws.com/caelum-online-public/typescript/api.zip). Após descompactar o arquivo, teremos a pasta api. Como levantar o servidor?

Basta abrirmos outro terminal (um é para o nosso lite-server outro para o servidor da API) e dentro da pasta api executarmos o comando:

npm start

Através do endereço http://localhost:8080/dados temos acesso a um JSON com a seguinte estrutura:

[{"montante":200.5,"vezes":2},{"montante":100.2,"vezes":5},{"montante":50.5,"vezes":1},{"montante":70.5,"vezes":2}]

Nele, temos os dados legados de negociações de outro sistema. Nele, não há o campo data e os campos quantidade e valor se chamam vezes e montanterespectivamente. Nosso cliente disse que podemos usar a data da exportação como data para termos os dados suficientes de uma negociação. É o velho papel de integração entre sistemas. Em suma, nosso objetivo será consumir essa API, criar uma lista adequada de negociações e adicioná-las na tabela que é exibida para o usuário.

Primeiro, vamos adicionar o botão importar em app/index/.html:

<!- app/index.html -->

<!-- código anterior omitido -->

<button class="btn btn-primary" type="submit">Incluir</button>

<button id="botao-importa" class="btn btn-primary" type="button">Importar</button>

</form>

<!-- código posterior omitido -->

Agora, em NegociacaoController, vamos criar o método importarDados que apenas exibe um alerta quando chamado:

// app/ts/controllers/NegociacaoController.js

importarDados() {

alert('oi');

}

Agora, em app/ts/app.ts vamos associar o clique do botão com a chamada do novo método:

import { NegociacaoController } from './controllers/NegociacaoController';

const controller = new NegociacaoController();

$('.form').submit(controller.adiciona.bind(controller));

$('#botao-importa').click(controller.importarDados.bind(controller));

Perfeito, acessando nossa aplicação, ao clicarmos no novo botão, o alerta é exibido. Agora que temos a estrutura pronta, podemos dar início à importação dos dados da API.

Para consumirmos a API externa, utilizaremos a API fetch que usa o padrão de projeto Promise. Por usar Promise, seu uso é mais simplificado do que trabalharmos com XMLHttpRequest.

// app/ts/controllers/NegociacaoController.js

importarDados() {

fetch('http://localhost:8080/dados')

}

Como fetch faz parte do ECMASCRIPT, o TypeScript é capaz de validar a sintaxe que utilizaremos, pois ele já possui um type definition para ele. No curso de JavaScript avançado da Alura, no terceiro curso, a Fetch API é abordada com seus pormenores, mas uma explicação breve para relembrar ou introduzir para quem nunca a viu a API pode acelerar o tempo de construção do nosso projeto.

// app/ts/controllers/NegociacaoController.js

importarDados() {

fetch('http://localhost:8080/dados')

.then(res => res.json())

}

Através da chamada da função then temos acesso à resposta que precisa ser convertida (parse) adequadamente e a Fetch API já traz na própria resposta o método .json() que realiza essa conversão de JSON para objetos em JavaScript. Como usamos arrow function sem bloco, o resultado da instrução res.json() é retornado automaticamente sem a necessidade de usarmos um return e quando fazemos isso, temos acesso ao retorno na próxima chamada encadeada à função then.

Contudo, pode ser que o servidor execute corretamente nossa operação (sem devolver erro 500, por exemplo), mas devolva algum código de status que indique erro, dessa forma, em vez de ficarmos acessando cada código em separadamente podemos acessar status.ok para saber se a resposta é válida com um status válido.

Vamos isolar a lógica do teste em uma função que retorna res caso esteja tudo correto ou lance uma exceção caso haja algum problema. No caso, podemos usar o tipo Response para res, garantindo assim a checagem do TypeScript evitando o acesso a métodos que não existam, seja por um momento lacônico nosso ou por um erro de digitação:

// app/ts/controllers/NegociacaoController.js

importarDados() {

function isOK(res: Response) {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

}

Agora, encadeando uma nova chamada ao then, temos acesso aos dados parseados:

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

.then(dados => {

})

Mas o TypeScript reclamará em breve que há um tipo implícito any sendo empregado pelo nosso código. Resolvemos isso explicitando que o tipo retornado é any[], um array de qualquer tipo:

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

.then((dados: any[]) => {

})

Agora basta convertemos os dados recebidos em uma lista de negociações e adicioná-las na tabela. Não podemos nos esquecer de chamarmos o método update da view que representa nossa tabela:

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

.then((dados: any[]) => {

dados

.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

})

Por fim, vamos adicionar um catch para lidar com qualquer erro que possa ter ocorrido em nossa requisição assíncrona:

importarDados() {

function isOK(res: Response) {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

.then((dados: any[]) => {

dados

.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

})

.catch(err => console.log(err.message));

}

Excelente, se clicarmos no botão importar conseguimos importar todos os dados. Lembre-se que podemos adotar new Date() como data dos dados recebidos, pois eles não definem a propriedade data.

