Guia prático de TypeScript

Melhore suas aplicações JavaScript

Casa do Código | alura

THIAGO DA SILVA ADRIANO

Edição

Vivian Matsui
Sabrina Barbosa

[2021]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso: 978-65-86110-77-7

Digital: 978-65-86110-78-4

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

SOBRE O LIVRO

Este livro é destinado a profissionais que têm interesse em aprender a trabalhar com TypeScript por meio de exemplos práticos e reais.

A ideia central é passar tudo o que eu aprendi desenvolvendo projetos com TypeScript nos últimos anos nos meus trabalhos como freelancer e na TV Bandeirantes. Vamos iniciar abordando conceitos básicos, como os tipos suportados pelo TypeScript até a construção de uma API que retorna os dados de uma base de dados MongoDB, que será configurada em um contêiner Docker.

Como pré-requisito, você precisa conhecer lógica de programação, ter um conhecimento básico de JavaScript e muita vontade de aprender algo novo e disposição para replicar cada um dos exemplos aqui demonstrados.

Ao final deste livro, você terá desenvolvido uma solução completa com TypeScript, passando por todas as etapas que eu utilizei para desenvolver uma API para o programa MasterChef em uma de suas edições anteriores.

Para isso, nós utilizaremos as seguintes ferramentas e tecnologias:

- Visual Studio Code
- Node.js
- TypeScript na versão 4.2.3
- Docker para ambiente de desenvolvimento
- MongoDB como base de dados

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus por tudo o que ele tem feito na minha vida! À minha mãe Elenice pelo amor, força, incentivo e por todo o apoio que me deu desde o meu primeiro dia de vida. À minha irmã Thamiris por estar sempre ao meu lado, nos momentos bons e ruins.

À minha esposa Juliana pelo apoio e motivação nesses últimos meses em que me dediquei a escrever este livro e pela parceria de vida.

À minha filha Agnes que, mesmo sem saber ler ou programar, fez algumas revisões comigo brincando e escutando galinha pintadinha.

Ao meu primo Rafael Izidoro, que me proporcionou o melhor investimento da minha vida: uma caixa de cerveja por três meses de aula de programação. Sem esse investimento eu não teria entrado para a área de tecnologia e não estaria escrevendo este livro hoje.

Ao Gabriel Schade Cardoso por ter acreditado no livro e ter feito a ponte para que ele se tornasse realidade.

Por fim, Vivian Matsui, muito obrigado pela parceria, paciência e dedicação nos últimos meses em que finalizamos este livro juntos e agradeço à Casa do Código pela oportunidade de realizar esse sonho de escrever um livro com vocês.

SOBRE O AUTOR

Figura 1: Thiago Silva Adriano

Sou Microsoft (MVP) e atualmente trabalho como líder técnico na empresa TV Bandeirantes. Nestes últimos anos, foquei nas tecnologias criadas pela Microsoft, mas estou sempre antenado com as novas tendências que estão surgindo no mercado. Sou uma pessoa apaixonada pelo que faz e tem a sua profissão como hobby.

- Blog: https://programadriano.medium.com/
- GitHub: https://github.com/programadriano
- Podcast: https://devshow.com.br, onde eu e alguns amigos falamos sobre vários assuntos em alta na comunidade dev.

Participo das comunidades:

- .NET SP: a maior comunidade de .NET: https://www.meetup.com/pt-BR/dotnet-Sao-Paulo/
- SampaDevs: comunidade criada para compartilhamento de conhecimento sobre todas as tecnologias: https://www.meetup.com/pt-BR/SampaDevs/
- AprendendoJS: comunidade nova criada para *meetups* sobre JavaScript: https://www.meetup.com/pt-BR/learning-nodejs/

Sumário

Versão: 25.8.30

Capítulo 1

INTRODUÇÃO AO TYPESCRIPT

O TypeScript é um pré-processador (*superset*) de códigos JavaScript *open source* desenvolvido e mantido pela Microsoft. Ele foi desenvolvido pelo time do Anders Hejlsberg, arquiteto líder do time TypeScript e desenvolvedor do C#, Delphi e do Turbo Pascal. A sua primeira versão, a 0.8, foi lançada em 1 de outubro de 2012.

Ele foi projetado para auxiliar no desenvolvimento de códigos simples até os mais complexos, utilizando os princípios de Orientação a Objetos, como classes, tipagens, interfaces, Generics etc.

Aqui chegamos à primeira dúvida de todo desenvolvedor que está iniciando com TypeScript: por que tipar o JavaScript?

Essa, na realidade, é uma das vantagens de se trabalhar com esse pré-processador. A utilização de tipagem ajuda no momento de depuração (debug) do nosso código, prevenindo alguns possíveis bugs ainda em tempo de desenvolvimento.

Mas como ele funciona?

Como os navegadores não interpretam código TypeScript, nós

precisamos transpilar o nosso código para uma das versões ECMAScript.

A figura a seguir demonstra o seu processo de compilação (*transpiler*). O arquivo `.ts` é o código desenvolvido em TypeScript, a engrenagem é o compilador, e o arquivo `.js` é o código final JavaScript.

Figura 1.1: Processo de compilação.

1.1 INSTALAÇÃO

Para instalar o TypeScript, é necessário ter o Node.js e o seu gerenciador de pacotes, o NPM, instalados no seu computador. Caso você ainda não tenha, basta acessar o link https://nodejs.org/en/ e baixar um executável de acordo com o seu sistema operacional.

Para verificar se ele foi instalado corretamente ou qual é a versão que você tem instalada, basta abrir um terminal no seu computador e digitar o comando: `node -v && npm -v` .

```
C:\Users\tadriano>node -v && npm -v
v14.15.5
6.14.11
```

Figura 1.2: Versão do Node.js e NPM.

Ao utilizar o `npm` seguido de `install`, estamos passando para o gerenciador que ele deve instalar o pacote informado depois da instrução `install`.

Um ponto importante para destacar neste momento é que, quando trabalhamos com pacotes `npm`, é comum ter alguns deles instalados em modo global, o que nos permite executar esse pacote de qualquer lugar do nosso computador.

Para ter um pacote `npm` instalado em modo global, basta adicionar o `-g` antes do nome do pacote.

Vamos instalar o TypeScript em modo global. Para isso, digite o comando: `npm install -g typescript` no seu terminal.

Uma vez instalado, o compilador do TypeScript estará disponível executando o comando `tsc`. Para verificar a versão instalada, basta digitar o comando `tsc -v` dentro de um terminal.

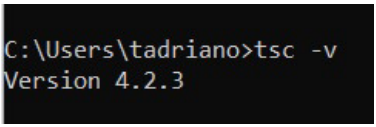

Figura 1.3: Versão do TypeScript.

1.2 EXECUTANDO MANUALMENTE O TYPESCRIPT

Com o TypeScript instalado em modo global, crie um novo diretório no seu computador para organizar os exemplos desta seção. Dentro desse diretório, crie um arquivo chamado: `meuprimeiro.ts`.

Para os próximos passos será necessária a utilização de um editor de textos. Vou utilizar o Visual Studio Code pela sua integração com o TypeScript, mas todos os exemplos aqui demonstrados funcionarão em qualquer editor de textos.

Para instalar o Visual Studio Code, basta baixar um executável correspondente ao seu SO (sistema operacional) e, em seguida, seguir os respectivos passos de instalação padrão. Segue link para download: https://code.visualstudio.com/download

Com o seu editor de textos aberto, abra o arquivo `meuprimeiro.ts` nele e em seguida atualize-o com o seguinte trecho de código:

```
const a : string = 'Hello World';
console.log(a);
```

Abra um terminal no seu computador, navegue até o diretório em que você criou o arquivo `meuprimeiro.ts` e execute o comando `tsc meuprimeiro.ts` .

Note que será criado um novo arquivo chamado `meuprimeiro.js` na raiz de sua pasta.

Agora, para validar o passo anterior, execute o comando `node meuprimeiro.js` no seu terminal. Caso tudo esteja OK, você deve receber a seguinte mensagem no seu terminal: `Hello World` .

1.3 ENTENDENDO O COMPILADOR DO TYPESCRIPT

O compilador do TypeScript é altamente configurável. Ele nos permite definir o local onde estão os arquivos `.ts` dentro do

nosso projeto, o diretório de destino dos arquivos *transpilados*, a versão ECMAScript que será utilizada, o nível de restrição do verificador de tipos e até se o compilador deve permitir arquivos JavaScript.

Cada uma das opções de configuração pode ser passada para um arquivo chamado `tsconfig.json`. Para quem não conhece, esse é o principal arquivo de configuração do TypeScript.

Para criar esse arquivo dentro de um novo projeto, basta executar o comando `tsc --init`. Isso vai criar um arquivo na raiz do seu projeto com algumas configurações padrões, como `target`, `module` entre outras. A seguir você tem uma imagem demonstrando esse arquivo com as configurações *default* dele.

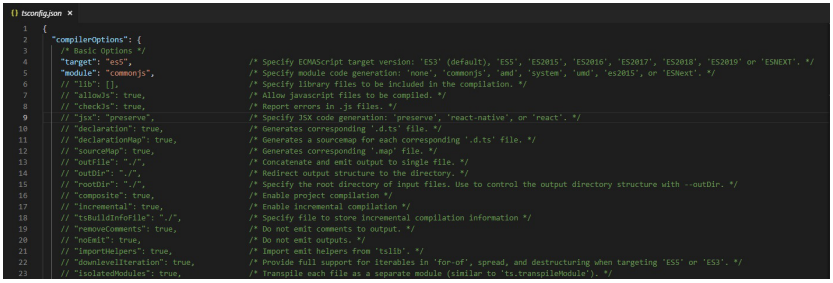

Figura 1.4: Versão do TypeScript.

Para que você possa entender melhor como funciona o compilador, vamos criar um novo arquivo TypeScript.

No mesmo diretório que você criou o seu arquivo de configurações no passo anterior, crie um arquivo com a extensão `.ts`. Para esse exemplo, eu criarei um arquivo chamado `index.ts`, mas você pode escolher um nome de sua preferência.

Um ponto importante que devemos sempre lembrar no momento de criação de novos arquivos, é respeitar a convenção que os desenvolvedores JavaScript já utilizam hoje, a utilização do `camelCase`. Exemplo: `myControl.ts`.

Com o seu arquivo TypeScript criado, atualize-o com o seguinte trecho de código:

```
var languages : Array<string> = [];
languages.push("TypeScript");
languages.push(3);
```

Analisando esse trecho de código, nós criamos um `array` de `string` e estamos adicionando a ele uma string com o valor `TypeScript` e um número com o valor 3.

Será que o compilador vai *transpilar* esse código? Note que nós temos um `array` de `string` e estamos tentando passar um número para ele.

Para validar se o compilador vai *transpilar* esse código, abra um terminal no seu computador, navegue até o diretório em que você criou o arquivo de configurações e o seu arquivo `.ts` e, em seguida, execute o comando `tsc nomeDoSeuArquivo.ts`, trocando "nomeDoSeuArquivo" pelo nome que você deu, no meu caso foi "index".

Note que o compilador *transpilou* o código e gerou o arquivo `index.js` na mesma pasta dos outros arquivos, mesmo entendendo que tem um erro no código.

Agora vem aquela dúvida: o TypeScript não foi desenvolvido para prevenir erros como esse em tempo de desenvolvimento?

A resposta é *sim*. O TypeScript ajuda a prevenir erros como esse em tempo de desenvolvimento, mas para que ele possa alertar ou barrar a geração de arquivos com erro, como o do exemplo anterior, nós devemos informar ao compilador como nós queremos trabalhar com ele através do arquivo `tsconfig.json`. Para isso, abra o seu arquivo e adicione a seguinte configuração nele:

```
"compilerOptions": {
   /*outras configurações*/
   "noEmitOnError": true,
```

Essa configuração fará com que o compilador analise o seu código e, caso ele encontre algum erro, não crie o arquivo `.js`.

Automatizando o compilador

O nosso processo está muito manual. Hoje o nosso exemplo tem somente um arquivo, mas quando estivermos trabalhando em um projeto com mais arquivos `.ts` será necessário executar o comando `tsc + nome do arquivo` em cada um deles?

A resposta é *não*. O TypeScript tem um parâmetro que nós podemos passar para o compilador ficar verificando todos os arquivos do projeto em tempo real, dando feedbacks e, em caso de sucesso, *transpilando* todos de uma vez.

Para utilizar esse parâmetro é bem simples, basta adicionar o `-w` na frente da instrução `tsc`.

Voltando para o nosso projeto e ainda com o seu terminal aberto, execute o comando `tsc -w`.

A seguir você tem uma imagem demonstrando o feedback com

erro do trecho de código anterior, onde estamos tentando passar um número para um `array de string`:

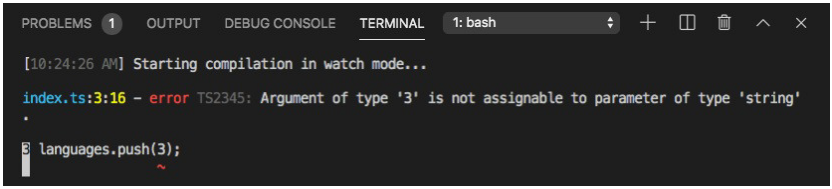

Figura 1.5: Mensagem de erro no compilador.

Uma outra configuração importante de se destacar é a `target`. Nela, nós passamos qual é a versão do ECMAScript que o projeto deve utilizar. Como default, ele vem configurado com o `ES5`, mas você pode atualizar para outra versão, que vai funcionar normalmente.

Nos próximos capítulos, nós criaremos alguns exemplos práticos, em que trabalharemos mais com o arquivo `tsconfig.json`. Caso você tenha interesse em saber mais sobre ele, recomendo a leitura do seguinte link: https://www.typescriptlang.org/docs/handbook/compiler-options.html.

Agora que nós estamos com o ambiente de desenvolvimento configurado, no próximo capítulo, vamos conhecer os types suportados pelo TypeScript.

Capítulo 2

CONHECENDO OS TYPES

Conforme vimos na introdução deste livro, o TypeScript é uma linguagem tipada, como o C#, Java, entre outras. Neste capítulo, vamos abordar os types que ele suporta através de alguns exemplos práticos.

2.1 VAR, LET E CONST

Antes de entrarmos nos types, vamos esclarecer uma das dúvidas mais recorrentes dos novos desenvolvedores JavaScript: qual é a diferença entre `var` , `let` e `const` ?

Var

Até a versão *ES5* do JavaScript, a declaração de variáveis acontecia por meio da palavra reservada `var` . Essas variáveis eram conhecidas como variáveis de escopo.

Um escopo no JavaScript é dado por funções e não por blocos, e a palavra reservada `var` permite que a variável declarada dentro de um escopo seja acessada de qualquer ponto de dentro do código. A seguir você tem um exemplo de declaração de variável utilizando o `var` :

```
var mensagemForaDoIf = 'mensagem fora do if';
```

```
if (true) {
    var mensagemDentroDoIf = 'mensagem dentro do if';
    console.log(mensagemDentroDoIf);
}
console.log(mensagemForaDoIf);
console.log(mensagemDentroDoIf);
```

Executando esse trecho de código, nós temos o seguinte resultado:

```
Resultado:
//mensagem dentro do if
//mensagem fora do if
//mensagem dentro do if
```

Agora vem aquela dúvida: se a variável `mensagemDentroDoIf` foi declarada dentro do `if`, como nós ainda temos acesso a ela fora do bloco de instrução?

Para ficar mais claro, vamos a um outro exemplo:

```
mensagem = 'MSG';
console.log(mensagem);
var mensagem;
```

Nesse exemplo, nós estamos declarando a variável mensagem depois de atribuir um valor. Será que ela vai ser exibida?

```
Resultado:
//MSG
```

Sim, mas como é possível chamar a variável mensagem antes mesmo de declará-la?

Isso é possível devido ao *Hoisting*. Mas o que é *Hoisting*? No JavaScript, toda variável é "elevada/içada" (*hoisting*) até o topo do seu contexto de execução. Então esse mecanismo move as variáveis para o topo do seu escopo antes da execução do código.

No exemplo anterior, a variável `mensagemDentroDoIf` está dentro de uma `function`, então sua declaração é elevada para o topo do seu contexto, ou seja, para o topo da `function`.

É por esse motivo que é possível acessá-la antes de ela ter sido declarada.

Let

A palavra reservada `let` é usada para declarar variáveis com escopo de bloco. Seu comportamento é idêntico ao `var` quando declarada fora de uma `function`, isto é, ela fica acessível no escopo global.

Mas, quando declarada dentro de qualquer bloco, seja ele uma `function`, um `if` ou um `loop`, ela fica acessível apenas dentro do bloco (e sub-blocos) em que foi declarada.

Para ficar mais claro, vamos atualizar o exemplo anterior com `let`.

```
let mensagemForaDoIf = 'mensagem fora do if';

if (true) {
    let mensagemDentroDoIf = 'mensagem dentro do if';
    console.log(mensagemDentroDoIf);
}
console.log(mensagemForaDoIf);
console.log(mensagemDentroDoIf);

Resultado:
//mensagem dentro do if
//mensagem fora do if
//ReferenceError: mensagemDentroDoIf is not defined
```

Note que retornaram as duas mensagens de dentro e fora do `if` e uma mensagem de erro na última linha. Isso ocorreu porque

a variável ficou limitada ao escopo do `if`.

Const

A palavra reservada `const` é usada para declarar variáveis *read-only*, isto é, a variável não pode ter o seu valor alterado, seu estado é imutável. Assim como as variáveis declaradas como `let` e `const` também ficam limitadas a um escopo.

```
const mensagem = 'MSG 1';
console.log(mensagem); // MSG 1
mensagem = 'MSG 2'; // TypeError: Assignment to constant variable


Resultado:
// MSG 1
// TypeError: Assignment to constant variable.
```

2.2 BOOLEAN

Agora entrando nos types, iniciaremos com os tipos booleanos. Como em outras linguagens tipadas, o Boolean suporta dois tipos de valores: `true` ou `false`.

```
let ativo : boolean;
ativo = false;
ativo = true;
```

Nós também podemos declarar uma variável sem o seu type, basta passar o seu valor inicial:

```
let ativo : boolean = true;
```

2.3 NUMBER

No TypeScript, todos os valores numéricos, como `floating`,

`decimal`, `hex`, `octal` devem ser tipados como `number`.

A seguir você tem um exemplo com os tipos numéricos suportados:

```
  let octal: number = 0o745;
  let binary: number = 0b1111;
  let decimal: number = 34;
  let hex: number = 0xf34d;
```

Ou como vimos acima no exemplo com os Booleans:

```
let octal = 0o745;
let binary = 0b1111;
let decimal = 34;
let hex = 0xf34d;
```

2.4 STRING

As strings armazenam valores do tipo texto. Diferente de outras linguagens de programação, no JavaScript/TypeScript nós podemos declarar uma string em aspas simples e aspas duplas.

```
  let cor: string = "verde";
  cor = 'azul';
```

Nós também podemos declarar uma variável do tipo string utilizando *template strings*, dessa forma nós podemos concatenar valores:

```
let nome: string = 'Anders Hejlsberg';
let idade: number = 58;
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${ida
de} anos.
```

2.5 TRABALHANDO COM STRINGS

É muito comum a manipulação de uma string no nosso dia a dia de desenvolvimento. Em alguns cenários, nós precisaremos criar alguns métodos para resolver uma determinada demanda, mas em outros cenários nós podemos utilizar alguns já disponíveis na linguagem. A seguir destacarei alguns dos métodos e propriedades que eu acredito serem os mais utilizados.

Length

Adicionando a palavra reservada `length` no final de uma string, nós temos o tamanho dela.