Apesar de funcional nosso código deixa a desejar. É isso que veremos no próximo vídeo.

Como os dados recebidos são do tipo any[], o TypeScript não consegue detectar erros como a escrita da propriedade errada. Podemos resolver isso criando uma Interface:

// app/ts/models/NegociacaoParcial.ts

export interface NegociacaoParcial {

vezes: number,

montante: number;

}

Veja que a interface define duas propriedades e seus tipos. Não é à toa que essas propriedades são as mesmas da nossa API externa. Agora, sem a necessidade de instanciarmos uma classe, até porque interfaces não podem ser instanciadas, podemos usar o tipo NegociacaoParcial[] em vez de any[] nos dados retornados pela nossa API. Isso será suficiente para que o TypeScript consiga analisar nosso código e checar por erros, inclusive fornecendo autocomplete:

importarDados() {

function isOK(res: Response) {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

fetch('http://localhost:8080/dados')

.then(res => isOK(res))

.then(res => res.json())

.then((dados: NegociacaoParcial[]) => {

dados

.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

})

.catch(err => console.log(err.message));

}

Essa solução não blinda da mudança inesperada da API, por exemplo, mudar montantepara valor. Continuaríamos com um erro em runtime. Mas uma vez detectado o erro, podemos alterar nossa interface NegociacaoParcial mudando de montante para valore isso resultará em um erro de compilação em todos os lugares que usam a interface. Perfeito, pois enquanto o desenvolvedor não ajustar para o formato da API que mudou, seu código não compilará. As chances de ele esquecer de mudar são nulas, pois seu código só compilará após a correção.

Do jeito que nossa aplicação esta, se clicarmos muitas vezes em seguida no botão de importar negociações executaremos várias requisições para o servidor. Não há necessidade de tanta ansiedade por parte do usuário, mas para nos prevenirmos disso, podemos ignorar cliques que estejam dentro de uma janela de tempo definida pelo programador. No caso, se o usuário clicar em menos de meio segundo, cancelaremos a ação dele e iniciaremos uma nova. Em suma, essa solução utilizará um timer para processar as ações do usuário:

// app/ts/controllers/NegociacaoController.ts

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacao, Negociacoes } from '../models/index';

import { domInject } from '../helpers/decorators/index';

import { NegociacaoParcial } from '../models/index';

let timer = 0;

export class NegociacaoController {

// código posterior omitido

importaDados() {

function isOk(res: Response) {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

clearTimeout(timer)

timer = setTimeout(() => {

fetch('http://localhost:8080/dados')

.then(res => isOk(res))

.then(res => res.json())

.then((dados: NegociacaoParcial[]) => {

dados

.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao))

this.\_negociacoesView.update(this.\_negociacoes);

})

.catch(err => console.log(err));

}, 500);

}

}

// código posterior omitido

Excelente, nosso código funciona. Quando o usuário clicar pela primeira vez, qualquer temporizador que esteja sendo processado será cancelado e um novo será criado. É esse temporizador que executará nosso código. Se os meio segundos passarem, o temporizador executará nosso código. Então, se ele ficar clicando várias vezes em menos de meio segundo, apenas a última operação será executada.

No entanto esse código é muito invasivo, pois mistura sua lógica com a lógica do método do controller. Além disso, podemos querer usar a mesma estratégia em outros métodos da nossa aplicação. Aprendemos a criar decorators que isolam um código e que podem ser utilizados em vários lugares. Vamos criar o decorator chamado throttle, aliás, o padrão de projeto que lida com o problema que estamos enfrentando:

// app/ts/helpers/decorators/throttle.ts

export function throttle(milissegundos = 500) {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

let timer = 0;

descriptor.value = function(...args: any[]) {

clearInterval(timer);

timer = setTimeout(() => metodoOriginal.apply(this, args), milissegundos);

}

return descriptor;

}

}

// app/ts/helpers/decorators/index.ts

export \* from './logarTempoDeExecucao';

export \* from './domInject';

export \* from './throttle';

Agora, basta utilizarmos nosso decorator no método do nosso controller:

// app/ts/controllers/NegociacaoController.ts

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacao, Negociacoes } from '../models/index';

import { domInject, throttle } from '../helpers/decorators/index';

import { NegociacaoParcial } from '../models/index';

// não precisa mais da variável time!

export class NegociacaoController {

// código anterior omitido

@throttle()

importaDados() {

function isOk(res: Response) {

if (res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

fetch('http://localhost:8080/dados')

.then(res => isOk(res))

.then(res => res.json())

.then((dados: NegociacaoParcial[]) => {

dados

.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

.forEach(negociacao => this.\_negociacoes.adiciona(negociacao))

this.\_negociacoesView.update(this.\_negociacoes);

})

.catch(err => console.log(err));