Para ficar mais claro, vamos verificar o tamanho da nossa string, que foi concatenada no exemplo anterior com a variável `sentence`.

```
//outras variáveis
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${ida
de} anos.`;
console.log(sentence.length); //Resultado 51
```

IndexOf

O método `IndexOf` nos permite encontrar a posição de um caractere ou string. Ele pode ser utilizado para recuperar a posição inicial de um elemento, dentro de uma sequência de caracteres. Caso esse elemento não exista, é retornado o valor -1, caso ele exista, retorna a sua posição.

Para ficar mais claro, vamos utilizar esse método no nosso exemplo anterior, para que ele retorne a posição da palavra `nome` dentro da variável `sentence`.

```
//outras variáveis
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${ida
de} anos.`;
console.log(sentence.indexOf('nome')); //posição 9
```

Caso a gente procure por uma palavra ou caractere que não existe, ele deve retornar `-1` conforme mencionado acima:

```
//outras variáveis
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${ida
de} anos.`;
console.log(sentence.indexOf('idade')); //retorno -1
console.log(sentence.indexOf('!')); //retorno -1
```

Note que ele retornou `-1` para `idade`. Isso aconteceu porque essa palavra é uma variável e não um valor dentro da string.

2.6 ARRAY

Como em outras linguagens de programação, nós declaramos um array no TypeScript utilizando as chaves `[]`.

```
let numeros: number[] = [1, 2, 3];
let textos : string[] = ["exemplo 1", "exemplo 2", "exemplo 3"];
```

Ou nós podemos utilizar a palavra reservada `Array<>`, como no exemplo a seguir:

```
let numeros: Array<number> = [1, 2, 3];
let textos: Array<string> = ["exemplo 1", "exemplo 2", "exemplo 3
];
```

Para adicionar um novo item ao array, nós podemos utilizar a palavra reservada `push`.

```
let numeros: Array<number> = [1, 2, 3];
let textos: Array<string> = ["exemplo 1", "exemplo 2", "exemplo 3
];

numeros.push(4);
```

```
textos.push("exemplo 3");

console.log(numeros);
console.log(textos);

//Resultado
[ 1, 2, 3, 4 ]
[ 'exemplo 1', 'exemplo 2', 'exemplo 3', 'exemplo 3' ]
```

2.7 READONLYARRAY

O `ReadonlyArray<T>` é um array que nos permite somente leitura. Ele remove todos os métodos de alteração de um array, como `push`, `pop` etc.

```
let numerosDaMega: ReadonlyArray<number> = [8, 5, 5, 11, 4, 28];
numerosDaMega[0] = 12; // error!
numerosDaMega.push(23); // error!
numerosDaMega.pop(); // error!
numerosDaMega.length = 100; // error!
```

Caso você tente transpilar esse código, ele vai gerar os seguintes erros no seu console:

```
//Resultado:
index.ts:2:1 - error TS2542: Index signature in type 'readonly nu
mber[]' only permits reading.
2 numerosDaMega[0] = 12;
  ~~~~~~~~~~~~~~~~

index.ts:3:15 - error TS2339: Property 'push' does not exist on t
ype 'readonly number[]'.

3 numerosDaMega.push(23);
                ~~~~

index.ts:4:15 - error TS2339: Property 'pop' does not exist on ty
pe 'readonly number[]'.

4 numerosDaMega.pop();
                ~~~
```

```
index.ts:5:15 - error TS2540: Cannot assign to 'length' because i
t is a read-only property.

5 numerosDaMega.length = 100;
                ~~~~~~

Found 4 errors.
```

À versão 3.4 do TypeScript, foi adicionada uma nova sintaxe para o `ReadonlyArray<T>`. Nessa versão, o time que cuida do TypeScript simplificou a declaração dos `ReadonlyArray<T>` para `Readonly<T>`.

A seguir você tem o exemplo com o array `numerosDaMega` atualizado com essa nova sintaxe `Readonly<T>`:

```
let numerosDaMega: Readonly<number[]> = [8, 5, 5, 11, 4, 28];
```

Acredito que neste momento você deva estar se perguntando: Onde eu poderia utilizar essa funcionalidade?

Pensando nisso, eu pesquisei algumas bibliotecas no portal `NPM` e encontrei uma que exporta dados `json` para um arquivo `.csv` chamada `json2csv`. Link da biblioteca: **json2csv** (https://www.npmjs.com/package/json2csv).

Analisando seus métodos, encontrei o `parseAsync`, que implementa `Readonly<T>`:

```
export function parseAsync<T>(
    data: Readonly<T> | ReadonlyArray<T> | Readable,
    opts?: json2csv.Options<T>,
    transformOpts?: TransformOptions
): Promise<string>;
```

2.8 TUPLE

As tuplas, ou *tuple* no inglês, representam uma estrutura de dados simples semelhante a um array. A grande diferença entre eles é que nos arrays nós trabalhamos somente com um tipo de dado, enquanto com as tuplas temos diferentes tipos.

```
let list: [string, number, string] =  ['string', 1, 'string 2'];
```

Uma das novidades da versão 4.0 do TypeScript é a possibilidade de incluir nomes nos parâmetros das nossas tuplas. Essa é uma das novidades que, na minha opinião, pode ajudar a deixar o nosso código mais legível.

```
let list: [nome: string, idade: number, email: string] = ['Bill G
ates', 65, 'bill@teste.com'];
```

Da mesma forma que nos arrays, para adicionar um novo registro em uma tupla, nós utilizamos o `.push()`.

```
let list: [string, number] = ['Bill Gates', 1];
list.push('Steve', 2);
```

Agora para testar um desses valores, podemos utilizar os seus índices.

```
let list: [string, number] = ['Bill Gates', 1];
console.log(list[0]); //Bill
console.log(list[1]); //1

//Resultado:
//Bill
//1
```

Assim como nos arrays, nós também podemos trabalhar com a palavra reservada `readonly` com as `tuplas`.

```
let list: readonly [string, number] = ['Bill Gates', 1];
list.push('Steve', 2);
```

```
//Resultado:
index.ts:2:6 - error TS2339: Property 'push' does not exist on ty
pe 'readonly [string, number]'.
2 list.push('Steve', 2);
```

2.9 ENUM

O `enum` nos permite declarar um conjunto de valores/constantes predefinidos. Existem três formas de se trabalhar com ele no TypeScript:

- Number.
- String.
- Heterogeneous.

Numérico

Os enums numéricos armazenam strings com valores numéricos. Nós podemos declará-los com um valor inicial.

```
export enum DiaDaSemana {
    Segunda = 1,
    Terca = 2,
    Quarta = 3,
    Quinta = 4,
    Sexta = 5,
    Sabado = 6,
    Domingo = 7
}
```

Caso você não passe um valor inicial na declaração do seu `enum` , o compilador do TypeScript fará um autoincremento de +1 iniciando em 0 até o último elemento do `enum` .

```
export enum DiaDaSemana {
    Segunda,
    Terca,
    Quarta,
```

```
    Quinta,
    Sexta,
    Sabado,
    Domingo
}
```

Ou, no caso de passarmos um valor numérico inicial, ele fará como no passo anterior, incrementando +1 até o último elemento:

```
export enum DiaDaSemana {
    Segunda = 18,
    Terca,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo
}
```

Para acessar um valor dentro de um `enum`, nós podemos utilizar uma das formas a seguir:

```
let dia = DiaDaSemana[19]; // Terca
let diaNumero = DiaDaSemana[dia]; // 19
let diaString= DiaDaSemana["Segunda"]; // 18

Resultado:
// Terca
// 19
// 18
```

Strings

Diferente dos enums numéricos, os enums do tipo string precisam iniciar com um valor.

```
export enum DiaDaSemana {
    Segunda = "Segunda-feira",
    Terca = "Terça-feira",
    Quarta = "Quarta-feira",
    Quinta = "Quinta-feira",
    Sexta = "Sexta-feira",
```

```
    Sabado = "Sábado",
    Domingo = "Domingo",
}
```

Para acessar os valores de um `enum` do tipo string, basta utilizar uma das formas a seguir:

```
console.log(DiaDaSemana.Sexta); //Sexta-feira
console.log(DiaDaSemana['Sabado']); //Sábado
```

```
Resultado:
//Sexta-feira
//Sábado
```

Heterogeneous

Os enums Heterogeneous são pouco conhecidos. Eles aceitam os dois tipos de valores: strings e números.

```
export enum Heterogeneous {
    Segunda = 'Segunda-feira',
    Terca = 1,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo = 18,
}
```

E como acessar esses valores? A seguir você tem um exemplo demonstrando esse passo:

```
console.log(Heterogeneous.Segunda);
console.log(Heterogeneous['Segunda']);

console.log(Heterogeneous['Terca']);
console.log(Heterogeneous[1]);

console.log(Heterogeneous['Quarta']);
console.log(Heterogeneous['Quinta']);
console.log(Heterogeneous['Sexta']);
console.log(Heterogeneous['Sabado']);
```

```
console.log(Heterogeneous['Domingo']);
```

```
Resultado:
//Segunda-feira
//Segunda-feira
//1
//Terca
//2
//3
//4
//5
//18
```

2.10 UNION

O `union` nos permite combinar um ou mais tipos. Sua sintaxe é um pouco diferente dos outros types, ele utiliza uma barra vertical para passar os tipos que ele deve aceitar.

```
Sintaxe:
(tipo1| tipo2 ...)
```

Para ficar mais claro, vamos a alguns exemplos:

```
let exemploVariavel: (string | number);
exemploVariavel = 123;
console.log(exemploVariavel);
exemploVariavel = "ABC";
console.log(exemploVariavel);
```

```
Resultado:
 //123
//ABC
```

Note que a variável `exemploVariavel` aceitou os dois valores: `123` (número) e `"ABC"` (string).

Agora vem aquela dúvida: nós não podemos passar mais de dois tipos para o `union` ? Caso você também tenha pensado nesse cenário, a resposta é "sim". A seguir você tem um exemplo

demonstrando como passar três tipos para o type `union`:

```
let exemploVariavel: (string | number | boolean);
exemploVariavel = 123;
console.log(exemploVariavel);
exemploVariavel = "ABC";
console.log(exemploVariavel);
exemploVariavel = true;
console.log(exemploVariavel);
```

```
Resultado:
//123
//ABC
//true
```

Mas o `union` não fica limitado à utilização de strings, números e booleans. Nós também podemos passar um `array` para ele:

```
var arr: (number[] | string[]);
var i: number;
arr = [1, 2, 4]

for (i = 0; i < arr.length; i++) {
    console.log(arr[i])
}

arr = ["A", "B", "C", "D"]

for (i = 0; i < arr.length; i++) {
    console.log(arr[i])
}
```

Como parâmetro de funções:

```
function deleteTeste(usuario: string | string[]) {
    if (typeof usuario == "string") {
        console.log(usuario, "deletado");
    } else {
        var i;
        for (i = 0; i < usuario.length; i++) {
            console.log(usuario[i], "deletado");
        }
```

```
    }

Resultado:
//função para deletar um registro
//função para deletar mais de um registro
```

Nesse exemplo, nós podemos passar um registro para o nosso método `deleteTeste` ou passar um array de registros para serem deletados.

Note que nós temos uma nova palavra reservada chamada `typeof` no exemplo anterior, onde criamos uma função que recebe um tipo `union` como parâmetro. O `typeof` é um tipo de guarda, ou, no inglês, *Type Guard*, do JavaScript, que nós podemos utilizar desde a versão 1.4 do TypeScript.

Ele é utilizado em cenários onde precisamos verificar o tipo de um objeto dentro de um bloco condicional, como `if` ou o `switch/case`.

Para ficar mais claro, vamos criar um exemplo.

```
let x: string | number | boolean = 13;
console.log(typeof(x))

//Resultado
number
```

Além do `typeof`, temos outro tipo de guarda chamado `instanceof`. Ele tem a mesma funcionalidade do `typeof`, com a diferença de que o `typeof` retorna o tipo do objeto e o `instanceof` retorna um `boolean`.

```
interface Z { x(): string }

class A implements Z {
    x(): string {
        throw new Error("Method not implemented.");
    }
```

```
}
class B implements Z {
    x(): string {
        throw new Error("Method not implemented.");
    }

}

function exemploComInstanceof(paramentro: Z) {
    if (paramentro instanceof A) {
        console.log("Sou a classe A");
    }
    else if (paramentro instanceof B) {
        console.log("Sou a classe B");
    }
}

exemploComInstanceof(new A());
```

Bem simples, não é? Nesse exemplo, nós criamos uma função chamada `exemploComInstanceof`, que recebe uma interface chamada de `Z`, e validamos se a classe que está sendo passada no parâmetro é a classe `A` ou a `B`.

Para quem está tendo o seu primeiro contato com as palavras reservadas, interfaces e classes neste capítulo, não se preocupe em entender o que elas são agora, nós abordaremos esses temas no decorrer deste livro.

2.11 ANY

O `any` é um dos types que mais causa confusão quando nós estamos iniciando com o TypeScript. Como o TS é tipado e nós temos o `Union` em casos de mais de um valor, a maioria dos desenvolvedores iniciantes em TypeScript se pergunta: Por que utilizar o `any`?

Imagine o seguinte cenário: você está fazendo integração a uma API de terceiros e, mesmo que você tenha uma documentação dessa API, você não conhece 100% a estrutura do outro projeto. Esse é um dos cenários em que o `any` é mais utilizado.

Para ficar mais claro como nós podemos utilizar o type `any`, vamos a um exemplo prático.

```
let variavelAny: any = "Variável";
variavelAny = 34;
variavelAny = true;
```

Note que no exemplo anterior a variável `variavelAny` recebeu três valores diferentes: uma string, um número e um bollean. Em cenários como esse é que podemos utilizar o type `any`.

2.12 TIPANDO FUNÇÕES

O TypeScript também nos permite tipar as nossas funções informando para o compilador qual é o tipo que elas devem retornar.

A seguir você tem um trecho de código com uma função que recebe dois parâmetros `number` e retorna uma string:

```
function calc(x: number, y: number): string {
    return `resultado: ${x + y}`;
}
```

Funções como tipos

No TypeScript, podemos tipar uma variável com o valor de uma função. Pegando o nosso exemplo anterior, vamos atribuir o valor da `function calc` a uma variável.

```
function calc(x: number, y: number): string {
    return `resultado: ${x + y}`;
}

let resultado : string;
resultado = calc(10,15);
console.log(resultado);

Resultado:
//resultado: 25
```

Caso você tente atribuir esse valor a uma variável de um tipo diferente de string, o compilador retornará a mensagem de *erro*:

```
function calc(x: number, y: number): string {
    return `resultado: ${x + y}`;
}

let resultado : number;
resultado = calc(10,15);

console.log(resultado);

Resultado:
//Type 'string' is not assignable to type 'number'.ts(2322)
```

2.13 VOID

O type `void` é bastante utilizado em funções. Diferente do type `any`, que espera o retorno de qualquer valor, o `void` passa para o compilador que aquela função não terá nenhum retorno.

```
function log(): void {
    console.log('Sem retorno');
}
```

2.14 NEVER

Embora o type `never` tenha sido adicionado à versão 2.0 do

TypeScript, ele é pouco conhecido pelos desenvolvedores. Esse type indica que algo nunca deve ocorrer. Para que você possa ter uma ideia de como utilizá-lo no seu dia a dia, vamos a alguns exemplos práticos.

Nós podemos utilizá-lo em funções com *exception*:

```
function verificandoTipo(x: string | number): boolean {
    if (typeof x === "string") {
      return true;
    } else if (typeof x === "number") {
      return false;
    }

    return fail("Esse método não aceita esse tipo de type!");
  }

  function fail(message: string): never { throw new Error(message
); }
```

Nesse exemplo, nós estamos recebendo via parâmetro um type `union` (string e número), que deve retornar um boolean. Caso seja passado um valor que não seja uma string ou um número, a chamada entra em uma outra função chamada `fail`, que retornará uma exceção.

```
verificandoTipo("teste String");
verificandoTipo(10);
let ativo = true;
verificandoTipo(ativo);
```

```
Resultado:
//Ok
//Ok
// retorna uma exception com a mensagem: Esse método não aceita e
sse tipo de type!
```

Nós também podemos utilizá-lo em funções sem retorno:

```
function Update(): never {
  while (true) {
```

```
    console.log("Carregando processos!");
  }
}
```

Essa função é comum no desenvolvimento de jogos onde nós temos um método que funciona como um loop infinito, carregando todos os módulos de um game.

Agora, qual é a diferença entre `void` e `never`?

O type `void` pode receber o valor `null`, que indica ausência de um objeto, ou `undefined`, que indica a ausência de qualquer valor.

O type `never` não pode receber valor. A seguir você tem um exemplo demonstrando os dois types como variáveis:

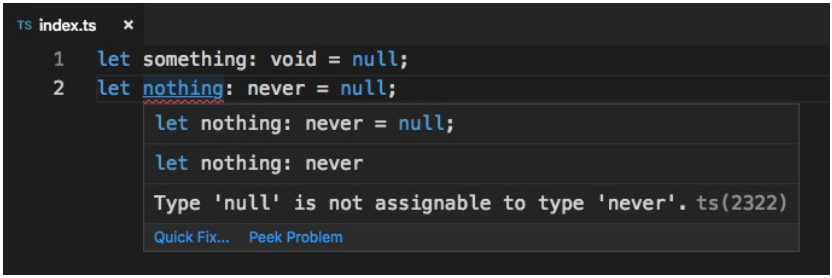

Figura 2.1: Type never x void.

2.15 TYPE ASSERTIONS

O `Type assertion` funciona da mesma forma que o `cast` em outras linguagens de programação. Com ele, nós podemos alterar o type de uma variável, sem que o compilador envie uma exception.

Para ficar mais claro, imagine o seguinte cenário: você tem

uma função X que recebe um parâmetro do type `any`, mas, no corpo da sua função, você tem uma variável do type `number`, que vai receber esse valor do parâmetro. Como resolver?

Com o `Type assertions` é bem simples, basta passar o type que o valor deve receber entre as chaves `<>`. Veja um exemplo demonstrando esse cenário:

```
function typeAssetions(codigoAny: any) {
    let codigoNumber: number = <number>codigoAny;
    return codigoNumber * 10;
}
typeAssetions(10);

Resultado:
//number
```

Agora que nós já conhecemos os types suportados pelo TypeScript, no próximo capítulo, vamos avançar para a estrutura de controle e repetição.

Capítulo 3

ESTRUTURAS DE CONTROLE E REPETIÇÃO

Podemos pensar em estrutura de controle como um bloco de programação que analisa variáveis e escolhe uma direção para seguir com base nos parâmetros predefinidos. A expressão *controle de fluxo* define bem a sua função, sendo nada mais do que o processo básico de tomada de decisões. As estruturas de repetição controlam quantas vezes a instrução deve ser repetida.

Para que essas estruturas fiquem mais claras, vamos a alguns exemplos práticos:

3.1 IF-ELSE

A instrução `if\else` trabalha com validação de valores booleanos. Essa é uma das instruções mais utilizadas no momento de desenvolvimento de um software.

```
let condition = true;

if (condition)
{
    console.log("a variável está com um valor true");
}
else
{
```

```
    console.log("a variável está com um valor false");
}
```

Executando o trecho de código anterior, nós temos o retorno `a variável está com um valor true`, porque o valor de `condition é true` (verdadeiro). Caso o valor de `condition` seja alterado para `false` (falso), o resultado será alterado para `a variável está com um valor false`.

3.2 IF-ELSE-IF

Caso seja necessário validar mais de uma condição de uma determinada variável, nós podemos utilizar o `if\elseif\else`.

A seguir, nós temos um exemplo com uma variável chamada `perfil`.