}

}

// código posterior omitido

Contudo, se usarmos o mesmo decorator como método adiciona() de NegociacaoController teremos um problema. A página será recarregada. Isso acontece, porque como postergarmos a execução do nosso código, postergamos também event.preventDefault(). A boa notícia, é que podemos acessar event implicitamente dentro da chamada de qualquer função, sem necessariamente ele ter sido passado como parâmetro ou não para ela. Nesse sentido, vamos realizar sempre um event.preventDefault() em todos os métodos que usarem nosso decorator throttle:

// app/ts/helpers/decorators/throttle.ts

export function throttle(milissegundos = 500) {

return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const metodoOriginal = descriptor.value;

let timer = 0;

descriptor.value = function(...args: any[]) {

if(event) event.preventDefault();

clearInterval(timer);

timer = setTimeout(() => metodoOriginal.apply(this, args), milissegundos);

}

return descriptor;

}

}

Veja também que o TypeScript saber que event é uma variável implícita que pode existir ou não em uma função e por isso adota o tipo implícito Event, fantástico!

A lógica de acesso à nossa API esta fixa no método importaDados de NegociacaoController. O problema dessa abordagem é que se quisermos acessar a API em outro controller, teremos que repetir código. Podemos enviar isso isolando a lógica de importação na classe NegociacaoService.

// app/ts/services/NegociacaoService.ts

import { NegociacaoParcial, Negociacao } from '../models/index';

export class NegociacaoService {

obterNegociacoes(handler: Function): Promise<Negociacao[]> {

return fetch('http://localhost:8080/dados')

.then(res => handler(res))

.then(res => res.json())

.then((dados: NegociacaoParcial[]) =>

dados.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

)

.catch(err => console.log(err));

}

}

Um ponto a destacar é que nosso método recebe uma handler, isto é, a função que considerará ou não a operação válida. Não podemos deixar simplesmente fixo esse código no serviço, queremos dar flexibilidade para o programador pode aplicar outros critérios que não sejam res.ok.

Por fim, o retorno do método será a Promise, resultado de fetch. No entanto, Promiseé um tipo genérico e precisamos indicar qual tipo estará disponível ao acessarmos seu retornado através de then. É por isso que usamos Promise<Negociacao[]>, pois através de then teremos acesso ao array de negociações.

Como de costumo, vamos criar um barrel para facilitar a importação do serviço:

// app/ts/servicos/index.ts

export \* from './NegociacaoService';

Agora, vamos importar nosso serviço, inclusive adicionar como propriedade de NegociacaoController uma instância dessa classe para que possamos utilizá-la:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacao, Negociacoes } from '../models/index';

import { domInject, throttle } from '../helpers/decorators/index';

import { NegociacaoParcial } from '../models/index';

import { NegociacaoService } from '../services/index';

export class NegociacaoController {

// código anterior omitido

// mais uma propriedade da classe!

private \_service = new NegociacaoService();

// código anterior omitido

@throttle()

importaDados() {

function isOk(res: Response) {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

}

this.\_service

.obterNegociacoes(isOk)

.then(negociacoes => {

negociacoes.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

});

}

Excelente! Agora, se o acesso a API mudar, mudaremos em um único lugar que ele se refletirá em todos os lugares que nossa classe de serviço for utilizada. No entanto, nosso código deixa a desejar. O motivo? Veremos no próximo vídeo.

O método importaNegociacoes de NegociacaoServive aceita receber uma função. Não é à toa que usamos o tipo Function. No entanto, podemos passar qualquer função. Por exemplo, uma que não recebe um Response e muito menos o retorna:

Alguém nos impede de fazermos:

// dará erro, passar function() {} ou () => {}

this.\_service

.obterNegociacoes(function() {})

.then(negociacoes => {

negociacoes.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

});

Podemos restringir um pouco mais quais funções podem ser passadas para o método através de uma interface. Com ela, apenas as funções que receberem o único parâmetro Response e retornarem Response serão válidos:

// app/ts/services/NegociacaoService.ts

import { NegociacaoParcial, Negociacao } from '../models/index';

export class NegociacaoService {

obterNegociacoes(handler: ResponseHandler): Promise<Negociacao[]> {

return fetch('http://localhost:8080/dados')

.then(res => handler(res))

.then(res => res.json())

.then((dados: NegociacaoParcial[]) =>

dados.map(dado => new Negociacao(new Date(), dado.vezes, dado.montante))

)

.catch(err => console.log(err));

}

}

export interface ResponseHandler {

(res: Response): Response

}

A interface ResponseHandler é o tipo que define que a função deve receber um parâmetro do tipo Response e devolver um Response. Inclusive, lá no método obterNegociacoes mudamos o tipo de Function para ResponseHandler.

Agora, em NegociacaoController, nosso código não compila com a função indevida que passarmos e somos alertados disso através do compilador do TypeScript. Se passarmos isOk novamente, tudo continuará funcionando, pois isOk segue a assinatura da interface.

Podemos até fazermos isso se desejarmos:

// app/ts/controllers/NegociacaoController.ts

// código anterior omitido

@throttle()

importaDados() {

const isOk: ResponseHandler = (res: Response) => {

if(res.ok) return res;

throw new Error(res.statusText);

}

this.\_service

.obterNegociacoes(isOk)

.then(negociacoes => {

negociacoes.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

});

}

// código posterior omitido

Ou, para evitarmos que digitar um pouco, podemos passar uma arrow function diretamente:

// app/ts/controllers/NegociacaoController.ts

// código anterior omitido

@throttle()

importaDados() {

this.\_service

.obterNegociacoes(res => {

if(res.ok) return res;

throw new Error(res.statusText);

})

.then(negociacoes => {

negociacoes.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

});

}

// código posterior omitido

Nesse caso nem precisamos mais importar ResponseHandler.

Durante a construção da nossa aplicação utilizamos de maneira corriqueira a função console.log para escrutinarmos o valor das propriedade de nossos objetos. Por exemplo:

console.log(negociacao);

Mas dessa forma, estamos pegando carona no padrão de impressão do console.log. Se quisermos algo mais detalhado teremos que explicar isso em nosso código. Vejamos:

console.log(

`Data: ${this.data}

Quantidade: ${this.quantidade},

Valor: ${this.valor},

Volume: ${this.volume}`

)

Muito mais explicativo, mas imagina termos que repetir esse código em todo lugar que quisermos imprimir nosso objeto Negociacao? Podemos isolar essa lógica na própria classe através do método paraTexto().

export class Negociacao {

constructor(readonly data: Date, readonly quantidade: number, readonly valor: number) {}

get volume() {

return this.quantidade \* this.valor;

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(

`Data: ${this.data}

Quantidade: ${this.quantidade},

Valor: ${this.valor},

Volume: ${this.volume}`

);

}

}

Agora, toda vez que quisermos escrutinas os valores das propriedades do instância de Negociacao para fazermos:

negociacao.paraTexto();

Esse comportamento não é exclusivo de Negociacao, queremos a mesma coisa para Negociacoes. Contudo, não podemos simplesmente reaproveitar o código do método paraTexto() em Negociacoes, porque, apesar do método ter o mesmo nome, quantidade de parâmetros (nenhum) e retorno void, sua implementação é diferente.

Vamos adicionar o método paraTexto() em Negociacoes:

import { Negociacao } from './Negociacao';

export class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(JSON.stringify(this.\_negociacoes));

}

}

Perfeito, agora podemos fazer:

negociacao.paraTexto();

this.\_negociacoes.paraTexto();

Contudo, muitas vezes queremos imprimir no console uma sequencia de objetos. Que tal criarmos uma função chamada imprime que receba como parâmetro um ou mais elementos que desejamos invocar o método paraTexto. Sabemos que através do REST Operator podemos passar uma quantidade indeterminada de parâmetros para a função.

Vamos criar nossa função em app/ts/helpers/Utils.ts. O nome Utils não é por acaso, todas nossas funções utilitárias ficarão nesse arquivo.

Agora, em Utils.ts, vamos criar e exportar a função imprime:

import { Negociacao } from '../models/index';

export function imprime(...negociacoes: Negociacao[]) {

negociacoes.forEach(negociacao => negociacao.paraTexto());

}

E claro, nosso barrel:

// app/ts/helpers/index.ts

export \* from './Utils';

Perfeito, agora vamos importar nossa função utilitária em NegociacaoController e utilizá-la para chamar, numa tacada só, o método paraTexto() de cada objeto que passarmos para ele:

import { NegociacoesView, MensagemView } from '../views/index';

import { Negociacao, Negociacoes } from '../models/index';

import { domInject, throttle } from '../helpers/decorators/index';

import { NegociacaoParcial, Imprimivel } from '../models/index';

import { NegociacaoService } from '../services/index';

// importou a função utilitária

import { imprime } from '../helpers/index';

// código anterior omitido

adiciona() {

// código anterior omitido

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

// imprime no console a negociacao

imprime(negociacao);

}

Excelente! Agora, vamos passar um segundo parâmetro, desta vez, this.\_negociacoes:

// código anterior omitido

adiciona() {

// código anterior omitido

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso!');

// OPS! Erro de compilação!

imprime(negociacao, this.\_negociacoes);

}

Temos um problema. A função imprime aceita receber apenas objetos do tipo Negociacao, por isso estamos tendo o erro

Argument of type 'Negociacoes' is not assignable to parameter of type 'Negociacao'.