```
let perfil = "admin";

if (perfil == "superuser") {
    console.log("Super usuário");
}
else if (perfil == "admin") {
    console.log("Adminitrador");
} else {
    console.log("Usuário comum");
}
```

Note que utilizando a instrução `if\elseif\else` nós podemos verificar se a variável `perfil` tem o valor inicial `superuser` ou `admin` e, caso não tenha nenhum desses dois valores, vai retornar que `perfil` é de um usuário comum.

3.3 OPERADOR TERNÁRIO

O operador condicional ternário avalia uma expressão

booleana e retorna o resultado de uma das duas expressões, conforme ela é avaliada como `true` ou `false`.

A seguir nós temos o mesmo exemplo com uma variável chamada `perfil`, só que dessa vez utilizando um operador ternário para avaliar os perfis:

```
let perfil = "admin";
console.log(perfil == "superuser" ? "Super usuário" : "Adminitrad
or");
```

Ou validando os três perfis:

```
let perfil = "admin";
console.log(perfil == "superuser" ? "Super usuário" : perfil == "
admin" ? "Adminitrador" : "Usuário comum");
```

3.4 NULLISH COALESCING

Ao TypeScript 3.7 foi adicionada uma funcionalidade chamada `Nullish Coalescing`, que nos permite verificar se um valor é `null` ou `undefined` utilizando os operadores `??`.

Essa funcionalidade verifica se o valor da direita é `null` ou `undefined` e, caso seja, ela retorna o resultado padrão; caso não, ela retorna o valor da direita.

Para ficar mais claro, vamos criar um exemplo prático com ele.

```
let perfil = "admin";
let perfil = null;
console.log(perfil ?? 'Usuário comum')
console.log(perfil ?? 'Usuário comum')

//Resultado
admin
Usuário comum
```

3.5 SWITCH

A instrução `if` é indicada para pequenos trechos de código. Caso seja necessário validar um trecho de código maior, a instrução `switch case` é a mais indicada.

Nela, nós podemos validar mais de um cenário em uma única instrução. Basta passar o valor a ser validado no parâmetro do `switch` e, em cada `case`, o valor que se deseja verificar.

A seguir você tem um exemplo de como utilizar essa instrução:

```
let perfil = "admin";

switch (perfil) {
    case "superuser":
        console.log("Super usuário");
        break;
    case "manager":
        console.log("Gerente");
        break;
    case "admin":
        console.log("Adminitrador");
        break;
    case "user":
        console.log("Usuário comum");
        break;
    default:
        console.log("sem perfil");
        break;
}
```

O exemplo anterior vai parar no terceiro *case*, o `case admin`. Caso você mude o valor da variável `perfil` para `superuser`, o `switch` retornará `Super usuário`.

Caso o valor de `perfil` não seja encontrado por nenhum dos cases, ele cairá no `default`, então, nesse cenário, o valor `sem perfil` será retornado.

3.6 WHILE

A estrutura de repetição `while` executa a repetição de um bloco de instruções enquanto uma condição é verdadeira. Ela é muito utilizada no desenvolvimento de games.

```
let condicao = true;

while (condicao)
{
    console.log('Carregando...')
}
```

3.7 DO-WHILE

O `do/while` tem quase o mesmo funcionamento da estrutura de repetição `while` , com a diferença de que, com o uso dele, teremos os comandos executados ao menos uma vez.

```
let condicao = false;
{
   console.log('Carregando...')
}
while (condicao);
```

O trecho de código anterior deve retornar o valor `Carregando...` uma vez. Quando ele entrar na estrutura `while` e validar o valor da variável condição que está como false, parará a repetição.

3.8 FOR

O `for` é semelhante ao `while` , pois ele repete o bloco enquanto a condição se mantiver verdadeira. A diferença é que nele passamos um valor inicial e um valor final para o loop iniciar

e terminar.

```
var languages = ["C#", "Java", "JavaScript", "TypeScript"];

for (let i = 0; i < languages.length; i++) {
  console.log(languages[i]);
}
```

Nesse exemplo, o `for` vai incrementar o valor de `i` (aumentar), enquanto `i` for menor que o `array languages`.

3.9 FOREACH

O `foreach` é uma simplificação do operador `for` para trabalhar em coleções de dados. Ele permite acessar cada elemento individualmente iterando sobre toda a coleção.

```
var languages = ["C#", "Java", "JavaScript", "TypeScript"];

languages.forEach(element => {
      console.log(element);
});
```

O exemplo anterior deve retornar todos os elementos do `array language` sem a necessidade de informação de índices como no `for`.

Neste capítulo, nós finalizamos a parte básica do livro. No próximo, vamos entrar em um paradigma muito utilizado no momento de desenvolvimento de software, a Orientação a Objetos.

Capítulo 4

POO (PROGRAMAÇÃO ORIENTADA A OBJETOS)

Antes de falar sobre o paradigma POO (Programação Orientada a Objetos), nós precisamos entender o que é um paradigma e como isso funciona na prática.

O paradigma de uma linguagem de programação é a sua identidade. Ele corresponde a um conjunto de características que, juntas, definem como ela opera e resolve os problemas.

A Orientação a Objetos é um dos primeiros paradigmas que nós aprendemos na faculdade. Caso você analise as principais linguagens da atualidade, a maioria delas possui uma forte base Orientada a Objetos, o que faz com que o seu aprendizado seja essencial.

E como funciona esse paradigma?

Na Orientação a Objetos, nós organizamos o nosso código em estruturas chamadas **classes**.

Uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Ela define o comportamento de seus objetos através de métodos e os estados possíveis desses objetos

através de atributos.

Em outras palavras, uma classe descreve os serviços oferecidos por seus objetos e quais informações eles podem armazenar. Dessa forma, nós podemos ver uma classe como um molde e este molde representa um objeto.

Um objeto é uma representação de algo do mundo real. Para ficar mais claro, veja o exemplo a seguir:

```
Objeto do tipo computador
   Modelo: G3
   Memória: 16gb
   Tipo: Notebook
   Marca: Dell
```

Temos um objeto do tipo computador. Ele contém características particulares, como o seu modelo, memória, tipo e sua marca.

Agora pensando em um outro exemplo:

```
Objeto do tipo computador
   Modelo: MacBook air
   Memória: 8gb
   Tipo: Notebook
   Marca: Apple
```

Assim como no exemplo anterior, temos um objeto do tipo computador. Porém, apesar de ambos compartilharem os mesmos atributos, as suas características são diferentes.

Podemos dizer que ambos são instâncias da classe `Computador`.

Agora que nós temos um exemplo do mundo real, vamos criar outros exemplos aplicando o paradigma de Orientação a Objetos.

4.1 CLASSES

Como esclarecemos, uma classe é um molde com o qual os objetos são *modelados*. É nela que passamos quais atributos um objeto deve ter e quais ações ele deve executar.

Imagine o seguinte cenário: você está modelando um sistema para um banco e precisa criar uma classe para gerenciamento de contas. A primeira pergunta que você deve fazer é: o que todas as contas têm? Em uma breve análise, nós logo identificamos: número da conta, nome do titular e saldo, correto?

Agora que nós temos alguns dos atributos que toda conta tem, vamos à criação da nossa classe.

No TypeScript, a declaração de uma classe é feita utilizando a palavra reservada `class` seguida do nome da classe que queremos implementar:

```
class Conta {
   /* atributos */
}
```

Como boa prática, o código da classe `Conta` deve ficar dentro de um arquivo com o mesmo nome da classe, logo a classe `Conta` será implementada em um arquivo chamado `conta.ts` .

Dentro dessa classe, queremos armazenar as informações que descrevem uma conta. Fazemos isso declarando variáveis dentro da classe, que são chamadas de atributos.

Seguindo o levantamento que nós fizemos anteriormente, a nossa classe `Conta` terá número da conta, titular e saldo.

```
class Conta {
```

```
    numeroDaConta: number;
    titular: string;
    saldo: number;
}
```

Para que possamos acessar esses valores, nós precisamos criar um construtor para a nossa classe e atribuir os valores do seu parâmetro aos nossos atributos.

Em outras linguagens de programação, nós declaramos o construtor com o mesmo nome da classe, mas, no TypeScript, nós utilizamos a palavra reservada `constructor` .

```
class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: nu
mber) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }
}
```

Agora, para criar um objeto a partir da nossa classe, nós precisamos instanciá-la.

Como em outras linguagens que utilizam o paradigma POO, nós utilizamos a palavra reservada `new` para *instanciar/criar* um novo objeto a partir de uma determinada classe.

```
/* implementação da class Conta*/

const primeiraConta = new Conta(1,"Thiago Adriano",1000);
```

Feito isso, temos um objeto do tipo conta com os seguintes

valores:

- número da conta: 1
- titular: Thiago Adriano
- saldo inicial: 1000

4.2 MÉTODOS

Agora que identificamos a nossa classe com seus atributos, podemos nos perguntar: como podemos manipular esses atributos? É nessa hora que o método entra em cena. Ele é responsável por identificar e executar as operações que a classe fornecerá, ou seja, quais serviços e ações a classe oferece. Eles são responsáveis por definir e realizar um determinado comportamento.

Agora que vimos o que é um método e como ele funciona, vamos criar três novos métodos na nossa classe `Conta` : um para adicionar saldo, um para sacar do saldo e um outro para consultar o saldo.

```
class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: nu
mber) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    consultaSaldo(): string {
        return `O seu saldo atual é: ${this.saldo}`;
    }
```

```
    adicionaSaldo(saldo: number): void {
        this.saldo + saldo;
    }

    sacarDoSaldo(valor: number): void {
        this.saldo -= valor;
    }
}
```

4.3 MODIFICADORES DE ACESSO

Com a evolução da nossa classe `Conta`, acabamos deixando uma brecha de segurança: ela nos permite alterar o valor do saldo da conta chamando a propriedade saldo direto. O correto seria alterar esse atributo somente através dos métodos `adicionaSaldo` e `sacarDoSaldo`.

Para resolver esse problema, nós podemos utilizar um modificador de acesso. Caso esse seja o seu primeiro contato com ele, segue uma lista com os modificadores de acesso suportados pelo TypeScript:

- *Public*: é o modificador padrão. Tudo o que for declarado sem um modificador de acesso automaticamente se torna público.
- *Private*: com este modificador, estamos dizendo que esse atributo ou método não pode ser acessado de fora da classe em que ele foi declarado.
- *Protected*: é bem parecido com o `private`, a diferença entre eles é que o `protected` pode ser acessado de uma classe que herda de uma outra classe.

Agora que já conhecemos os modificadores de acesso, vamos corrigir a brecha de segurança do nosso código alterando o atributo `saldo` para `private`.

```
export class Conta {
   /* outros atributos*/
    private saldo: number;
    /* outros métodos*/
```

Dessa forma, conseguimos garantir que o saldo de um determinado cliente seja alterado somente através dos métodos `adicionaSaldo` e `sacarDoSaldo`.

4.4 HERANÇA

Um dos pilares da POO é a herança, ela nos permite reutilizar código sem a necessidade de duplicá-lo.

Imagine o seguinte cenário: chegou uma nova demanda para criarmos dois novos tipos de contas, com alguns atributos diferentes, uma conta para pessoa física e uma outra para pessoa jurídica.

Uma solução seria copiar a mesma classe, mudando seu nome e alguns atributos, mas assim nós estaríamos duplicando o nosso código, o que o tornaria mais complexo e sua manutenção, árdua.

Então como resolver essa demanda sem duplicarmos o nosso código?

Neste momento, entramos em um dos pilares da POO, a herança.

A herança otimiza a produção da nossa aplicação em tempo e

linhas de código. Através dela, podemos herdar os métodos e os atributos de uma outra classe.

Para ficar mais claro, vamos a um exemplo prático. No TypeScript, para herdarmos uma classe, nós utilizamos a palavra reservada `extends`.

```
class ContaPF extends Conta {}
class ContaPJ extends Conta {}
```

Caso você instancie essas classes, note que elas herdarão todas as funcionalidades da classe pai `Conta`.

```
class ContaPF extends Conta {}
class ContaPJ extends Conta {}

const pessoaFisica = new ContaPF(1, "Thiago Adriano", 1000);

const pessoaJuridica = new ContaPJ(1, "Thiago Adriano", 1000);
```

Bem mais simples do que duplicar o código, não é?

Agora, voltando à descrição da nossa demanda, essas duas novas classes têm atributos distintos. Para pessoa física, nós devemos adicionar o atributo CPF e, para pessoa jurídica, o CNPJ.

Vamos adicionar essas novas propriedades começando por PF (pessoa física):

```
class ContaPF extends Conta {
    cpf: number;

    constructor(cpf: number, numeroDaConta: number, titular: stri
ng, saldo: number) {
        super(numeroDaConta, titular, saldo);
        this.cpf = cpf;
    }
}
```

Note que nós temos uma nova palavra no nosso código, a palavra reservada `super`. Ela é utilizada para passar os valores que estamos recebendo via construtor para o construtor da nossa classe pai.

Agora, adicionando o atributo CNPJ na conta PJ (pessoa jurídica):

```
class ContaPJ extends Conta {
    cnpj: number;

    constructor(cnpj: number, numeroDaConta: number, titular: str
ing, saldo: number) {
        super(numeroDaConta, titular, saldo);
        this.cnpj = cnpj;
    }
}

const pessoaJuridica = new ContaPJ(46173051000116, 1, "Thiago Adr
iano", 1000);
```

Com essa implementação, nós conseguimos finalizar a nossa demanda, correto? Sim, mas o nosso código está redondinho? Ainda não.

Analisando a classe `Conta`, note que podemos melhorar alguns pontos, como:

- O número da conta deve ser gerado pela classe e não passado para ela no momento de criação de um novo objeto.
- O método `sacar` deve ter alguma validação de saldo.
- Os métodos da classe pai devem ficar protegidos para que somente as classes filhas possam acessá-los.

Veja a implementação de cada melhoria a seguir:

Geração do número da conta

O primeiro passo será deixar o atributo `numeroDaConta` como `private`, dessa forma ele fica restrito ao escopo da classe `Conta`.

```
export class Conta {
    private numeroDaConta: number;
```

Feito isso, vamos criar um algoritmo para a geração de números aleatórios para que possamos remover o parâmetro `numeroDaConta` do construtor da classe `Conta`:

```
constructor(titular: string, saldo: number) {
        this.numeroDaConta = Math.floor(Math.random() * 1000) + 1
;
```

Observação: esse trecho de código é somente para ilustrar a criação de um número aleatório. Em cenários de produção, nós devemos utilizar algo mais elaborado.

Protegendo os métodos da classe pai

Para proteger os métodos da classe pai, basta adicionarmos a palavra reservada `protected`, que vimos anteriormente, na frente de cada método:

```
export class Conta {
    private numeroDaConta: number;
    titular: string;
    private saldo: number;

    constructor(titular: string, saldo: number) {
        this.numeroDaConta = Math.floor(Math.random() * 1000) + 1
;
        this.titular = titular;
        this.saldo = saldo;
    }
```

```
    protected consultaSaldo(): string {
        return `O seu saldo atual é: ${this.saldo}`;
    }

    protected adicionaSaldo(saldo: number): void {
        this.saldo + saldo;
    }

    protected sacarDoSaldo(valor: number): void {
        this.saldo -= valor;
    }
}
```

Com isso, nós conseguimos garantir que os métodos `consultaSaldo`, `adicionaSaldo` e `sacarDoSaldo` sejam acessados somente pela própria classe `Conta` e pelas classes que a herdarem.

Validando o método de saque

Analisando os tipos de conta PF e PJ, notamos que eles têm uma política diferente no momento do saque. Os clientes que são cadastrados como PF não podem ficar negativos, mas os clientes PJ podem. Como resolver essa demanda?

O primeiro passo será alterar o método `consultaSaldo` na classe `Conta`, para que ele retorne um número e não uma string com o valor atual do saldo da conta.

```
/* implementação da classe Conta*/
 protected consultaSaldo(): number {
        return this.saldo;
    }

/* outros métodos*/
```

O próximo passo será sobrescrever o método `consultaSaldo`

nas classes `ContaPF` e `ContaPJ`:

```
//PF
class ContaPF extends Conta {
    /* implementação da classe Conta*/

     consultar(): string {
        return `Saldo atual: ${this.consultaSaldo()}`;
    }

    /* outros métodos*/
}

//PJ
class ContaPJ extends Conta {
    /* implementação da classe Conta*/

     consultar(): string {
        return `Saldo atual: ${this.consultaSaldo()}`;
    }

    /* outros métodos*/
}
```

Agora vamos atualizar o método `sacarDoSaldo` conforme a demanda que temos começando por PF, onde o cliente não pode ficar negativo. Vamos criar um novo método chamado `sacar`, que valide se o valor do saque é maior que o saldo da conta e se o saldo é maior que 0, e adicionar um trecho de código a ele.

```
class ContaPF extends Conta {
    /* implementação da classe Conta*/
 sacar(valor: number) {
        if (this.consultaSaldo() > 0 && valor <= this.consultaSal
do()) {
            this.sacarDoSaldo(valor);
        }
    }
    /* outros métodos*/
}
```

Como na conta PJ o cliente pode ficar negativo, vamos criar um método chamado `sacar` e passar o valor do seu parâmetro para o método `sacarDoSaldo` da classe pai.

```
class ContaPJ extends Conta {
    /* implementação da classe Conta*/
    sacar(valor: number) {
        this.sacarDoSaldo(valor);
    }
    /* outros métodos*/
}
```

Agora a nossa solução está quase ficando redondinha. Caso você analise novamente a classe `Conta`, note que depois das nossas alterações nós não conseguimos mais pegar o número de uma conta. Na próxima seção, veremos uma forma de resolver esse problema.

4.5 GETTERS & SETTERS

O TypeScript tem suporte aos getters/setters. Caso esse seja o seu primeiro contato com eles, a seguir você tem uma breve descrição sobre cada um:

- **getter**: esse método é utilizado quando queremos acessar o valor de uma propriedade de um objeto.
- **setter**: esse método é utilizado quando queremos alterar o valor de uma propriedade de um objeto.

Voltando ao nosso código, como podemos saber qual é o número da conta de um cliente agora que o atributo `_numeroDaConta` está como privado?

Para resolver esse problema, nós podemos utilizar o `getter`:

```
class Conta {
    private _numeroDaConta: number;
    titular: string;
    private _saldo: number;

    get numeroDaConta(): number {
        return this._numeroDaConta;
    }
 /* outros métodos*/
```

Note que foram adicionados dois underlines, um antes de `numeroDaConta` e outro antes de `saldo`. Essa é uma das convenções utilizadas pela comunidade de desenvolvedores JavaScript para informar que um atributo é privado.

Agora para consultar o número da conta de um cliente PF ou PJ ficou bem simples, basta chamar a propriedade `numeroDaConta`.

Pegando a instância de uma conta PJ como exemplo, nós temos:

```
const pessoaJuridica = new ContaPJ(46173051000116, "Thiago Adrian
o", 1000);
console.log(pessoaJuridica.numeroDaConta);
```

4.6 CLASSE ABSTRATA

Analisando o nosso código novamente, podemos observar que a nossa classe `Conta` está servindo de modelo para as classes `ClientePF` e `ClientePJ`. Agora todo novo cliente será criado a partir dessas duas novas classes. Aqui chegamos a mais um conceito da POO, as classes abstratas.

As classes abstratas não permitem realizar qualquer tipo de instância, elas são utilizadas como modelos para outras classes, que

são conhecidas como classes concretas.

Para tornar a nossa classe `Conta` abstrata, basta adicionar a palavra reservada `abstract` na frente da palavra reservada `class`:

```
abstract class Conta {
    /* implementação da classe*/
}
```

Agora a nossa classe `Conta` será somente um modelo para criação de outras classes, ela não pode mais ser instanciada. Caso você tente instanciá-la, deve receber o seguinte erro:

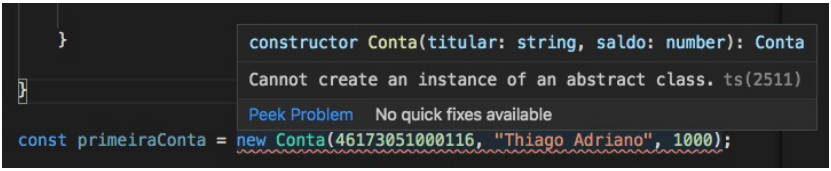

Figura 4.1: Erro ao tentar instanciar classe abstrata.

4.7 READONLY

Analisando os possíveis ajustes na classe `Conta`, notamos que está faltando mais um ponto. O atributo `_numeroDaConta` está como privado, mas nada impede que algum outro método dentro da própria classe altere o seu valor. Como resolver esse problema?

Para isso, temos a palavra reservada `readonly`, que faz com que uma propriedade dentro de uma classe seja utilizada como somente leitura. Uma vez setado o seu valor, ele não pode ser alterado.