E agora? Precisamos fazer com que ele aceite qualquer objeto. Que tal usarmos o tipo any[] em vez de Negociacao[]? Isso significa que podemos passar qualquer objeto.

export function imprime(...objetos: any[]) {

objetos.forEach(objeto => objeto.paraTexto());

}

Excelente, essa alteração silencia o compilador e nosso código funciona como esperado. No entanto, o que acontecerá se fizermos isso:

imprime(negociacao, this.\_negociacoes, data);

Estamos passando um objeto do tipo Date. Como usarmos o tipo any[], ele será aceito. Mas será que nossa função imprime funcionará em tempo de execução? Um teste exibe o seguinte erro no console:

Uncaught TypeError: objeto.paraTexto is not a function

É, não vai funcionar, porque Date não possui o método paraTexto. Do jeito que esta, nada impede do programador passar qualquer objeto que não tenha o método paraTexto. O ideal seria que o compilador nos avisa-se da ausência desse método, para que possamos descobrir este erro em tempo de compilação e não runtime. A boa notícia é que há um recurso que podemos utilizar, assunto do próximo vídeo.

Uma solução é criarmos a classe Imprimivel para que seja herdada por Negociacao Negociacoes. Como Negociacao é um Imprimivel e Negociacoes também, podemos referenciar instâncias dessas classes através do tipo Imprimivel. Como a super classe não sabe de antemão qual a implementação do método paraTexto(), vamos torná-lo um método abstrato, aquele que obriga a classe filha implementá-lo.

// app/ts/models/Imprimivel.ts

export abstract class Imprimivel {

abstract paraTexto(): void;

}

export \* from './Negociacao';

export \* from './Negociacoes';

export \* from './NegociacaoParcial';

export \* from './Imprimivel';

Agora, só precisamos estender a classe Imprimivel em Negociacao e Negociacoes. Como ambas já possuem o método paraTexto() com a mesma assinatura (forma) do método abstrato, não teremos erro de compilação pois já atendemos a obrigação de implementar o método:

import { Imprimivel } from './Imprimivel';

export class Negociacao extends Imprimivel {

constructor(readonly data: Date, readonly quantidade: number, readonly valor: number) {

super();

}

get volume() {

return this.quantidade \* this.valor;

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(

`Data: ${this.data}

Quantidade: ${this.quantidade},

Valor: ${this.valor},

Volume: ${this.volume}`

)

}

}

Veja que não bastou apenas estender Imprimivel, como o constructor da classe filha é diferente da classe pai, fomos obrigados a chamar super() no constructor da classe filha para que o constructor da classe pai seja chamado.

Agora, faremos a mesma coisa com Negociacoes:

import { Negociacao } from './Negociacao';

import { Imprimivel } from './Imprimivel';

export class Negociacoes extends Imprimivel {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(JSON.stringify(this.\_negociacoes));

}

}

Neste caso, não precisamos chamar super(), porque a classe filha não define um constructor e, nesse caso, por padrão, o TypeScript adotará o constructor da classe pai.

Por fim, basta alterarmos agora nossa função impressão para que trabalhe com um array do tipo Imprimivel[]:

import { Imprimivel } from '../models/index';

export function imprime(...objetos: Imprimivel[]) {

objetos.forEach(objeto => objeto.paraTexto());

}

Veja que o autocomplete funciona, mostrando apenas os métodos disponíveis na classe pai, omitindo os métodos classe filha. Não há problema nenhum nisso, porque queremos apenas lidar com objetos que são imprimíveis, e basta eles terem o método paraTexto().

Agora, em NegociacaoController, nosso código não compilará, porque Date não é um Imprimivel:

Argument of type 'Date' is not assignable to parameter of type 'Imprimivel'.

Ao removermos o parâmetro inválido (que não fazia mesmo sentido passarmos) nosso código volta a compilar. Lançamos mão do polimorfismo que consiste em tratar instâncias de objetos de diferentes formas. Sendo assim, apesar de na memória termos objetos do tipo Negociacao e Negociacoes, eles são referênciados através do tipo Imprimivel. Contudo, nossa solução ainda não esta 100% e precisamos meditar sobre ela antes de continuarmos.

Em TypeScript, uma classe só pode herdar de uma classe apenas. Sendo assim, já gastamos essa possibilidade ao herdarmos de Imprimivel. No entanto, não herdamos código nenhum, porque Imprimivel é uma classe abstrata que define apenas um método abstrato, diferente da classe View que definimos como abstrata, mas que possui outros métodos concretos para serem reutilizados. Em suma, lançamos mão da herança para ganharmos polimorfimo e para podemos obrigar as classes Negociacao e Negociacoes a implementarem o método paraTexto(). Quando queremos apenas obrigar classes a seguirem um método contrato de método, podemos também trabalhar com interfaces. Já trabalharmos com interfaces antes, vamos trabalhar com ela agora de outra forma.

Vamos transformar Imprimivel em uma interface que define a obrigatoriedade de implementarmos o método paraTexto():

export interface Imprimivel {

paraTexto(): void;

}

Mudamos de class para interface. Além disso, não é necessário adicionar o modificador abstract, pois todos os métodos de uma interface são abstratos por natureza, não possuem uma implementação, um corpo.

Agora, vamos fazer com que Negociacao e Negociacoes implementes a inteface Imprimivel:

import { Imprimivel } from './Imprimivel';

export class Negociacao implements Imprimivel {

constructor(readonly data: Date, readonly quantidade: number, readonly valor: number) {}

get volume() {

return this.quantidade \* this.valor;

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(

`Data: ${this.data}

Quantidade: ${this.quantidade},

Valor: ${this.valor},

Volume: ${this.volume}`

)

}

}

Veja que não há mais a necessidade de usarmos super(), pois não estamos mais estendendo outra classe.

Alterando Negociacoes agora:

import { Negociacao } from './Negociacao';

import { Imprimivel } from './Imprimivel';

export class Negociacoes implements Imprimivel {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(JSON.stringify(this.\_negociacoes));

}

}

Não precisamos alterar a função imprime, pois o tipo continua sendo Imprimivel[], mas em vez de Imprimivel ser uma classe, agora é uma interface.

Nosso código continua funcionando, excelente. Deixamos de herdar desnecessariamente de uma classe e garantimos um comportamento comum entre objetos através de uma interface. Contudo, há algo curioso.

Não é necessário que uma classe implemente a interface para que seja um valor válido. A implementação de uma interface não deixa o desenvolvedor esquecer de implementar o método, mas podemos fazer algo assim que é totalmente válido:

imprime(negociacao, this.\_negociacoes, { paraTexto: () => console.log('oi')});

Veja que o terceiro parâmetro é um objeto JavaScript que possui a função paraTexto com a mesma assinatura da interface Imprimivel. TypeScript é inteligente e quando encontra uma paridade com o tipo de interface compila o código. Basta trocarmos uma letrinha por exemplo, paraText para que o código não compile.

Vimos que ao implementarmos uma interface que possua métodos, seremos obrigados a implementá-los respeitando sua assinatura. Além disso, podemos utilizar o tipo da interface para referenciar objetos que implemente a interface. Falando em método, isto é, comportamento de objetos, não é raro termos que comparar um objeto com outro. Aliás, a implementação desta comparação pode divergir entre diferentes objetos.

Dessa forma, podemos criar a interface Igualavel que possua o método ehIgual. O método deve receber como parâmetro um objeto de mesmo tipo do objeto envolvido na comparação, retornando true ou fase para indicar igualdade ou não.

// app/ts/models/Igualavel.ts

export interface Igualavel<T> {

ehIgual(objeto: T): boolean

}

Tivermos que lançar mão do uso de generics, pois o método ehIgual deve receber o tipo indicado por T.

Não podemos nos esquecer de exportá-la através de index.ts:

// app/ts/models/index.ts

export \* from './Negociacao';

export \* from './Negociacoes';

export \* from './NegociacaoParcial';

export \* from './Imprimivel';

export \* from './Igualavel';

Agora, vamos fazer com que a classe Negociacao implemente a interface. Diferente da herança com extends, podemos implementar quantas interfaces quisermos:

import { Imprimivel, Igualavel } from './index';

export class Negociacao implements Imprimivel, Igualavel<Negociacao> {

constructor(readonly data: Date, readonly quantidade: number, readonly valor: number) {}

get volume() {

return this.quantidade \* this.valor;

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(

`Data: ${this.data}

Quantidade: ${this.quantidade},

Valor: ${this.valor},

Volume: ${this.volume}`

)

}

ehIgual(negociacao: Negociacao): boolean {

return this.data.getDate() == negociacao.data.getDate()

&& this.data.getMonth() == negociacao.data.getMonth()

&& this.data.getFullYear() == negociacao.data.getFullYear();

}

}

Agora, vamos usar nosso método ehIgual para evitar a importação de negociações duplicada, usam conhecimento fundamental das funções filter, some e forEach:

@throttle()

importaDados() {

this.\_service

.obterNegociacoes(res => {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

})

.then(negociacoesParaImportar => {

const negociacoesJaImportadas = this.\_negociacoes.paraArray();

negociacoesParaImportar

.filter(negociacao =>

!negociacoesJaImportadas.some(jaImportada =>

negociacao.ehIgual(jaImportada)))

.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

});

}

Excelente, mas nosso código pode ficar ainda melhor.

Excelente, mas a implementação de Imprimivel e Igualavel é algo bem comum de implementarmos para praticamente todos nossas classes do modelo. É por isso que podemos criar uma interface que estende essas duas que criamos, bastando implementarmos a nova interface agregado.

Vamos criar a interface MeuObjeto:

import { Imprimivel, Igualavel } from './index';

export interface MeuObjeto<T> extends Imprimivel, Igualavel<T> { }

export \* from './Negociacao';

export \* from './Negociacoes';

export \* from './NegociacaoParcial';

export \* from './Imprimivel';

export \* from './Igualavel';

export \* from './MeuObjeto';

Uma interface pode estender quantas interfaces forem necessárias. No caso, MeuObjeto teve que usar o tipo genético <T> para que esse seja considerado na interface Igualave.