```
abstract class Conta {
    private readonly _numeroDaConta: number;
```

```
    /* implementação da classe*/
}
```

Dessa forma, nós conseguimos garantir que a propriedade `_numeroDaConta` não seja alterada por nenhum outro método de dentro da classe `Conta`.

Com isso, nós finalizamos mais este capítulo. No próximo, veremos como trabalhar com as interfaces no TypeScript.

Capítulo 5

INTERFACES

5.1 INTRODUÇÃO A INTERFACES

Além dos tipos primitivos, que abordamos no segundo capítulo deste livro, o TypeScript permite que tipos complexos, como funções e objetos sejam definidos e usados como restrições de tipo. Assim como os objetos literais são a raiz da definição de objeto em JavaScript, os tipos literais de objeto são as definições de um tipo de objeto no TypeScript. Em sua forma mais básica, parece muito com um objeto literal do JavaScript.

A seguir, você tem um exemplo demonstrando a definição de uma variável chamada `pessoa`, que aceita qualquer objeto com as propriedades `nome`, `idade`, `email` e `telefone`.

```
let pessoa: {
  nome: string;
  idade: number;
  email: string;
  telefone: number;
};
```

Note que, diferente de um objeto literal do JavaScript, o tipo literal de objeto separa os campos usando ponto e vírgula, e não vírgulas.

Quando o TypeScript compara dois tipos de objeto para

decidir se eles correspondem ou não, isso é feito estruturalmente. Isso significa que, em vez de comparar os tipos verificando se os dois herdam o mesmo objeto de restrição de base, ele compara as propriedades de cada um dos objetos.

Caso um objeto tenha todas as propriedades que foram definidas no momento da definição da variável, elas são consideradas compatíveis. Para ficar mais claro, vamos a um exemplo prático:

```
let pessoa: { nome: string; idade: number; email: string; telefon
e: number;};

pessoa = { nome: 'Bill', idade: '63', email: 'bill@gmail.com', te
lefone: 555555555};
// Compatíveis, pois esse objeto contém as mesmas propriedades
```

Caso alguma das propriedades não seja compatível ou uma de suas definições esteja faltando, o tipo de objeto será considerado não compatível e o compilador deve gerar um dos erros a seguir:

```
pessoa = { nome: 'Bill', idade: 63, email: 'bill@gmail.com', tele
fone: '555555555' };

/*
Erro: (property) telefone: number
Type 'string' is not assignable to type 'number'.ts(2322)
The expected type comes from property 'telefone' which is declare
d here on type '{ nome: string; idade: number; email: string; tel
efone: number; }'
*/

pessoa = { nome: 'Bill', idade: 63, email: 'bill@gmail.com' };
/*
Erro: Property 'telefone' is missing in type '{ nome: string; ida
de: number; email: string; }' but required in type '{ nome: strin
g; idade: number; email: string; telefone: number; }'.ts(2741)
 'telefone' is declared here.
*/
```

```
pessoa = { nome: 'Bill', idade: 63, email: 'bill@gmail.com', tele
fone: 555555555, endereco: 'rua x' };

/*
Erro: Type '{ nome: string; idade: number; email: string; telefon
e: number; endereco: string; }' is not assignable to type '{ nome
: string; idade: number; email: string; telefone: number; }'.
  Object literal may only specify known properties, and 'endereco
' does not exist in type '{ nome: string; idade: number; email: s
tring; telefone: number; }'.ts(2322)
*/
```

Agora que nós já entendemos como funciona um tipo de objeto e aprendemos a trabalhar com POO, como definir outra variável com os mesmos tipos que foram definidos para `pessoa` sem repetir todas as propriedades?

```
//Exemplo duplicando as propriedades
let pessoa: { nome: string; idade: number; email: string; telefon
e: number;};
let pessoa2: { nome: string; idade: number; email: string; telefo
ne: number;};
```

Uma solução seria utilizar o operador `typeof` para definir uma restrição de tipo.

```
let pessoa: { nome: string; idade: number; email: string; telefon
e: number;};
let pessoa2: typeof pessoa;
```

Esse mecanismo ajuda a reduzir a quantidade de código necessário para fazer referências ao mesmo tipo, mas existe outra abstração ainda mais poderosa no TypeScript para a reutilização de tipos de objetos: as **interfaces**.

A interface é a essência de um tipo literal de objeto. Ela é um conjunto de métodos e propriedades que descrevem um objeto, porém não inicializa nem os implementa. Para ficar mais claro,

vamos alterar o exemplo anterior utilizando uma interface:

```
interface Pessoa {
    nome: string;
    idade: number;
    email: string;
    telefone: number;
}
```

Agora nós podemos passar o tipo `Pessoa` para outras variáveis sem a necessidade de duplicar as suas propriedades:

```
let pessoa: Pessoa;
let pessoa2: Pessoa;
```

Como você pode observar no exemplo anterior, essa alteração permite que o tipo `Pessoa` seja utilizado em vários locais dentro do código sem a necessidade de redefinir seus detalhes repetidas vezes. As interfaces também podem estender outras interfaces ou classes usando a palavra reservada `extends` para compor tipos mais complexos a partir de tipos simples:

```
interface PessoaJuridica extends Pessoa {
    conta: number;
    cnpj: number;
}
```

Neste exemplo, o tipo `PessoaJuridica` estende as propriedades `nome`, `idade`, `email` e `telefone` da interface `IPessoa`, e declara duas novas propriedades: `conta` e `cnpj`.

As propriedades podem ser especificadas como opcionais e, para isso, basta adicionar o operador `?` ao final do nome da propriedade:

```
interface IPessoa {
    nome: string;
    idade: number;
    email: string;
```

```
    telefone?: number;
}
```

Até agora só criamos tipos de objeto com propriedades, mas não demonstramos como adicionar um método a um objeto. O TypeScript nos fornece uma sintaxe abreviada para especificá-los:

```
interface PessoaJuridica extends Pessoa {
    conta: number;
    cnpj: number;
    abrirConta(): boolean;
}
```

Nesse exemplo, adicionamos o método `abrirConta()` na interface `PessoaJuridica`, que não aceita argumento e retorna um `boolean`.

Como as propriedades, os métodos também podem ser opcionais, basta adicionar o operador `?` depois do nome do método:

```
interface PessoaJuridica extends Pessoa {
    conta: number;
    cnpj: number;
    abrirConta?(): boolean;
}
```

Voltando a uma das versões anteriores do TypeScript, a 2.7, a ela foi adicionada a funcionalidade de ter propriedades de nomes, como *constantes*, nos tipos. Isso significa que as interfaces podem ser definidas por strings, constantes, números ou literais.

```
const example1 = 'string';
const example2 = Symbol();

interface MeuExemplo {
    [example1]: string;
    [example2]: boolean;
}
```

Avançando na implementação de uma interface

Bom, até aqui vimos o básico sobre a implementação de uma interface, abordamos o que ela é e como trabalhar com ela tipando um objeto.

Agora avançando nesse assunto, vamos a uma das formas mais comuns de se trabalhar com uma interface na Orientação a Objetos: sua implementação por uma classe.

Quando uma interface é implementada por uma classe, ela é vista como um contrato, obrigando a classe que a está implementando a definir todos os seus métodos e propriedades.

Para que você possa ter um melhor entendimento sobre esse assunto, vamos a um exemplo prático utilizando a linha de raciocínio do capítulo anterior sobre POO. Para isso, imagine o seguinte cenário: agora todas as nossas contas PJ e PF precisam ter um método que calcule o seu valor tributário anual. Como fazer com que os dois tipos de conta implementem esse método?

Utilizando uma interface é bem simples, nós só precisamos definir o nosso contrato, como no exemplo a seguir:

```
interface Tributavel {
    CalculaTributo(): number;
}
```

E implementar essa interface nas nossas contas PF e PJ, utilizando a palavra reservada `implements` mais o nome da nossa interface:

```
class ContaPJ extends Conta implements Tributavel {

  CalculaTributo(): number {
        //implementação do cálculo para o valor tributável para c
```

```
ontas PJ
    }
}

class ContaPF extends Conta implements Tributavel {

  CalculaTributo(): number {
        //implementação do cálculo para o valor tributável para c
ontas PF
    }
}
```

Note que, quando você implementou a interface `Tributavel`, o compilador identificou que dentro desse contrato tem um método chamado `CalculaTributo()` e sublinhou a classe em vermelho passando a notificação de que o método deve ser implementado:

```
//implementação
class ContaPJ implements Tributavel {}


/* Resultado com o erro:
class ContaPJ
Class 'ContaPJ' incorrectly implements interface 'Tributavel'.
  Property 'CalculaTributo' is missing in type 'ContaPJ' but requ
ired in type 'ITributavel'.ts(2420)
conta.ts(2, 5): 'CalculaTributo' is declared here.
*/
```

Agora as nossas duas contas devem implementar o método `CalculaTributo()`, que foi definido na interface `Tributavel`, e adicionar o cálculo de acordo com o seu tipo de conta, sendo PF ou PJ.

Com isso, finalizamos este capítulo sobre interfaces. No próximo, nós abordaremos os tipos genéricos.

Capítulo 6

GENERICS

Seguindo a definição de *Generics* na documentação do TypeScript, ele nos permite criar um componente que pode funcionar em vários tipos, em vez de em um único. Ao contrário dos tipos de objetos não genéricos, os tipos genéricos só podem ser criados com interfaces ou classes.

O Generics não é exclusivo do TypeScript. Quem já trabalha há algum tempo com desenvolvimento de software, já deve ter criado um método genérico ou já utilizou algum da linguagem com a qual trabalhou.

Um Generics muito utilizado é o `Array`. Abrindo o GitHub do fonte do TypeScript, nós podemos ver como ele foi definido utilizando um Generics de uma interface que deve retornar um `T`:

```
interface Array<T> {
    /**
     * Determines whether an array includes a certain element, re
turning true or false as appropriate.
     * @param searchElement The element to search for.
     * @param fromIndex The position in this array at which to be
gin searching for searchElement.
     */
    includes(searchElement: T, fromIndex?: number): boolean;
}
```

Link do trecho de código no GitHub:

https://github.com/microsoft/TypeScript/blob/master/lib/lib.es2016.array.include.d.ts

Agora qual tipo o `T` representa? A princípio ainda não sabemos, só vamos descobrir quando a interface for implementada.

```
//Nesse exemplo T é uma string
const nomes: Array<string> = ['pessoa1','pessoa2','pessoa3']

//Nesse exemplo T é um number
const dias: Array<number> = [5, 25, 28]
```

Depois dessa rápida introdução, vamos criar alguns exemplos práticos.

6.1 CRIANDO UMA FUNÇÃO GENÉRICA

Para criar uma função genérica, basta adicionar as chaves `<T>` .

```
function funcaoGenerica<T>() {}
```

Agora podemos passar qualquer tipo para função `funcaoGenerica` que ela deve aceitar:

```
function funcaoGenerica<T>() {}

funcaoGenerica<number>()
funcaoGenerica<string>()
funcaoGenerica<boolean>()
```

Ficou muito simples, não é? Vamos melhorar o nosso exemplo deixando-o mais próximo do nosso dia a dia:

```
function funcaoGenerica<T>(value: T): T {
    return value;
}
```

Nessa função, nós devemos passar um tipo para `funcaoGenerica<T>` , um argumento genérico e um retorno genérico.

```
function funcaoGenerica<T>(value: T): T {
    return value;
}

console.log(funcaoGenerica<Number>(1))
//retorna 1

console.log(funcaoGenerica<string>('teste'))
//retorna teste

console.log(funcaoGenerica<boolean>(true))
//retorna true
```

Nós podemos também passar mais de um parâmetro para nossa função genérica:

```
function fun<T, U, V>(args1:T, args2: U, args3: V): V {
  return args3;
}

console.log(fun<string, number, boolean>('teste', 1, true))
//retorna true
```

Note que podemos passar outros valores no lugar de `T` .

6.2 CRIANDO UMA CLASSE GENÉRICA

Como nas funções genéricas, nós também podemos passar um parâmetro genérico para uma classe.

```
class classeGenerica<T>  {
    private arr: Array<T> = [];

    adicionaValor(item: T) {
        this.arr.push(item);
    }
```

```
    retornaValores() {
        return this.arr;
    }
}

let classeGenerica1 = new classeGenerica<number>();
classeGenerica1.adicionaValor(4);
console.log(classeGenerica1.retornaValores());
//Retorna [ 4 ]

let classeGenerica2 = new classeGenerica<string>();
classeGenerica2.adicionaValor('Homem de ferro');
console.log(classeGenerica2.retornaValores());
//Retorna [ 'Homem de ferro' ]
```

6.3 CRIANDO UMA INTERFACE GENÉRICA

Para demonstrar a criação de uma interface genérica, criaremos uma interface chamada `InterfaceGenerica` com um método chamado `removeItem`. Esse método deve ser utilizado para remover um item da nossa `classeGenerica`:

```
interface InterfaceGenerica<I> {
    removeItem(item: I)
}

class classeGenerica<T> implements InterfaceGenerica<T> {
   //os outros métodos
     removeItem(item: T) {
        let index = this.arr.indexOf(item);
        if (index > -1)
            this.arr.splice(index, 1);
}
```

Validando a implementação do método `removeItem` na classe `classeGenerica`:

```
let classeGenerica1 = new classeGenerica<number>();
classeGenerica1.adicionaValor(1);
classeGenerica1.adicionaValor(2);
```

```
classeGenerica1.adicionaValor(3);
console.log(classeGenerica1.retornaValores());
//Retorna [ 1, 2, 3 ]
classeGenerica1.removeItem(1);
console.log(classeGenerica1.retornaValores());
//Retorna [ 2, 3 ]

let classeGenerica2 = new classeGenerica<string>();
classeGenerica2.adicionaValor('Homem de ferro');
classeGenerica2.adicionaValor('Homem aranha');
console.log(classeGenerica2.retornaValores());
//Retorna [ 'Homem de ferro', 'Homem aranha' ]
classeGenerica2.removeItem('Homem aranha');
console.log(classeGenerica2.retornaValores());
//Retorna ['Homem de ferro']
```

Com isso, finalizamos mais um módulo. Esse é um tema que alguns desenvolvedores iniciantes não conseguem aplicar no dia a dia, mas acredito que depois dos exemplos demonstrados você já esteja pensando em como utilizá-lo.

No próximo capítulo, nós veremos os decorators.

Capítulo 7

DECORATOR

Neste capítulo, nós abordaremos um recurso muito poderoso que podemos utilizar com TypeScript, o *decorator*.

Os decorators nos permitem *decorar* dinamicamente as características de uma classe.

Atualmente, eles estão disponíveis como um recurso experimental no TypeScript, mas mesmo assim já estão presentes em grandes projetos de código aberto, como o *Angular* e *Inversify*.

Para habilitar esse recurso, será necessário descomentar a linha a seguir dentro do seu arquivo `tsconfig.json` .

```
 "compilerOptions": {
    /*outras configurações*/
   "experimentalDecorators": true, /* Enables experimental suppor
t for ES7 Decorator. */
```

Caso você não se recorde do arquivo `tsconfig.json` , nós falamos sobre ele no capítulo de introdução deste livro.

Com essa funcionalidade habilitada no seu projeto, vamos a alguns exemplos práticos para conhecê-lo melhor e ver como ele pode ajudar no nosso dia a dia.

7.1 ANALISANDO OS DECORATORS EXISTENTES NO TYPESCRIPT

Como mencionado, alguns projetos de código aberto já estão utilizando os decorators. Quem já teve contato com o desenvolvimento de uma aplicação Angular já se deparou com algum dos decorators adiante:

- `@NgModule`
- `@Component`
- `@Injectable`
- `@Directive`
- `@Input`
- `@Output`
- `@ViewChild`

Como você pode observar, um decorator é definido pelo caractere `@` mais o seu nome.

Agora, para que possamos entender melhor como é a criação de um decorator no TypeScript, vamos navegar no seu código-fonte e ver como os seus decorators foram definidos:

```
declare type ClassDecorator = <TFunction extends Function>(target
: TFunction) => TFunction | void;
declare type PropertyDecorator = (target: Object, propertyKey: st
ring | symbol) => void;
declare type MethodDecorator = <T>(target: Object, propertyKey: s
tring | symbol, descriptor: TypedPropertyDescriptor<T>) => TypedP
ropertyDescriptor<T> | void;
declare type ParameterDecorator = (target: Object, propertyKey: s
tring | symbol, parameterIndex: number) => void;
```

Trecho de código retirado do código-fonte do TypeScript que atualmente está no seguinte link do GitHub:

https://github.com/microsoft/TypeScript/blob/d28e38f57306af063e1ee84432f8efa6e7c22093/src/lib/es5.d.ts

Analisando os decorators `PropertyDecorator`, `MethodDecorator` e `ParameterDecorator`, podemos defini-los como funções que recebem três parâmetros:

- `target` (alvo).
- `propertyKey` (chave).
- `descriptor` (descritor).

Vamos analisar cada um deles:

- **target (alvo):** pode ser um método estático ou uma `function` construtora de uma classe.
- **propertyKey (chave):** nome do membro da instância que será utilizado no alvo.
- **descriptor (descritor):** a propriedade `descriptor` do membro da instância, chamando o método `Object.getOwnPropertyDescriptor()`.

Para quem não conhece o `Object.getOwnPropertyDescriptor()`, ele retorna um descritor de propriedades para uma outra propriedade.

7.2 CRIANDO UM MÉTODO DECORATOR

Depois dessa rápida introdução sobre o que é e como funciona um decorator, nada melhor do que criarmos um para entender o seu funcionamento.

Como vimos, basicamente precisamos criar uma função que

recebe três parâmetros: `target`, `propertyKey` e `descriptor`. Para esse exemplo, vamos criar um decorator para analisar o método `consultaSaldo()`, da nossa classe `Conta`, que vimos no capítulo sobre POO.

```
function analisaSaldo(target: any, key: any, descriptor: any) {}
```

Agora vamos decorar o método `consultaSaldo()` da nossa classe `Conta`.

```
function analisaSaldo(target: any, key: any, descriptor: any) {
    //implementação
}

class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: nu
mber) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    @analisaSaldo
    consultaSaldo(): string {
        return `O seu saldo atual é: ${this.saldo}`;
    }
}
```

Resultado:

```
Conta { consultaSaldo: [Function] } consultaSaldo undefined
```

Bem simples, não é? Vejamos agora a criação de outros tipos de decorator que podemos utilizar com TypeScript.

7.3 DECORATOR DE PROPRIEDADE

Um decorator de propriedade pode ser definido por uma função com dois parâmetros:

- `target:`
- `propertyKey:`

O `target` é o protótipo da classe em que está sendo aplicado o decorator, e `key` é o nome da propriedade da classe.

```
function validaTitular(target: any, propertyKey: any) {
    //implementação
}

class Conta {
    numeroDaConta: number;
    @validaTitular
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: nu
mber) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    /* outros  métodos*/
}
```

Resultado:

```
Conta {} titular
```

7.4 DECORATOR DE PARÂMETRO

Um decorator de `parameter` deve ser declarado antes da declaração de um parâmetro e recebe três parâmetros. O primeiro,

como na maioria dos decorators que já vimos, é o `target`, que é o protótipo da classe. O segundo é o `propertyKey`, que é o nome do método que contém o parâmetro com o qual estamos trabalhando. O último é o `parameterIndex`, que é o número da posição do parâmetro na função, lembrando que começa a partir do `0`.

```
function saldo() {
    return (
        target: any,
        propertyKey: number,
        parameterIndex: number,
    ) => {
        console.log('target', target);
        console.log('property key', propertyKey);
        console.log('parameter index', parameterIndex);
    }
}

class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: nu
mber) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    adicionaSaldo(@saldo() saldo: number): void {
        this.saldo + saldo;
    }
}
```

7.5 CRIANDO UM DECORATOR PARA CLASS

O decorator para classe deve ser declarado antes da declaração

da própria classe. Esse decorator recebe um único parâmetro, que é o construtor da classe.

```
function log(ctor: any) {
    console.log(ctor)
}

@log
class Conta {}
```

Resultado:

```
[Function: Conta]
```

7.6 DECORATOR FACTORY

Em alguns cenários em que precisamos fazer uma interação entre uma classe `target` e um decorator, como na implementação de um log onde precisamos passar um determinado valor para o decorator para que ele tome uma ação, como escrever esse valor em um arquivo `.txt` ou enviar um evento *mensagem* para uma API, nós podemos utilizar o decorator `Factory`.

Sua sintaxe é a mesma que a de um decorator de uma classe, com a pequena diferença de que ele recebe um valor através de um parâmetro. A seguir você tem um exemplo, onde estamos passando para o decorator `analisaConta` a informação de que a classe que o está implementando é do tipo `PJ`.

```
function analisaConta(tipoConta: any) {
    return (_target: any) => {
        console.log(`${tipoConta} - ${_target}`);
    }
}

@analisaConta('PJ')
```

```
class Conta {}
```

Resultado:

```
PJ - function Conta() { }
```

7.7 MÚLTIPLOS DECORATORS

Em alguns cenários, precisaremos criar mais de um decorator para as nossas classes. Para ficar mais claro, imagine o seguinte cenário: chegou uma demanda e agora nós precisamos criar um decorator de `log`, um outro para validação de uma regra de negócio X e um outro para verificar se a conta é `PJ` ou `PF`.

Como podemos implementá-los?

O TypeScript nos permite implementar mais de um decorator em uma classe. Seguindo o nosso exemplo, teríamos algo como:

```
function analisaConta(tipoConta: string) {
    return (_target: any) => {
        console.log(`${tipoConta} - ${_target}`);
    }
}

function log(ctor: any) { }
function validaRegra(ctor: any) { }

@log
@validaRegra
@analisaConta('PJ')
class Conta { }
```

Com isso, nós finalizamos mais este capítulo. No próximo, veremos como organizar o nosso código utilizando namespaces.

CAPÍTULO 8

MODULES E NAMESPACES

Neste capítulo, nós abordaremos algumas formas de organizar o nosso código utilizando *modules* e *namespaces*.

8.1 NAMESPACES

Namespace não é algo exclusivo do TypeScript. Quem programa em C# já está habituado a sua utilização por ele ser padrão na criação de uma classe em C#.

Mas caso esse seja o seu primeiro contato com ele, saiba que o namespace nos permite organizar o nosso código, deixando-o agrupado por nome. Para que esse conceito fique mais claro, nada melhor que um exemplo prático.

Vamos voltar ao capítulo 4, sobre Programação Orientada a Objetos, no qual desenvolvemos uma classe `abstract` chamada `Conta`, e a partir dela nós criamos a `ContaPF` e `ContaPJ`.

Tendo esse cenário em mente, imagine que agora chegou uma nova demanda para criarmos duas novas implementações da classe `Conta`, uma para `ContaSalario` e uma outra para `ContaInvestimento`. Até aqui está bem simples, basta criarmos algo como no exemplo a seguir para resolver a nossa demanda.

```
class ContaSalario extends Conta {}
class ContaInvestimento extends Conta {}
```

Mas com essa resolução, nós chegamos a um grande problema que surge com o crescimento dos nossos sistemas: a organização de todas as nossas classes.

Para evitar que o sistema fique caótico, podemos agrupar as classes por características comuns e dar um nome para cada um desses grupos. Isto é, agrupar um conjunto de classes em um espaço em comum e lhe dar um nome, por exemplo `Banco`. Esse espaço definido por um nome é chamado de namespace.

Atualizando o nosso código com a utilização do namespace nós teríamos algo como no exemplo a seguir:

```
namespace Banco {
    export class Conta {
        numeroDaConta: number;
        titular: string;
        saldo: number;
        /* outras implementações */
    }
}
```

Agora para herdarmos a nossa classe `Conta` de um local fora do namespace `Banco`, nós precisamos passar o nome do namespace antes do nome da classe:

```
class ContaPF extends Banco.Conta {}
class ContaPJ extends Banco.Conta {}
```

Caso contrário, o compilador retornará o seguinte erro:

```
class ContaPF extends Conta {}
class ContaPJ extends Conta {}
```

Resultado:

```
//Erro Cannot find name 'Conta'.ts(2304)
```

Analisando o nosso código, não ficaria mais organizado se todas as nossas implementações que surgiram a partir da classe `Conta` fossem agrupadas pelo mesmo namespace `Banco` ? Sim, pois todas estão no mesmo contexto.

A seguir você tem um exemplo de como ficaria essa implementação:

```
namespace Banco {
  export  class ContaPF extends Conta { }
}

namespace Banco {
  export  class ContaPJ extends Conta { }
}

namespace Banco {
  export  class ContaSalario extends Conta { }
}

namespace Banco {
  export  class ContaInvestimento extends Conta { }
}
```

Namespaces aninhados

Um outro recurso que nós temos ao trabalhar com namespaces é a definição de um namespace dentro de outro. Meio confuso, não?

Para ficar mais claro, vamos voltar ao exemplo do namespace `Banco` , dentro do qual nós agrupamos as classes `ContaPJ` , `ContaPF` , `ContaSalario` e `ContaInvestimento` . Sim, elas pertencem ao namespace `Banco` , mas não seria interessante se elas estivessem agrupadas por investimento também?

Tendo esse cenário em mente, nós podemos atualizar o nosso código da seguinte forma:

```
namespace Banco {
    export namespace Investimento {
        export class ContaSalario extends Conta { }
    }
}

namespace Banco {
    export namespace Investimento {
        export class ContaInvestimento extends Conta { }
    }
}
```

Agora, para que possamos instanciar essas classes, basta passar os namespaces antes do nome da classe, como no exemplo a seguir:

```
let novaContaInvestimento = new Banco.Investimento.ContaInvestime
nto();
let novaContaSalario = new Banco.Investimento.ContaSalario();
```

Mas se a ideia é organizar o nosso código, como devemos fazer para separá-lo em arquivos diferentes? Complicou?

Na realidade é bem simples, basta chamá-los da mesma forma que já fizemos, sem a necessidade de nenhuma referência.

```
//contaInvestimento.ts
namespace Banco {
    export namespace Investimento {
        export class ContaInvestimento extends Conta { }
    }
}

//contaSalario.ts
namespace Banco {
    export namespace Investimento {
        export class ContaSalario extends Conta { }
    }
}
```

Agora chamando-os de um outro arquivo:

```
//index.ts
let novaContaInvestimento = new Banco.Investimento.ContaInvestime
nto();
let novaContaSalario = new Banco.Investimento.ContaSalario();
```

8.2 MODULES

Como os namespaces, os *modules* no TypeScript nos ajudam a organizar o nosso código separando as nossas classes. Eles utilizam o mesmo conceito dos namespaces com a utilização da palavra reservada `export`, mas com algumas diferenças: para que possamos trabalhar com eles, nós precisamos de um `module loader` e, para chamar um `module` de um outro arquivo, nós precisamos utilizar a palavra reservada `import`.

Para quem não conhece, um `module loader` é uma biblioteca que nos permite trabalhar com os modules nos nossos projetos. As bibliotecas mais conhecidas são: `CommonJs e Require.js`.

Na parte prática deste livro, nós utilizaremos os modules para organização do nosso código, mas, para que você já possa ter uma base de como utilizá-lo, vamos criar alguns exemplos agora.

Alterando o nosso exemplo anterior com os namespaces em arquivos separados, utilizando os modules nós teríamos o seguinte resultado:

```
//conta.ts
export class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;
    /* outras implementações */
}
```

```
//contaInvestimento.ts
import { Conta } from "./Conta";
export class ContaInvestimento extends Conta { }

//contaSalario.ts
import { Conta } from "./Conta";
export class ContaSalario extends Conta { }

//index.ts
import { ContaInvestimento } from "./contaInvestimento";
import { ContaSalario } from "./contaSalario";

let novaContaInvestimento = new ContaInvestimento();
let novaContaSalario = new ContaSalario();
```

Note que eu removi todos os namespaces do exemplo anterior e, para importar os nossos modules `Conta`, `ContaSalario` e `ContaInvestimento`, estou utilizando a palavra reservada `import`.

8.3 MODULES OU NAMESPACES? QUANDO UTILIZAR?

Agora que nós conhecemos os modules e os namespaces vem aquela dúvida, quando utilizar um ou o outro? Qual é a principal diferença entre eles?

Para responder a essas e outras perguntas sobre esse assunto, a seguir você tem alguns pontos sobre cada um:

Começando com os namespaces:

- Trabalham com escopo global, por esse motivo não precisamos importá-los como nos modules.
- A organização dos namespaces é mais lógica e feita em objetos, independente de estar em um ou mais arquivos.

- Um namespace pode ser utilizado em mais de um arquivo, como vimos nos exemplos.
- Eles não precisam de nenhuma *biblioteca* para ser utilizado.

Agora falando nos modules, temos:

- Não ficam no escopo global, por esse motivo precisamos utilizar a palavra reservada `import` .
- Toda implementação se torna um module separado.
- Eles precisam de uma *biblioteca*, como a `CommonJs` ou a `Require.js` para serem utilizados.

Com isso, nós finalizamos mais um capítulo. No próximo, nós iniciaremos a parte prática deste livro. Mãos na massa!

Capítulo 9

VISUAL STUDIO CODE

Como vimos na introdução do livro, para a parte prática vamos utilizar o Visual Studio Code, ou VS Code pela sua integração com o TypeScript.

Caso você ainda não o tenha instalado, basta acessar o link: https://code.visualstudio.com/Download e fazer o download de uma versão de acordo com o seu sistema operacional, Windows, Linux ou Mac.

Caso esse seja o seu primeiro contato com o VS Code, a seguir estou destacando alguns pontos que podem otimizar o seu trabalho com ele. Caso você já o utilize no seu dia a dia, recomendo que pule para o próximo capítulo.

Seguindo a documentação oficial do VS Code, nós podemos dividi-lo em cinco partes:

- Editor — Área onde editamos os arquivos. O VS Code nos permite abrir mais de um arquivo lado a lado.
- Side Bar — Podemos adicionar alguns plugins ou, como default, deixar o plugin do Git que fica monitorando as nossas alterações.
- Status Bar — Informação sobre o projeto aberto na solução.

- Activity Bar — A meu ver, depois do Editor Groups, essa é a parte que nós mais utilizamos. Nela temos um painel que podemos abrir e fechar, um botão de *search*, um controlador de versão e o melhor, um local para que possamos gerenciar os nossos plugins.

- Panels — No painel, nós podemos monitorar os erros de *debug*, temos o *output* e o melhor, podemos trabalhar com um ou mais terminais sem precisar utilizar o CMD do Windows ou terminal do Linux/MAC. Essa é uma das funcionalidades que eu acredito que a maioria dos desenvolvedores front-end e back-end mais utiliza.

A seguir você tem uma imagem destacando cada uma das partes:

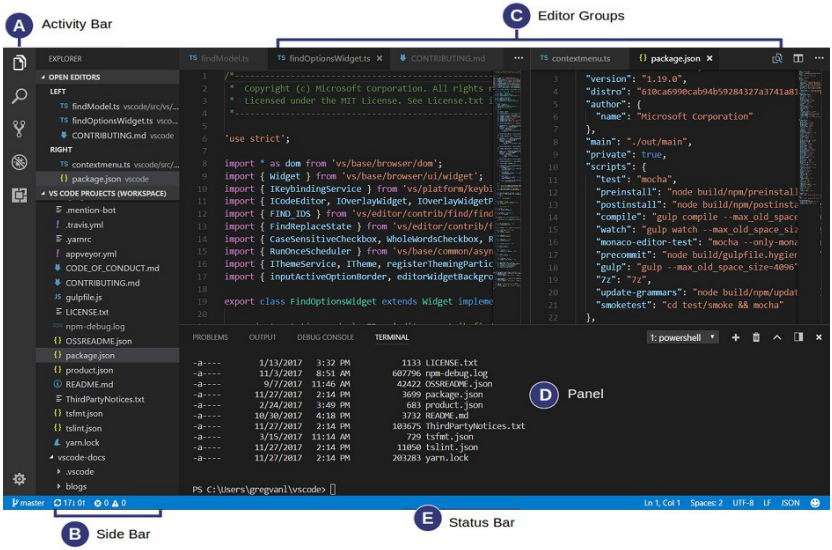

Figura 9.1: Visual Studio Code.

Atalhos

Como toda IDE, o VS Code tem muitos atalhos. Para ver uma lista com todos eles, vá até *file -> preferences -> Keyboard Shortcuts* ou utilize um atalho (no Windows): `Ctrl + k Ctrl + S` . A seguir você tem uma imagem mostrando a lista de atalhos que nós temos no VS Code.

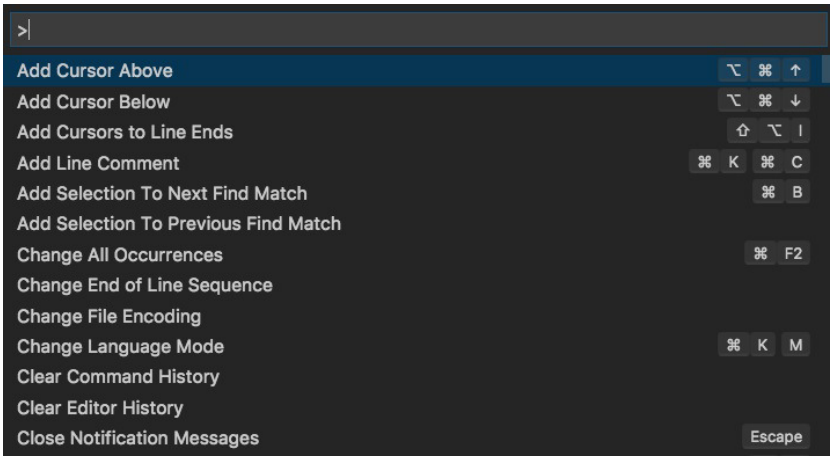

Figura 9.2: Visual Studio Code - Atalhos.

Aproveito para destacar os que eu mais utilizo no meu dia a dia, pois pode facilitar para você:

- `ctrl + f` : busca no arquivo que está em foco.
- `ctrl + p` : busca um arquivo dentro da `solution` .
- `ctrl + shift + p` : para executar algum comando da IDE, por exemplo, desabilitar todos os breakpoints.
- `ctrl + shift + g` : mostra todas as alterações feitas no projeto no seu repositório local.

IntelliSense

O IntelliSense é uma ajuda de preenchimento de código que inclui inúmeras funcionalidades: *Listar Membros*, *Informações do Parâmetro*, *Informações Rápidas* e *Completar Palavra*.

Essas funcionalidades ajudam você a aprender mais sobre o código que está usando, a manter o acompanhamento dos parâmetros que está digitando e a adicionar chamadas a métodos e propriedades pressionando apenas algumas teclas.

A seguir você tem uma imagem demonstrando essa funcionalidade:

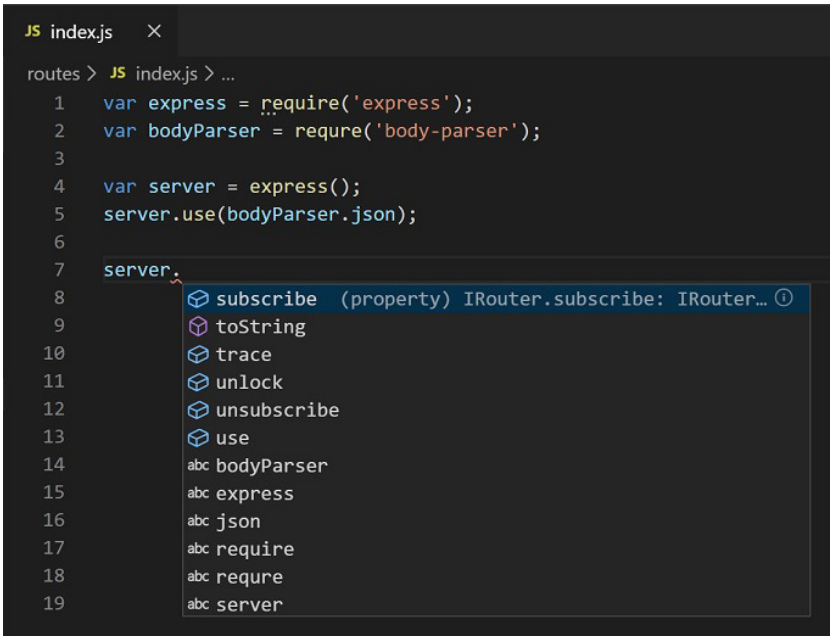

Figura 9.3: Visual Studio Code - IntelliSense.

Snippets

Os snippets são blocos de códigos que podem ser utilizados para ajudar a agilizar o trabalho de quem desenvolve. A ideia é facilitar a utilização de fragmentos de códigos repetitivos em diversas partes da aplicação que está sendo desenvolvida.

No VS Code, os snippets aparecem junto do IntelliSense. Utilizando o `Ctrl+Space` você tem um retorno com algumas sugestões de preenchimento de código, como chamada a um método de uma classe ou a criação de uma `function` completa. A seguir você tem um exemplo de um snippet no VS Code.

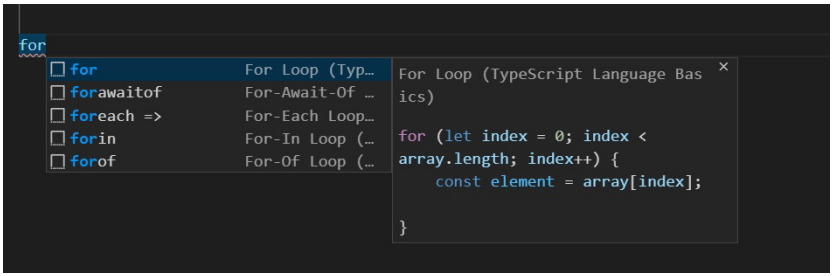

Figura 9.4: Visual Studio - Code Snippets.

JSDoc support

O IntelliSense no VS Code exibe informações que você pode adicionar a um script usando comentários *JSDoc* padrão. Para quem não conhece, o JSDoc é uma linguagem de marcação que podemos utilizar para documentar código JavaScript.

Ele nos permite usar comentários para fornecer informações sobre elementos de código, como funções, campos e variáveis.

A seguir você tem as marcas que podem ser utilizadas para

adicionar informações do seu código utilizando o JSDoc:

- `@deprecated` : especifica uma função ou um método preterido.
- `@description` : especifica a descrição de uma função ou um método.
- `@param` : especifica informações para um parâmetro em uma função ou método. O TypeScript também dá suporte a `@paramTag` .
- `@property` : especifica informações, incluindo uma descrição para um campo ou membro definido em um objeto.
- `@returns` : especifica um valor de retorno.
- `@summary` : especifica a descrição de uma função ou método.
- `@type` : especifica o tipo para uma constante ou uma variável.
- `@typedef` : especifica um tipo personalizado.

Para ficar mais claro, vamos ver um exemplo do uso das marcas `@description` e `@param` utilizando um trecho de código retirado do nosso capítulo 4 sobre Orientação a Objetos.