Agora, vamos fazer com que Negociacao e Negociacoes implementem MeuObjeto:

import { MeuObjeto } from './index';

export class Negociacao implements MeuObjeto<Negociacao> {

// código omitido

}

Por fim, vamos implementar em Negociacoes:

import { Negociacao, MeuObjeto } from './index';

export class Negociacoes implements MeuObjeto<Negociacoes> {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao): void {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return ([] as Negociacao[]).concat(this.\_negociacoes);

}

paraTexto(): void {

console.log('-- paraTexto --');

console.log(JSON.stringify(this.\_negociacoes));

}

// precisamos implementar o método ehIgual

ehIgual(negociacoes: Negociacoes): boolean {

return JSON.stringify(this.\_negociacoes) == JSON.stringify(negociacoes.paraArray());

}

}

TypeScript não é uma linguagem exclusiva para frontend, ela pode ser usada também no backend com Node.js. Contudo, como existem milhares (sem exagero) de módulos criados no repositório do npm (um dos maiores do mundo), as chances dos módulos da sua aplicação não terem seu respectivo TypeScript Definition file são gigantes. A única garantia que você terá são as definições dos módulos padrões do Node.js:

npm install @types/node --save-dev

## Como fica o tsconfig.json?

Outro ponto importante, aliás, uma dica, é evitarmos o uso do strictNullChecks e do noImplicityAny. Caso estejam presentes no arquivo tsconfig.js seus valores devem ser false. A ativação dessas configurações geraram inúmeros problemas com possíveis definições que você venha a baixar.

## Como fica o sistema de módulos?

Os módulos do Node.js usam o padrão **commonjs**. Felizmente o compilador TypeScript aceita este parâmetro em na propriedade module do arquivo tsconfig.json.

Vejamos um exemplo de arquivo que configura o TypeScript para um ambiente Node.js:

{

"compilerOptions": {

"target": "es6",

"outDir": "js",

"noEmitOnError": true,

"noImplicitAny": false,

"removeComments": true,

"module": "commonjs",

"strictNullChecks": false,

"experimentalDecorators": true

},

"include": [

"ts/\*\*/\*"

]

}

Há mais um detalhe importante.

## Como realizaremos os imports?

Vejamos um código escrito sem TypeScript. Ele nada mais faz do que criar um novo arquivo no sistema de arquivos:

const { writeFile } = require('fs');

writeFile('teste.txt', 'Gravei no arquivo', err => {

if(err) {

return console.log('Não foi possível criar o arquivo');

}

console.log('arquivo criado com sucesso');

});

No entanto, quando estamos usando TypeScript, precisamos mudar a forma com que importamos nossos módulos:

// veja a difença

import { writeFile } from 'fs';

writeFile('teste.txt', 'Gravei no arquivo', err => {

if(err) {

return console.log('Não foi possível criar o arquivo');

}

console.log('arquivo criado com sucesso');

});

Por debaixo dos panos o TypeScript transcompilará o arquivo para:

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

const fs\_1 = require("fs");

fs\_1.writeFile('teste.txt', 'Gravei no arquivo', err => {

if (err) {

return console.log('Não foi possível criar o arquivo');

}

console.log('arquivo criado com sucesso');

});

Para um codebase já existente, realizar essa mudança pode ser algo muito custoso.

## Exemplo para download

Você pode baixar o exemplo apresentado [neste](https://s3.amazonaws.com/caelum-online-public/typescript/ts_node.zip) link. Para testá-lo, descompacte-o e execute dentro da pasta ts\_node o comando npm install. Depois, npm run compile para compilar o arquivo. Por fim, entre dentro de ts\_node/js e execute o arquivo local.js com a instrução node local.

Aprendemos que o uso de herança e interfaces nos permite escrever códigos genéricos que se beneficiam da tipagem estática. Contudo, há outra forma de escrevemos um código genérico sem o uso de intefaces e sem apelarmos diretamente para o uso do tipo any.

## Union Types

Temos a seguinte função que espera receber um token de autenticação para então tratá-lo (você pode criar um arquivo uion.ts para testar gradativamente o código que veremos. Não é necessário executá-lo, apesar verificar os erros de compilação no VSCode).

function processaToken(token: string) {

// muda o dígito 2 por X!

return token.replace(/2/g,'X');

}

const tokenProcessado = processaToken('1234');

Excelente, mas no sistema legado que estamos utilizando TypeScript, em algums momentos processaToken não recebe uma string, mas um number.

// erro de compilação

const tokenProcessado = processaToken(1234);

Para resolvermos este problema, poderíamos ter usado o tipo any, mas nesse caso perderíamos o autocomplete e a checagem mais rígida do TypeScript que garante a saúde do nosso código.