```
/** @description método criado para adicionar saldo para um clien
te.
 * @param {number} saldo parametro criado para adicionar um bonus
 no saldo de um cliente.
 */
 protected adicionaSaldo(saldo: number): void {
        this.saldo + saldo;
    }
```

Resultado:

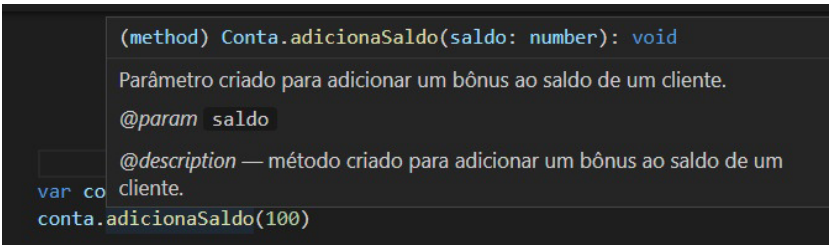

Figura 9.5: Visual Studio - JSDoc.

Auto imports

Quando você está importando uma biblioteca de terceiros ou um módulo do seu projeto, o VS Code informará que ele não conhece essa classe ou método e vai percorrer pela sua solução, *seu projeto*, e procurar esse código retornando algumas sugestões de import para que você possa importar esse módulo, script ou biblioteca.

Para ficar mais claro, vamos voltar ao capítulo anterior sobre namespaces. Caso você tente chamar o namespace `Banco` de um outro arquivo, o VS Code retornará o seguinte erro:

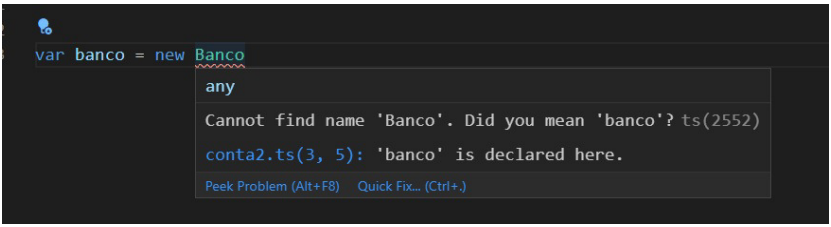

Figura 9.6: Visual Studio - Erro de referência.

E caso você clique na lâmpada azul, que está no canto superior esquerdo da imagem anterior, a IDE vai sugerir alguns imports para que você possa adicioná-los ao arquivo que está tentando

referenciar. No nosso exemplo anterior, ele vai retornar as referências para o namespace `Banco`.

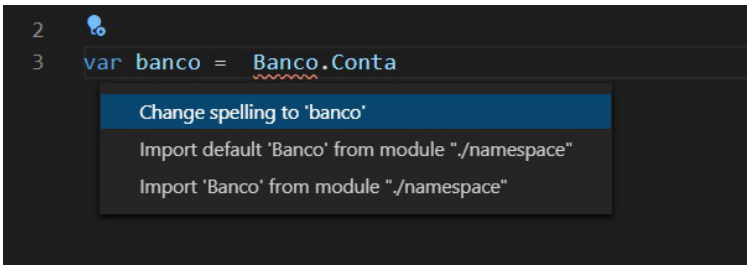

Figura 9.7: Visual Studio - Auto Import.

Code navigation

O *Code navigation*, ou, em português, navegação de código, nos permite navegar pelo código de uma maneira mais rápida. A seguir você tem uma lista de teclas que podem nos ajudar a navegar pelo nosso código.

- `F12` : leva-nos para a definição do nosso código.
- `Alt + F12` : leva-nos para o ponto anterior à navegação com o F12.
- `Shift + F12` : mostra todos os lugares no nosso código que estão referenciando um determinado trecho de código.
- `Ctrl + F12` : leva-nos para a implementação de uma interface.

Rename

No VS Code, quando pressionamos o *F2* em cima de um método, nós conseguimos renomeá-lo, bem como todas as suas referências.

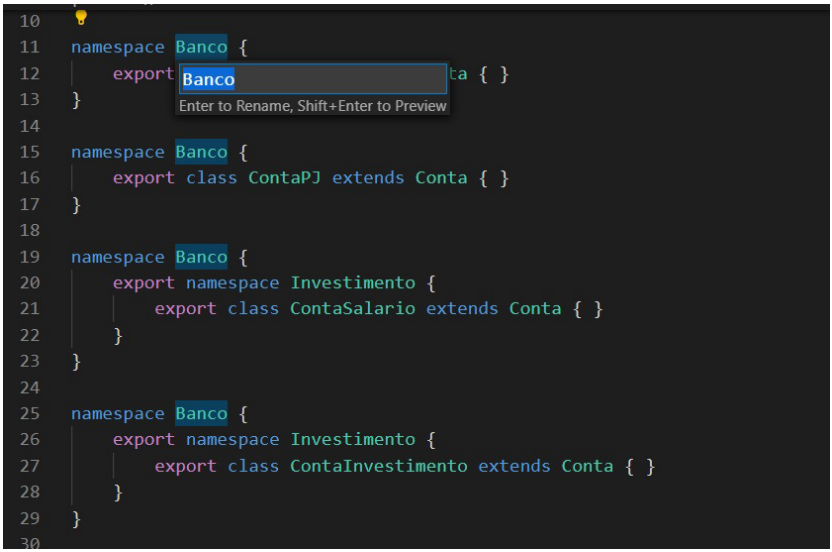

Figura 9.8: Visual Studio - Rename.

Refactoring

O VS Code tem algumas sugestões de refatoração rápida, como para extrair uma classe ou interface para um outro arquivo. Para que você possa ver as sugestões de refatoração, basta pressionar `Control + .`.

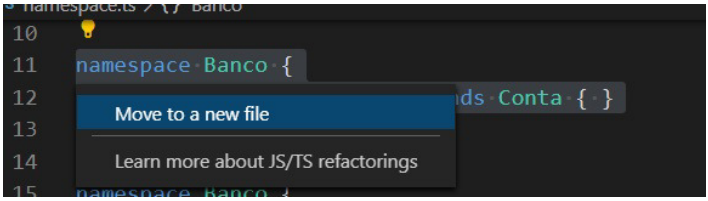

Figura 9.9: Visual Studio - Refactoring.

References CodeLens

Quando habilitado *CodeLens* no VS Code, ele mostra todas as referências das suas classes, interfaces, métodos, propriedades, e os objetos exportados.

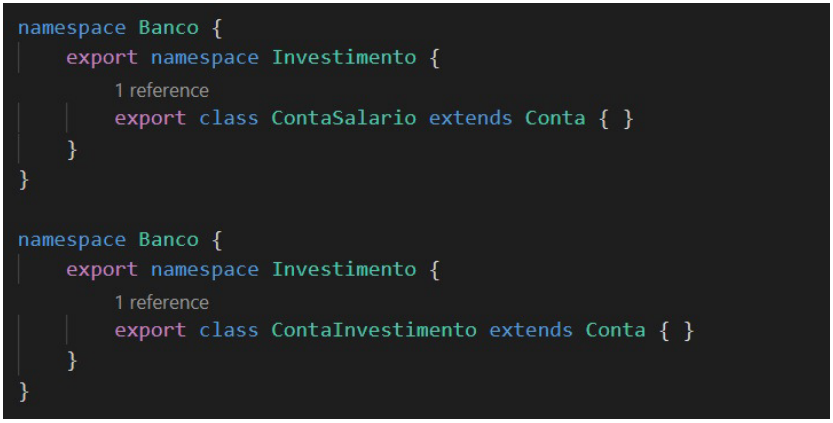

Figura 9.10: Visual Studio - CodeLens.

Para habilitá-lo é bem simples, basta utilizar o atalho `Control + Shift + P`, abrir o arquivo `Settings.js` e adicionar a seguinte linha a ele: `typescript.referencesCodeLens.enabled": true`.

Debugging

O suporte a *debug* é um dos motivos pelo qual eu sempre recomendo a utilização do VS Code para o desenvolvimento de projeto com TypeScript. Ele tem suporte para projetos desenvolvidos no back-end utilizando Node.js com TypeScript ou no front-end em projeto TypeScript com Angular, por exemplo.

Para fazer debug de um projeto, basta clicar no canto esquerdo da IDE no ícone de *Play* com uma *baratinha*.

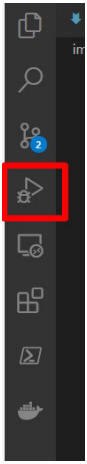

Figura 9.11: Visual Studio - debug.

TypeScript extensions

Para finalizar este capítulo, o VS Code tem muitos plugins que podem ajudar no nosso dia a dia. Para explorar essa lista, basta clicar no quarto ícone do menu lateral. A seguir você tem uma imagem demonstrando alguns plugins que nós podemos baixar para trabalhar com TypeScript:

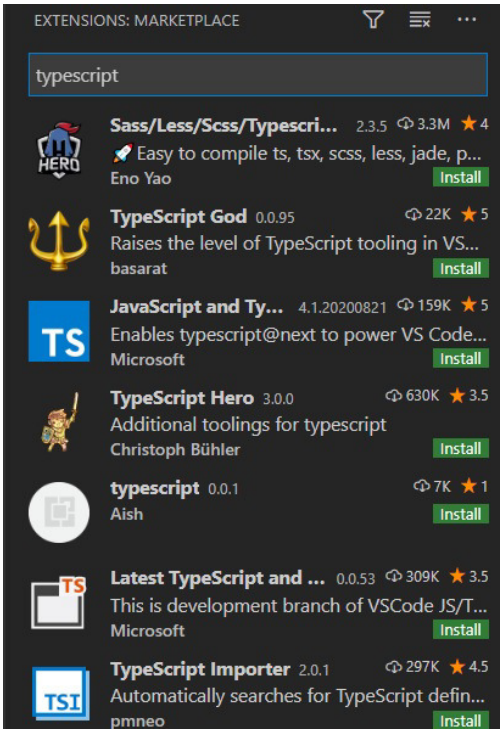

Figura 9.12: Visual Studio - plugins.

Com isso, finalizamos mais este capítulo. No próximo, vamos iniciar o desenvolvimento de uma API.

Capítulo 10

DOCKER: CONFIGURANDO AMBIENTE DE BANCO DE DADOS

Neste capítulo, configuraremos um ambiente de desenvolvimento com Docker para a criação do nosso banco de dados.

A ideia aqui não será aprofundar no que é o Docker e nem em como ele funciona por baixo dos panos. Mas vamos passar rapidamente por alguns conceitos básicos, para que possamos montar o nosso ambiente de desenvolvimento.

Por que eu escolhi o Docker? Inicialmente pensei em utilizar uma plataforma pronta como o Azure com o Cosmos DB, mas como estamos criando uma aplicação do zero, seria legal aproveitarmos para ver algo que está sendo muito utilizado pela comunidade e requisitado por muitas empresas como conhecimento necessário para processos seletivos.

Se você já conhece o Docker, pode pular para o final deste capítulo. Agora, caso esse seja o seu primeiro contato com ele,

acompanhe o passo a passo a seguir.

10.1 DOCKER

O Docker é um projeto open source escrito na linguagem Go, que torna a criação e o gerenciamento de contêineres muito mais fácil.

Um contêiner (*container*) é construído utilizando alguns recursos do *kernel*, como namespaces, cgroups, chroot, que permitem que ele seja criado e rode como um processo isolado dentro do nosso Sistema Operacional.

Atualmente, o Docker está dividido em dois produtos, *Community Edition (CE)* e *Enterprise Edition (EE)*. Como os nomes sugerem, a versão Community é gratuita e voltada para a comunidade, enquanto a Enterprise é recomendada para uso empresarial. Para o nosso exemplo, vamos utilizar a versão grátis, a Community.

O nosso primeiro passo para trabalhar com Docker será a criação de uma conta no portal Docker Hub. Para isso, acesse o link https://hub.docker.com/signup e siga os passos necessários para o preenchimento do formulário de criação de uma nova conta.

Caso você já tenha uma conta criada, clique no link https://id.docker.com/login/ e acesse com o seu *docker ID* e sua *senha*. Com o passo da conta OK e já logado no portal, clique em *Download* e instale-o no seu computador.

Comandos básicos

Junto com o pacote de instalação do Docker vem uma CLI (Interface de Linha de Comando), que nós podemos acessar através de um terminal no nosso computador. Vamos conhecer alguns comandos básicos que devem ajudar no seu dia a dia.

Abra um terminal no seu computador e digite o seguinte comando nele:

```
docker info
```

```
Resultado
```

```
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 19.03.8
 Storage Driver: overlay2
  Backing Filesystem: <unknown>
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local loge
ntries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc version: dc9208a3303feef5b3839f4323d9beb36df0a9dd
 init version: fec3683
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 4.19.76-linuxkit
```

```
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 2
 Total Memory: 1.945GiB
 Name: docker-desktop
 ID: EPCH:EZQW:LGNN:SWMC:RSAL:NYFO:QR4C:GM6L:646G:D2FX:A4CC:ZS6Q
 Docker Root Dir: /var/lib/docker
 Debug Mode: true
  File Descriptors: 38
  Goroutines: 49
  System Time: 2020-08-29T20:30:28.742899774Z
  EventsListeners: 3
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
 Product License: Community Engine
```

Como você pode observar, esse comando retorna todas as informações relacionadas ao Docker Host instalado no nosso computador, desde a versão instalada até a quantidade de `images` instaladas.

Caso você precise de um comando rápido para ver a versão que está instalada no seu ambiente, basta executar o comando `docker version` no seu terminal.

```
 Resultado

Client: Docker Engine - Community
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.12.17
 Git commit:        afacb8b
 Built:             Wed Mar 11 01:23:10 2020
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
```

```
Engine:
 Version:          19.03.8
 API version:      1.40 (minimum version 1.12)
 Go version:       go1.12.17
 Git commit:       afacb8b
 Built:            Wed Mar 11 01:29:16 2020
 OS/Arch:          linux/amd64
 Experimental:     false
containerd:
 Version:          v1.2.13
 GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
runc:
 Version:          1.0.0-rc10
 GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
docker-init:
 Version:          0.18.0
 GitCommit:        fec3683
```

Um ponto importante de se destacar nesse resultado é o `OS/Arch`. Note que eu estou em um ambiente Windows e, no `OS/Arch` que está em `Server: Docker Engine - Community`, está retornando `linux/amd64`. Isso acontece porque eu estou em um ambiente Windows trabalhando com imagens Linux.

Isso é muito importante de se destacar, pois a maioria das imagens disponíveis no Docker Hub foram criadas para rodar em ambiente Linux. Para utilizá-las em um ambiente com o SO Windows, nós precisamos alterar o `OS/Arch` conforme está no retorno anterior.

Fazer isso é bem simples, basta clicar na baleia, que está na sua barra de tarefas com o botão direito do seu mouse, e em *Switch to Linux Containers*.

Falamos anteriormente de imagens em ambiente Linux e Windows, mas não falamos o que seria uma imagem. As `images` Docker são compostas por sistemas de arquivos em camadas que

ficam umas sobre as outras. Elas são a base para a construção de uma aplicação.

Nós podemos criar as nossas imagens a partir de outras imagens e versioná-las em alguns portais, como Docker Hub ou Azure Container Registry.

Para que esse passo fique mais claro, vamos criar um contêiner utilizando uma imagem do MongoDB. Para isso, execute o seguinte comando no seu terminal:

```
docker pull tutum/mongodb
```

O comando `pull` baixa uma imagem para o seu computador. Para listar todas as imagens que estão no seu computador, basta executar o comando `docker images` .

```
Resultado

REPOSITORY                          TAG                 IMAGE
ID            CREATED             SIZE
tutum/mongodb                       latest              64ca95
21c703        4 years ago         503MB
```

Com a imagem do MongoDB no nosso computador, vamos criar um contêiner Docker para a nossa base de dados. Para isso, execute o seguinte comando no seu terminal:

```
docker run -d -p 27017:27017 -p 28017:28017 -e AUTH=no tutum/mong
odb
```

Analisando esse comando nós temos:

- `run` : esse comando é utilizado para a criação de um contêiner.
- `-d` : esse parâmetro está informando que o contêiner tem que rodar em background.

- `-p` : devemos utilizar esse parâmetro para especificar a porta que vamos utilizar para conectar no nosso contêiner. A porta que está antes dos `:` é a do nosso host e a que está depois é a que vamos utilizar para acessar o nosso contêiner.

Para verificar se o seu contêiner foi criado corretamente, basta abrir um terminal no seu computador e executar o comando `docker ps` nele.

```
Resultado

CONTAINER ID        IMAGE               COMMAND             CREAT
ED              STATUS              PORTS
                    NAMES
754b9c8b7e4f        tutum/mongodb       "/run.sh"           4 sec
ond ago         Up 4 seconds        0.0.0.0:27017->27017/tcp, 0.0.(
.0:28017->28017/tcp   elastic_brahmagupta
```

Esse comando lista os contêineres que estão em execução no seu computador. Agora, para que possamos acessar o MongoDB dentro do contêiner e criar uma nova base de dados, vamos utilizar um cliente de acesso a banco de dados mongo, o `Robo 3T` .

Caso você não o tenha instalado, segue link para download do seu instalador: https://robomongo.org/.

Com ele instalado no seu computador, clique nos dois monitores que ficam no canto direito da ferramenta, em seguida, em `create` e preencha os campos com os seguintes dados:

- `Name` : nome da sua conexão. Para esse exemplo vou utilizar `db_portal` .
- `Address` : deixe os dados default `localhost` na porta `27017` .

Com os dados preenchidos, clique em *save* e depois em *connect*. Agora que estamos conectados ao nosso servidor de banco de dados, vamos criar uma nova base de dados chamada `db_portal`.

Essa base será utilizada no próximo capítulo deste livro. Clique com o botão direito em *localhost*, em seguida, em *Create database* e, no campo *Database Name:*, preencha com o nome `db_portal`. Agora clique em *create* para que a base seja criada.

Antes de darmos o próximo passo, é muito importante todos estarmos familiarizados com alguns termos utilizados quando trabalhamos com uma base de dados MongoDB. A seguir você tem uma breve descrição de alguns que utilizaremos neste livro:

- *database*: base de dados.
- *collections*: para quem trabalha ou já teve contato com um banco de dados SQL, seriam as nossas tabelas, mas pensando em quem está começando na área, uma collection seria uma coleção de dados.
- *fields*: seriam as propriedades das nossas classes, que quando forem mapeadas para a nossa collection serão os campos onde armazenaremos os valores.

Para finalizar este capítulo, vamos criar uma nova collection chamada *news* dentro do nosso database `db_portal`. Para isso, clique com o botão direito em *Collections* e em *Create Collection*. Preencha o campo *Collection Name:* com o valor `news`.

Com a criação do nosso contêiner MongoDB e a criação do nosso database `db_portal`, nós podemos seguir para o próximo capítulo, onde daremos início à construção da nossa API.

Capítulo 11

CRIANDO API TYPESCRIPT, NODE.JS, MONGODB E DOCKER

11.1 ARQUITETURA BÁSICA DO PROJETO

Neste capítulo, nós colocaremos em prática alguns conceitos que aprendemos no decorrer do livro. Utilizaremos os conceitos de tipagem básica, interfaces e até Programação Orientada a Objetos. Desenvolveremos uma API utilizando as tecnologias TypeScript, Node.js, MongoDB e o Docker, que abordamos no capítulo anterior.

Para que você possa ter um exemplo real de como se trabalhar com essas tecnologias, passaremos por todos os passos utilizados para desenvolver uma API que retorna as últimas notícias do site Masterchef Brasil, em uma de suas edições.

Criando os arquivos de configuração básica

Já vamos começar direto partindo para a parte prática. Nosso primeiro passo será abrir um terminal em seu diretório de preferência e digitar o seguinte comando nele: `mkdir api_mc .`

Esse comando deve criar um novo diretório com o nome `api_mc`.

Conforme vimos no capítulo de introdução, para trabalhar com TypeScript nós precisamos criar o arquivo `tsconfig.json`. Para isso, execute o seguinte comando no seu terminal: `tsc --init`.

O comando `tsc --init` cria o arquivo `tsconfig.json` com algumas configurações padrões. Para que possamos desenvolver a nossa API, será necessário fazer alguns ajustes nesse arquivo. Abra-o em seu editor de textos e atualize-o com o seguinte trecho de código:

```
{
    "compilerOptions": {
        "target": "es6",
        "module": "commonjs",
        "esModuleInterop": true,
        "outDir": "dist",
        "typeRoots": [
            "../node_modules/@types"
        ],
        "types": [
            "node"
        ]
    },
    "include": [
        "**/*.ts",
        "*.ts",
    ],
    "exclude": [
        "node_modules"
    ]
}
```

O que alteramos nele? Dentro de `compilerOptions` nós passamos:

- `target`: versão do ECMAScript que vamos utilizar no

projeto.

- `module` : biblioteca que vamos utilizar para trabalhar com os módulos.
- `esModuleInterop` : permite importarmos os módulos `CommonJS` em conformidade com as especificações do `target es6` .
- `outDir` : diretório de destino onde serão transpilados os nossos arquivos.
- `typeRoots` : caminho dos pacotes `@types` que nós importaremos para trabalhar com TypeScript; veremos com mais detalhes no decorrer do capítulo.

Fora de `compilerOptions` , temos:

- `include` : local onde o compilador deve procurar os nossos arquivos para fazer o `transpile` ;
- `exclude` : os diretórios que devem ser ignorados no momento do `transpile` .

Com o arquivo `tsconfig.json` atualizado, vamos criar o nosso arquivo `package.json` . Digite o seguinte comando no seu terminal: `npm init -y` .

Caso esse seja o seu primeiro contato com esse arquivo, o `package.json` armazena algumas informações do nosso projeto, como nome, versão, descrição, o arquivo de inicialização e os pacotes que foram adicionados ao projeto.

Com os arquivos `tsconfig.json` e `package.json` criados, o próximo passo será a instalação dos pacotes necessários para o desenvolvimento da nossa API. Para isso, execute os seguintes comandos no seu terminal:

```
npm i  typescript  express  mongoose  --save
npm i  @types/express @types/mongoose --save-dev
```

Analisando esses pacotes, nós temos:

- `typescript` : biblioteca necessária para que possamos informar a versão do TypeScript que está no nosso projeto.
- `express` : framework utilizado para desenvolvimento de aplicativo web do Node.js.
- `mongoose` : pacote necessário para que possamos trabalhar com o banco de dados MongoDB.

Note que, na segunda linha, nós temos os mesmos pacotes, só que com o prefixo `@type` . Para quem não conhece esses pacotes, eles são interfaces do TypeScript que nos permitem trabalhar com as bibliotecas JavaScript. Como já sabemos, os navegadores não interpretam códigos TypeScript, então precisamos dessas interfaces para trabalhar no nosso ambiente de desenvolvimento.

Para ficar mais claro, navegue até o diretório `node_modules\@types\express` do seu projeto, abra o arquivo `index.d.ts` no seu editor de textos e note que dentro dele nós temos algumas interfaces:

```
 interface Application extends core.Application { }
    interface CookieOptions extends core.CookieOptions { }
    interface Errback extends core.Errback { }
    interface ErrorRequestHandler<P = core.ParamsDictionary, ResB
ody = any, ReqBody = any, ReqQuery = core.Query>
        extends core.ErrorRequestHandler<P, ResBody, ReqBody, Req
Query> { }
    interface Express extends core.Express { }
    interface Handler extends core.Handler { }
    interface IRoute extends core.IRoute { }
    interface IRouter extends core.IRouter { }
    interface IRouterHandler<T> extends core.IRouterHandler<T> {
}
```

```
    interface IRouterMatcher<T> extends core.IRouterMatcher<T> {
}
    interface MediaType extends core.MediaType { }
    interface NextFunction extends core.NextFunction { }
    interface Request<P = core.ParamsDictionary, ResBody = any, R
eqBody = any, ReqQuery = core.Query> extends core.Request<P, ResB
ody, ReqBody, ReqQuery> { }
    interface RequestHandler<P = core.ParamsDictionary, ResBody =
 any, ReqBody = any, ReqQuery = core.Query> extends core.RequestH
andler<P, ResBody, ReqBody, ReqQuery> { }
    interface RequestParamHandler extends core.RequestParamHandle
r { }
    export interface Response<ResBody = any> extends core.Respons
e<ResBody> { }
    interface Router extends core.Router { }
    interface Send extends core.Send { }
```

Caso você abra o pacote `express`, que fica em `\node_modules\express`, vai encontrar a implementação dessas interfaces em JavaScript.

Dando continuidade ao desenvolvimento do nosso projeto, para validar se todos pacotes foram importados corretamente, abra o seu arquivo `package.json` e observe se ele tem duas novas propriedades: uma chamada `dependencies`, para os pacotes `JavaScript` que foram importados na primeira linha com `--save`, e outra chamada `devDependencies`, para as interfaces TypeScript que foram importadas utilizando o sufixo `--save-dev`.

A seguir você tem o arquivo `package.json` atualizado com as dependências necessárias para o desenvolvimento da nossa API.

```
{
  "name": "api_mc",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
    "mongoose": "^5.12.2",
    "typescript": "^4.2.3"
  },
  "devDependencies": {
    "@types/express": "^4.17.11",
    "@types/mongoose": "^5.10.4"
  }
}
```

Aproveitando que estamos analisando o arquivo `package.json`, vamos atualizar a parte de `scripts` com um trecho de código que vamos utilizar para executar a nossa API:

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "compile": "tsc -w",
    "start": "node ./dist/program.js"
  },
```

Nesse código, temos duas instruções:

- `compile`: esse parâmetro já foi visto na introdução do livro. Ao adicionar o `-w` depois da instrução `tsc`, o compilador passa a monitorar os arquivos `.ts` do nosso projeto e, em caso de alteração, ele faz o transpile dos nossos arquivos.
- `start`: aqui estamos passando qual será a instrução de inicialização da nossa API.

Neste momento não se preocupe com o arquivo `program.js`, nós o criaremos nos próximos passos.

Criando a estrutura básica do projeto

Com os pacotes importados, criaremos agora a estrutura básica do nosso projeto. Para isso, crie os seguintes diretórios:

- `models` : nesse diretório, nós criaremos as nossas models. Vamos entender melhor o que são essas classes nos próximos passos.
- `repository` : aqui nós vamos mapear as nossas models com as nossas collections do MongoDB.
- `contracts` : nesse diretório, nós criaremos as nossas interfaces.
- `services` : nesse diretório, nós criaremos a comunicação com o nosso banco de dados.
- `controller` : nesse diretório, nós criaremos a entrada do nosso projeto.
- `infra` : nesse diretório, nós criaremos alguns arquivos de configuração, como um arquivo de conexão com banco de dados.

Com a estrutura básica criada e os pacotes necessários importados, vamos iniciar o desenvolvimento da nossa API.

11.2 DESENVOLVIMENTO DA API

O primeiro passo para o desenvolvimento de um projeto é a definição de suas models. As models são a representação de um conjunto de informações sobre determinado conceito do sistema. Toda model possui atributos, que são as informações que a referenciam.

Para ficar mais claro, vamos voltar ao capítulo 4 sobre

Programação Orientada a Objetos. Você se lembra da classe `Conta`? Ela pode ser vista como uma model ou entidade, como é conhecida na modelagem de software, e tem os seguintes atributos: `numeroDaConta`, `titular` e `saldo`. A definição das models é um conceito básico, porém muito importante na modelagem de um software.

Agora que sabemos o que é uma model, vamos criar uma para a nossa listagem de notícias. Para isso, nós precisamos responder à seguinte pergunta: *O que toda notícia tem?* Em uma breve análise nós temos: título, chapéu, texto, autor, imagem, data de publicação, tags, um link para a notícia e um atributo para dizer se ela está ativa ou não.

Com a definição dos atributos da nossa model, o próximo passo será criá-la. Crie um novo arquivo chamado `newsSchema.ts` dentro do diretório `models` e atualize-o com o seguinte trecho de código:

```
import mongoose from "mongoose";

export const NewsSchema = new mongoose.Schema({
  titulo: { type: String },
  chapeu: { type: String },
  texto: { type: String },
  autor: { type: String },
  imagem: { type: String },
  dataPublicacao: { type: Date },
  tags: { type: String },
  link: { type: String },
  ativo: { type: Boolean }
});
```

O código está simples, estamos exportando uma constante chamada `NewsSchema`, que está instanciando `Schema` do pacote `mongoose`, em seguida, passamos os atributos, que nós mapeamos

no passo anterior, para a criação da nossa model.

Com `NewsSchema` criada, o próximo passo será mapeá-la com o nosso banco de dados. Para isso, nós precisamos criar o nosso `repository`.

Como vimos anteriormente, o `repository` é responsável por mapear as nossas models com o banco de dados. Como estamos utilizando MongoDB, a nossa classe `NewsRepository` vai mapear a nossa classe `NewsSchema` com uma collection.

Para ficar mais claro, crie um novo arquivo chamado `newsRepository.ts` dentro do diretório `repository` e atualize-o com o seguinte trecho de código:

```
import mongoose from "mongoose";
import { NewsSchema } from "../models/newsSchema";

export const NewsRepository = mongoose.model("news", NewsSchema);
```

Em um breve resumo, estamos dizendo ao pacote `mongoose` que ele deve criar uma collection chamada `news`, com os dados nossa model `NewsSchema`.

Trabalhando com interfaces

Com o mapeamento OK, o próximo passo será a criação de um arquivo de contrato, algo que diga o que os nossos serviços devem ter. Voltando ao capítulo 5, sobre interfaces, sabemos que esse é um dos papéis de uma interface.

Crie um arquivo chamado `iNewsService.ts` dentro do diretório `contracts` e atualize-o com o seguinte trecho de código:

```
import { Result } from "../infra/result";

export interface INewsService {

    get(id: string);

    getAll(page: number, qtd: number): Promise<Result>;
}
```

Na interface `INewsService`, nós temos dois métodos: um com parâmetro `Id` recebendo um número e um outro recebendo dois parâmetros, `page` e `qtd`, retornando uma `promise` de `Result` (classe que vamos criar a seguir).

Para quem não conhece uma `promise`, como o próprio nome diz, uma `promise` é uma promessa. A sua ideia principal é representar fluxos assíncronos de forma sequencial, além de favorecer o tratamento de exceções.

A seguir você tem os estados que nós podemos utilizar quando trabalhamos com um objeto `promises`:

- `Resolve`: quando ela retorna o valor esperado.
- `Reject`: no caso de algum erro, ela retorna que essa chamada foi rejeitada.
- `Pending`: quando ela ainda está esperando o valor ou o resultado final da requisição.

Agora para resolver o erro de referência que está dando na interface `INewsService`, vamos criar a classe `Result`. Para isso, crie um novo arquivo chamado `result.ts` dentro do diretório `infra` e atualize-o com o seguinte trecho de código:

```
export class Result {
    Qtd: number;
    Page: number;
```

```
    Total: number;
    Data: any
}
```

A utilização da classe `Result` é uma boa prática no desenvolvimento de APIs. Ela nos permite ter um padrão de retorno como: qual é a quantidade de dados que estamos retornando, `Qtd`, a página em que estamos (pensando em listagem de resultados), `Page`, o total de registros que nós temos salvos no nosso banco de dados para as próximas páginas, `Total`, e o resultado, em `Data`.

Criando as nossas pesquisas

Agora que nós criamos o nosso contrato, o próximo passo será a criação de uma classe para o implementarmos. Para isso, crie um novo arquivo chamado `newsService.ts` dentro do diretório `services` e atualize-o com o seguinte trecho de código:

```
import { INewsService } from "../contracts/iNewsService";
import { Result } from "../infra/result";
import { NewsRepository } from "../repository/newsRepository";

export class NewsService implements INewsService {

    async get(_id: string) {
        let result = await NewsRepository.findById(_id);
        return result;
    }

    async getAll(page: number, qtd: number): Promise<Result> {
        let result = new Result();
        result.Page = page;
        result.Qtd = qtd;
        result.Total = await NewsRepository.count({});
        result.Data = await NewsRepository.find({}).skip((page *
qtd) - qtd).limit(qtd);
        return result;
    }
```

```
}
```

Analisando o trecho de código, nós temos:

- No início do arquivo, nós estamos implementando a interface `INewsService` e importando a classe `NewsRepository`. Como vimos anteriormente, essa classe é responsável pela integração com o nosso banco de dados.
- Em seguida, nós estamos implementando os métodos que estão no contrato/interface `INewsService`.

Note que temos duas novas palavras reservadas no trecho em `NewsService`, a `async` e a `await`. Para quem não as conhece, quando declaramos uma função com `async`, estamos passando que essa função é assíncrona e que ela não deve bloquear o segmento principal do nosso código. A palavra `await` é utilizada dentro de uma função `async`, então adicioná-la faz com que o fluxo da função assíncrona não seja interrompido esperando pelo retorno da função.

Criando a nossa Controller

Agora que nós já implementamos o nosso contrato, o próximo passo será a criação de um arquivo de entrada do nosso projeto. Para isso, crie um novo arquivo chamado `newsController.ts` dentro do diretório `controller` e atualize-o com o seguinte trecho de código:

```
import { NewsService } from "../services/newsService";
import { Request, Response } from "express";


class NewsController {
```

```
    private _service: NewsService;

    constructor() {
        this._service = new NewsService();
    }

    async get(request: Request, response: Response) {
        try {
            const page = request.params.page ? parseInt(request.p
arams.page) : 1;
            const qtd = request.params.qtd ? parseInt(request.par
ams.qtd) : 10;
            let result = await this._service.getAll(page, qtd);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }

    async getById(request: Request, response: Response) {
        try {
            const _id = request.params.id;
            let result = await this._service.get(_id);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }
}

export default new NewsController();
```

Analisando a nossa classe `NewsController` nós temos:

- No início do arquivo nós estamos importando a classe `NewsService` e instanciando-a dentro do `construtor`.
- Em seguida, nós criamos dois métodos `async`, um para consulta por `_id` e um outro para retornar uma listagem

paginada.

Note que nós temos duas novas palavras reservadas nessa classe, a `try` e a `catch`. Para quem está tendo o seu primeiro contato com elas neste livro, nós utilizamos `try/catch` para tratamento de exceções:

- `try`: consegue recuperar erros que possam ocorrer no código.
- `catch`: faz o tratamento dos erros que aconteceram, retorna uma `exception` que pode ser tratada e retornada no `response`.

Com o arquivo de entrada criado, o próximo passo será a criação de um arquivo de configuração para conexão com o banco de dados. Para isso, crie um novo arquivo chamado `db.ts` dentro do diretório `infra` e atualize-o com o seguinte trecho de código:

```
import mongoose from 'mongoose';

class Database {
    private DB_URL = "mongodb://localhost:27017/db_portal";

    createConnection() {
        mongoose.connect(this.DB_URL);
    }
}

export default Database;
```

Nesse arquivo, estamos passando a string de conexão do nosso banco de dados para o método `connect` do `mongoose`. Ele se encarregará de conectar com o banco de dados e criar as nossas transações.

11.3 ARQUIVO DE INICIALIZAÇÃO DO PROJETO

Agora que nós já temos a estrutura básica do nosso projeto, o próximo passo será a criação do arquivo de inicialização.

Existem algumas formas de criar esse arquivo. Eu já trabalhei em alguns projetos nos quais o time optou por deixar tudo em um arquivo chamado `app.ts`, trabalhamos com essa arquitetura por um tempo, mas depois de estudar outras arquiteturas, como a do .NET Core, eu resolvi separar esse arquivo de inicialização em dois arquivos: um chamado `startUp.ts` e um outro chamado `program.ts`.

Começando pelo arquivo `startUp.ts`, esse seria o arquivo de configurações do nosso projeto. É nele que devemos inicializar o `express`, o nosso arquivo de conexão com o banco de dados `db.ts` e a nossa classe `NewsController`. Para que você possa ter um melhor entendimento, vou separar a implementação da classe `startUp.ts` em algumas partes e explicar cada uma delas.

O primeiro passo será importar os pacotes necessários para as nossas configurações:

```
import express, { Application, Request, Response } from "express"
;
import database from "./infra/db";
import NewsController from "./controller/newsController";
```

Voltando ao capítulo 4 sobre Orientação a Objetos, vamos criar uma nova classe chamada `StartUp` com um `construtor` vazio e um método chamado `routes` para que possamos criar as rotas de entrada do nosso projeto.

Não se preocupe com o `construtor` nesse momento, nós vamos atualizá-lo nos próximos passos.

```
class StartUp {
    constructor() { }
    routes() { }
}
```

Agora crie duas novas propriedades, uma `public`, para trabalhar com a biblioteca do `express`, e uma outra `private`, para conexão com o banco de dados:

```
class StartUp {
    public app: Application;
    private _db: database = new database();

    //outras implementações
```

Em `app`, nós estamos passando `Application` do pacote `express` e, em `_db`, estamos instanciando a nossa classe `database`, para que possamos conectar o projeto ao nosso banco de dados. Com a configuração do `express` e a conexão à nossa base de dados OK, vamos criar as rotas de entrada do nosso projeto:

```
class StartUp {

  //outras implementações

    routes() {
        this.app.route("/").get((req, res) => {
            res.send({ versao: "0.0.1" });
        });

        this.app.route("/api/v1/news/:page/:qtd").get((req: Reque
st, res: Response) => {
            return NewsController.get(req, res);
        });

        this.app.route("/api/v1/news/:id").get((req: Request, res
```

```
: Response) => {
            return NewsController.getById(req, res);
        });
    }

   //outras implementações
}
```

Analisando o trecho de código acima nós temos:

- Em `this.app.route("/")`, nós estamos criando uma rota para testar a nossa API. O pessoal da comunidade chama essa rota de `Health Check`, ela geralmente é utilizada com outros serviços, como `zabbix`, `application insights` para monitorar se a API está funcionando corretamente.
- Em seguida, estamos criando duas novas rotas que devem chamar os dois métodos da nossa `NewsController`, o `get` e o `getById`.

Agora atualize o seu `construtor` com o seguinte trecho de código:

```
  constructor() {
        this.app = express();
        this._db.createConnection();
         this.routes();
     }
```

E para finalizar a configuração da classe `StartUp`, nós devemos exportá-la para que possamos chamá-la no nosso arquivo `program.ts`. Para isso, adicione o seguinte trecho de código no final do seu arquivo `startUp.ts`:

```
export default new StartUp();
```

A seguir você tem a classe `StartUp` completa:

```
import express, { Application, Request, Response } from "express"
;
import database from "./infra/db";
import NewsController from "./controller/newsController";

class StartUp {

    public app: Application;
    private _db: database = new database();

    constructor() {
        this.app = express();
        this._db.createConnection();
        this.routes();
    }


    routes() {
        this.app.route("/").get((req, res) => {
            res.send({ versao: "0.0.1" });
        });

        this.app.route("/api/v1/news/:page/:qtd").get((req: Reque
st, res: Response) => {
            return NewsController.get(req, res);
        });

        this.app.route("/api/v1/news/:id").get((req: Request, res
: Response) => {
            return NewsController.getById(req, res);
        });
    }
}

export default new StartUp();
```

Com a finalização da classe `StartUp`, crie o seu arquivo `program.ts` na raiz do seu projeto e atualize-o com o seguinte trecho de código:

```
import StartUp from "./startUp";

let port = "5000";
```

```
StartUp.app.listen(port, function () {
    console.log(`servidor rodando na porta: ${port}`);
});
```

Analisando o arquivo `program.ts` nós temos:

- Na primeira linha, estamos importando a nossa classe `StartUp` .
- Em seguida, estamos passando a porta em que o `express` deve executar a nossa aplicação.