Podemos fazer com que a função aceite tanto string quanto number através de **union types**. Alterando nosso código:

// ATENÇÃO, NOSSO CÓDIGO AINDA NÃO COMPILARÁ

// agora aceita os tipos string e number!

function processaToken(token: string | number) {

// muda o dígito 2 por X!

/ erro de compilação aqui

return token.replace(/2/g,'X');

}

// compila

const tokenProcessado1 = processaToken('1234');

// compila

const tokenProcessado2 = processaToken(1234);

Podemos agora passar os tipos string e number para processaToken que o compilador TypeScript aceitará, contudo, haverá um erro de compilação dentro da função processaToken, pois o método replace só existe em string e não em number. E agora? Para solucionar esse problema entra em ação o **Type Guards**.

## Type Guards

Podemos fazer com que nosso código compile checando o tipo dentro da função:

function processaToken(token: string | number) {

if(typeof(token) === 'string') {

// typescript entende que é o tipo string e faz autocomplete para este tipo. A função replace só existe em string

return token.replace(/2/g,'X');

} else {

// toFixed só existe em mumber!

return token.toFixed().replace(/2/g,'X');

}

}

const tokenProcessado1 = processaToken('1234');

const tokenProcessado2 = processaToken(1234);

Com Type Guards, quando realizamos o teste verificando o tipo do parâmetro, dentro da condição o compilador saber inferir o tipo correto e a checagem e autocomplete para aquele tipo é ativado. Dentro do if o tipo é considerado string e no else number.

## Considerações sobre o uso de Type Guards

Apesar de ser um recurso da linguagem, essa estratégia remete à programação procedural pois envolve uma sucessão de if's para detectar o tipo dos elementos. É por este motivo que não foi utilizado em nosso projeto e demos preferência ao polimorfismo.

Vimos um requinte da linguagem TypeScript é que é o uso de Union Types e Type Guards.

Vejamos um exemplo:

function processaToken(token: string | number) {

if(typeof(token) === 'string') {

return token.replace(/2/g,'X');

} else {

return token.toFixed().replace(/2/g,'X');

}

}

Contudo, é possível criar um alias para todos os tipos envolvidos no union type. Vejamos:

// criando o alias!

type MeuToken = string | number;

function processaToken(token: MeuToken) {

if(typeof(token) === 'string') {

return token.replace(/2/g,'X');

} else {

return token.toFixed().replace(/2/g,'X');

}

}

O código continua compilando, mas em vez de indicarmos no parâmetro da função os tipos string e number, passamos o alias que criamos, no caso MeuToken. O Type alias é interessante quando trabalhamos com muitos union types e queremos padronizar em um único local esses conjunto de tipos.

É um recurso que não utilizamos em nossa aplicação, mas que o aluno precisa conhecer para poder analisar código de terceiros com maior segurança.

TypeScript se baseia no ES2015, todavia, na versão ES2017 foi introduzida a sintaxe async/await. Ela funciona da seguinte maneira. Dentro de uma uma função ou método async, isto é, uma função ou método declarado como async NomeDoMetodoOuFuncao, podemos tratar o retorno de promises de uma maneira muito especial.

Por padrão, capturamos o retorno de uma promise dentro da função then. Mas se dentro de uma função async, usamos a instrução await antes da chamada de um método **que retorne uma promise**, podemos capturar seu retorno sem a necessidade da chamada de then, como se ela fosse uma função síncrona tradicional.

Vejamos um exemplo:

// o método importDados é um método async!

@throttle()

async importaDados() {

try {

// usou await antes da chamada de this.service.obterNegociacoes()

const negociacoesParaImportar = await this.\_service

.obterNegociacoes(res => {

if(res.ok) {

return res;

} else {

throw new Error(res.statusText);

}

});

const negociacoesJaImportadas = this.\_negociacoes.paraArray();

negociacoesParaImportar

.filter(negociacao =>

!negociacoesJaImportadas.some(jaImportada =>

negociacao.ehIgual(jaImportada)))

.forEach(negociacao =>

this.\_negociacoes.adiciona(negociacao));

this.\_negociacoesView.update(this.\_negociacoes);

} catch(err) {

this.\_mensagemView.update(err.message);

}

}

}

Mas se não chamamos mais then, não chamaremos também catch, certo? Então, como conseguiremos tratar possíveis erros? Quando usamos async/wait, por mais que o código seja assíncrono, podemos usar try e catch para lidar com possíveis exceções em nosso código. Por mais que nosso código pareça um código síncrono, ele continua sendo um código assíncrono.

A boa notícia é que mesmo o TypeScript suportando apenas o ES2015 ele introduziu em sua sintaxe o async/await do ES2017 a partir da sua versão 2.3. Isso não quer dizer que somos obrigados a utilizá-la, mas seu uso melhor bastante a legibilidade do nosso código.