Agora para que possamos validar se tudo está configurado corretamente, execute o comando `npm run compile` no seu terminal. Você se lembra dessa instrução? Nós a adicionamos ao arquivo `package.json` e vimos como ele funciona no capítulo de introdução do livro:

```
"scripts": {
   //outras implementaçoes
    "compile": "tsc -w",
  },
```

```
Resultado

> tsc -w
 Starting compilation in watch mode...
 Found 0 errors. Watching for file changes.
```

11.4 INCREMENTAL FLAG

Um ponto importante de se destacar nesse momento é que à versão *3.4* do TypeScript foi adicionada uma funcionalidade chamada `Incremental flag` , que pode ser utilizada no seu compilador. Essa nova funcionalidade salva as últimas alterações do compilador, fazendo com que ele seja mais rápido no momento

de desenvolvimento. Para utilizar essa funcionalidade, basta adicioná-la dentro de `compilerOptions` no seu arquivo `tsconfig.json`.

```
"compilerOptions": {
       //outras implementaçoes
       "incremental": true,
```

Mas antes de atualizar o seu arquivo `tsconfig.json`, vamos passar um outro parâmetro para o nosso compilador, o `-diagnostics`.

```
  "scripts": {
     //outras implementaçoes
    "compile": "tsc -w  -diagnostics",
```

Esse parâmetro deve retornar no seu terminal um diagnóstico completo, como quanto de memória foi utilizado para o transpile, CPU etc. Com o arquivo `package.json` atualizado, execute o comando `npm run compile` novamente no seu terminal.

```
//Resultado
Files:               99
Lines:            53010
Nodes:           225351
Identifiers:      80391
Symbols:          90546
Types:            33794
Instantiations:  104249
Memory used:    137637K
I/O read:         0.35s
I/O write:        0.06s
Parse time:       2.16s
Bind time:        0.85s
Check time:       3.98s
Emit time:        0.25s
Total time:       7.23s
```

Agora, para que possamos comparar como o compilador do TypeScript ficou antes e depois da funcionalidade `Incremental`

`flag` , adicione-a ao seu arquivo `tsconfig.json` , em seguida, altere qualquer arquivo no seu projeto e rode o comando `npm run compile` novamente:

```
//Resultado
Files:               99
Lines:            52955
Nodes:           226127
Identifiers:      80635
Symbols:          52682
Types:               78
Instantiations:       0
Memory used:    144461K
I/O read:         0.00s
I/O write:        0.00s
Parse time:       0.00s
Bind time:        0.00s
Check time:       0.00s
Emit time:        0.00s
Total time:       0.01s
```

Note a diferença: no primeiro exemplo, sem a `Incremental flag` , nós tivemos *7.23s* no tempo do transpile e, no segundo, tivemos *0.01s*. Bem mais rápido, não?

E agora, para que possamos validar o desenvolvimento dos passos anteriores, execute o comando `npm start` no seu terminal.

```
Resultado

node ./dist/program.js
servidor rodando na porta: 5000
```

Abra o seguinte endereço no seu navegador: http://localhost:5000

```
Resultado

{
```

```
"versao": "0.0.1"
}
```

Nessa chamada, nós estamos requisitando a nossa rota `default`, a rota de `health check`. Como nossa API está funcionando corretamente, o próximo passo será acessar as rotas da nossa `NewsController`.

Mas antes, nós precisamos inserir alguns registros no nosso banco de dados, para que possamos retornar às nossas pesquisas. Abra o seu `Robo 3T`, acesse o seu servidor `localhost` e vá até a sua base de dados `db_portal`. Clique em *Collections*, clique com o botão direito do seu mouse em *news* e em *Insert Document*. Em seguida, cole o seguinte trecho de código na modal e clique em *save*.

```
{
    "chapeu" : "vitória no MasterChef",
    "titulo" : "'O grito saiu da garganta', conta Danielle após r
ever vitória no MasterChef",
    "texto" : "Vencedora do sétimo episódio do MasterChef Brasil
2020, a economista Danielle comentou os melhores momentos de sua
passagem pela cozinha mais famosa do país. A campeã disse que a v
itória coroou sua dedicação à Gastronomia e que vencer o programa
 mudou a sua vida profissional. “O grito saiu da garganta”, conto
u após rever a decisão dos jurados.",
    "autor" : "Da Redação",
    "imagem" : "https://imagem.band.com.br/novahome/451a72ca-e766
-4422-81c2-fa2f0a09e2d2.jpg",
    "link" : "https://entretenimento.band.uol.com.br/masterchef/n
oticias/16308993/--o-grito-saiu-da-garganta---conta-danielle-apos
-rever-vitoria-no-masterchef",
    "dataPublicacao" : "2020-08-28T14:50:43.653",
    "tags": "mc2020",
    "ativo" : true
}

{

    "chapeu" : "Masterchef 2020",
```

```
    "titulo" : "10 fotos que provam que Erick Jacquin é muito est
iloso",
    "texto" : "No episódio dessa terça-feira, 25, o jurado Erick
Jacquin surpreendeu a todos ao participar do MasterChef Brasil co
m muita pompa e estilo. O chef de cozinha apostou em um terno azu
l ciano e pantufas! O detalhe não passou desapercebido por Ana Pa
ula Padrão, mas essa não é a primeira vez que o francês chama ate
nção por seus looks. Por isso, o Portal da Band separou 10 fotos
que provam que Jacquin é um dos caras mais estilosos que nós conh
ecemos.",
    "autor" : "Da Redação",
    "imagem" : "https://imagem.band.com.br/novahome/00ee18f0-270c
-4e3d-94d1-62250745f449.jpg",
    "link" : "https://entretenimento.band.uol.com.br/masterchef/n
oticias/16308890/10-fotos-que-provam-que-erick-jacquin-e-muito-es
tiloso",
    "dataPublicacao" : "2020-08-30T14:50:43.653",
    "tags": "mc2020",
    "ativo" : true
}
```

Agora que nós populamos a nossa base de dados, vamos fazer algumas requisições nas rotas da nossa classe `NewsController` . Vamos começar com a listagem completa. Para isso, abra o seguinte endereço no seu navegador: http://localhost:5000/api/v1/news/1/2

Resultado


```
{
"result": {
"Page": 1,
"Qtd": 10,
"Total": 2,
"Data": [
{
"_id": "5f84e9679fbb97678fa00c8f",
"chapeu": "vitória no MasterChef",
"titulo": "'O grito saiu da garganta', conta Danielle após rever
vitória no MasterChef",
"texto": "Vencedora do sétimo episódio do MasterChef Brasil 2020,
 a economista Danielle comentou os melhores momentos de sua passa
```

```
gem pela cozinha mais famosa do país. A campeã disse que a vitóri
a coroou sua dedicação à Gastronomia e que vencer o programa mudo
u a sua vida profissional. “O grito saiu da garganta”, contou após
s rever a decisão dos jurados.",
"autor": "Da Redação",
"imagem": "https://imagem.band.com.br/novahome/451a72ca-e766-4422
-81c2-fa2f0a09e2d2.jpg",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16308993/--o-grito-saiu-da-garganta---conta-danielle-apos-reve
r-vitoria-no-masterchef",
"dataPublicacao": "2020-08-28T17:50:43.653Z",
 "tags": "mc2020",
"ativo": true
},
{
"_id": "5f84e9679fbb97678fa00c92",
"chapeu": "Masterchef 2020",
"titulo": "10 fotos que provam que Erick Jacquin é muito estiloso
,
"texto": "No episódio dessa terça-feira, 25, o jurado Erick Jacqu
in surpreendeu a todos ao participar do MasterChef Brasil com mui
ta pompa e estilo. O chef de cozinha apostou em um terno azul cia
no e pantufas! O detalhe não passou desapercebido por Ana Paula P
adrão, mas essa não é a primeira vez que o francês chama atenção
por seus looks. Por isso, o Portal da Band separou 10 fotos que p
rovam que Jacquin é um dos caras mais estilosos que nós conhecemo
s.",
"autor": "Da Redação",
"imagem": "https://imagem.band.com.br/novahome/00ee18f0-270c-4e3d
-94d1-62250745f449.jpg",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16308890/10-fotos-que-provam-que-erick-jacquin-e-muito-estilos
o",
"dataPublicacao": "2020-08-30T17:50:43.653Z",
 "tags": "mc2020",
"ativo": true
}
]
}
}
```


Note que nesse resultado nós temos:

- `Page` : a página que passamos como parâmetro:

`/api/news/1` .

- `Qtd` : quantidade que estamos solicitando via parâmetro: `/api/news/1/10` .
- `Total` : total de registros que nós temos no nosso banco de dados.
- `Data` : resultado da nossa busca. Note que retornaram os dois registros adicionados no passo anterior através do `Robo 3T` .

Agora abra o endereço: http://localhost:3050/api/v1/news/ seguido de um dos `_id` .

`Resultado`


```
{
"result": {
"_id": "5f84e9679fbb97678fa00c8f",
"chapeu": "vitória no MasterChef",
"titulo": "'O grito saiu da garganta', conta Danielle após rever
vitória no MasterChef",
"texto": "Vencedora do sétimo episódio do MasterChef Brasil 2020,
 a economista Danielle comentou os melhores momentos de sua passa
gem pela cozinha mais famosa do país. A campeã disse que a vitóri
a coroou sua dedicação à Gastronomia e que vencer o programa mudo
u a sua vida profissional. “O grito saiu da garganta”, contou após
s rever a decisão dos jurados.",
"autor": "Da Redação",
"imagem": "https://imagem.band.com.br/novahome/451a72ca-e766-4422
-81c2-fa2f0a09e2d2.jpg",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16308993/--o-grito-saiu-da-garganta---conta-danielle-apos-reve
r-vitoria-no-masterchef",
"dataPublicacao": "2020-08-28T17:50:43.653Z",
 "tags": "mc2020",
"ativo": true
}
}
```


Caso queria buscar uma outra notícia, basta trocar o `_id` na

sua requisição.

Antes de finalizar este capítulo, vamos adicionar um novo pacote ao nosso projeto chamado `nodemon` . Para quem não conhece esse pacote, o `nodemon` reinicia o servidor automaticamente sempre que você salva um arquivo que o servidor está utilizando. O que isso significa?

Significa que nós não precisamos dar stop/start na nossa aplicação toda vez que fizermos uma alteração no nosso projeto, pois ele verifica que ocorreu uma alteração e já faz o *refresh* automaticamente. Para importá-lo digite o seguinte comando no seu terminal:

```
npm install  nodemon --save-dev
```

Com o `nodemon` instalado, atualize o seguinte trecho de código no seu arquivo `package.json` :

```
"scripts": {
  //outras implementações
   "start": "nodemon ./dist/program.js"
```

Com esse ajuste, toda vez que você alterar algum arquivo do seu projeto, ele vai reconhecer e dar *refresh* na sua aplicação.

Antes de finalizar este capítulo, que tal um resumo de tudo o que aprendemos?

Iniciando pela parte de modelagem de software, de maneira bem simples, nós modelamos uma API que retorna as notícias de um programa. Voltamos aos capítulos 4 e 5 deste livro, para reforçar os conceitos de POO (Programação Orientada a Objetos) e interfaces.

Aprendemos o que são `promises`, vimos como trabalhar com `try/catch` para tratamento de exceções e, além de desenvolver a nossa API, nós aprendemos algumas boas práticas no momento de desenhar a arquitetura do nosso projeto.

Com isso, nós finalizamos mais este capítulo. No próximo, daremos continuidade ao desenvolvimento da nossa API criando novas rotas.

Capítulo 12

CRIANDO NOVAS MODELS

Com a estrutura básica do nosso projeto OK, vamos avançar nele desenvolvendo duas novas rotas. Para isso, imagine o seguinte cenário: chegou uma nova demanda para a qual precisamos desenvolver duas novas rotas, uma para listagem de vídeos e uma outra para listagem de galeria de fotos.

Voltando ao capítulo anterior, o primeiro passo para a criação das nossas rotas será a definição das nossas models. Para isso, nós precisamos responder às seguintes perguntas:

- *O que todo vídeo tem?*
- *O que toda galeria de fotos tem?*

Em uma breve análise, nós temos:

Model de vídeos: título, texto, imagem, *duração*, link, *url*, data de publicação, tags e um atributo para dizer se o vídeo está ativo ou não.

Model de galeria de fotos: título, texto, data de publicação, tags, uma imagem de destaque da galeria, link da galeria de fotos, *uma listagem de fotos* e um atributo para dizer se ela está ativa ou não.

Caso você não tenha percebido, as models acima têm alguns atributos em comum com a nossa `NewsSchema`:

```
export const NewsSchema = new mongoose.Schema({
  titulo: { type: String },
  chapeu: { type: String },
  texto: { type: String },
  autor: { type: String },
  imagem: { type: String },
  dataPublicacao: { type: Date },
  tags: { type: String },
  link: { type: String },
  ativo: { type: Boolean }
});
```

Em uma breve análise, podemos observar que, entre as models de notícias e o levantamento que fizemos para as models de vídeos e galeria de fotos, temos as seguintes propriedades em comum: *título*, *texto*, *imagem*, *data de publicação*, *tags*, *link* e *ativo*.

Tendo isso em mente e pensando no paradigma de Orientação a Objetos, qual seria a melhor forma de criarmos essas novas models aproveitando as propriedades que elas têm em comum?

Caso a sua resposta seja criando uma classe **abstrata** e utilizando o conceito de **herança**, você está certa(o). Mas, para que possamos criar uma classe modelo para que todas as outras herdem dela, nós precisaremos fazer algumas modificações no nosso código.

Caso não se recorde do que é uma classe abstrata, eu recomendo que você volte ao capítulo 4 deste livro, sobre Orientação a Objetos.

12.1 POO (PROGRAMAÇÃO ORIENTADA A OBJETOS) NA PRÁTICA

Nosso primeiro passo será a criação da classe modelo, a nossa classe **abstrata**. Crie um novo arquivo chamado `core.ts` dentro do diretório `models` e atualize-o com o seguinte trecho de código:

```
export abstract class Core {
    titulo: String;
    texto: String;
    imagem: String;
    dataPublicacao: Date;
    tags: String;
    link: String;
    ativo: Boolean;
}
```

Note que a classe `Core` tem todas as propriedades em comum entre as nossas models.

O nosso próximo passo será a atualização do arquivo `newsSchema.ts`. Precisamos mover a configuração com o pacote `mongoose` para o arquivo `newsRepository.ts` e criar uma nova classe para a model de notícias, para que ela possa herdar as propriedades de `Core.ts`.

Renomeie o arquivo `newsSchema.ts` para `news.ts`, em seguida atualize-o com o seguinte trecho de código:

```
import { Core } from "./core";

export class News extends Core {
    chapeu: String;
    autor: String;
}
```

Agora atualize o arquivo `newsRepository.ts` com o seguinte trecho de código:

```
import mongoose from "mongoose";
import { News } from "../models/news";

const NewsSchema = new mongoose.Schema<News>({
  titulo: { type: String },
  chapeu: { type: String },
  texto: { type: String },
  autor: { type: String },
  imagem: { type: String },
  dataPublicacao: { type: Date },
  tags: { type: String },
  link: { type: String },
  ativo: { type: Boolean }
});

export const NewsRepository = mongoose.model<News>("news", NewsSc
hema);
```

Até aqui, nós criamos uma nova classe chamada `News`, que está herdando todas as propriedades da nossa classe `Core` e, em seguida, passamos o valor de `NewsSchema` para o arquivo `newsRepository.ts`.

Caso você esteja com o seu compilador executando, no console do Visual Studio Code está aparecendo o seguinte erro:

```
/* Erro na console*/

 Type 'News' does not satisfy the constraint 'Document<any>'.
  Type 'News' is missing the following properties from type 'Docu
ment<any>': $ignore, $isDefault, $isDeleted, $isEmpty, and 45 mor
e.
```

Esse erro diz que a model que estamos passando para o nosso repositório não está estendendo de `Document` do pacote `mongoose`. Para resolver isso, basta estender `Document` na classe `Core`:

```
import { Document } from 'mongoose';

export abstract class Core extends Document {
```

```
   /* atributos da classe news*/
}
```

Com a criação da classe abstrata `Core` e a atualização da classe de notícias, vamos agora criar as nossas models de vídeos e galeria de fotos.

Crie um arquivo chamado `videos.ts`, um chamado `galeria.ts` e um outro chamado `fotos.ts` dentro do diretório `models`. Em seguida, atualize-os com os seguintes trechos de código:

```
/* videos.ts*/
import { Core } from "./core";

export class Videos extends Core {
    url: String;
    duracao: String;
}
```

```
/*galeria.ts*/
import { Core } from "./core";
import { Fotos } from "./fotos";

export class Galeria extends Core {
    fotos: Array<Fotos>;
}
```

```
/*fotos.ts*/

export class Fotos  {
    thumb: String;
    thumbNail: String;
    credito: String;
    legenda: String;
}
```

Com esses arquivos criados, nós finalizamos a parte de criação das nossas models. O próximo passo será a criação dos nossos repositórios.

Criando os novos repositórios

Para a parte de criação dos repositórios, nós somente precisamos seguir o modelo atualizado em `newsRepository.ts`.

Crie dois novos arquivos dentro do diretório `repository`, um chamado `videosRepository.ts` e um outro chamado `galeriaRepository.ts`, em seguida atualize-os com os trechos de código:

```
/*videosRepository.ts*/
import mongoose from "mongoose";
import { Videos } from "../models/videos";

const VideosSchema = new mongoose.Schema<Videos>({
    titulo: { String },
    texto: { String },
    imagem: { String },
    duracao: { String },
    link: { String },
    url: { String },
    dataPublicacao: { Date },
    tags: { String },
    ativo: { Boolean }
});

export const VideosRepository = mongoose.model<Videos>("videos",
VideosSchema);

/*galeriaRepository.ts*/
import mongoose from "mongoose";
import { Fotos } from "../models/fotos";
import { Galeria } from "../models/galeria";

const GaleriaSchema = new mongoose.Schema<Galeria>({
    titulo: { String },
    texto: { String },
    dataPublicacao: { Date },
    fotos: [Array<Fotos>()],
    ativo: { Boolean }
});
```

```
export const GaleriaRepository = mongoose.model<Galeria>("galeria
, GaleriaSchema);
```

Com os repositórios criados, criaremos agora os nossos contratos, services e as controllers das nossas novas models.

12.2 GENERICS E TIPAGEM DE RETORNO DE FUNÇÕES NA PRÁTICA

Iniciando pelos contratos, nós precisaremos criar duas novas interfaces, uma para vídeos e outra para galeria de fotos. E cada uma delas deve conter dois métodos, um para buscar um determinado resultado pelo seu `id` e outro para retornar todos os dados de uma collection da nossa base de dados, passando dois parâmetros para paginação.

Analisando a interface `INewsService`, que nós criamos no capítulo anterior, note que nós já temos um contrato que retorna exatamente o que nós precisamos:

```
/*iNewsService.ts.ts*/
export interface INewsService {

    get(id: string);

    getAll(page: number, qtd: number): Promise<Result>;
}
```

Nele nós temos:

- Um método que retorna os dados de uma *entidade* pelo seu `id`;
- E um outro método que retorna uma `promise` da classe `Result`.

```
export class Result {
```

```
    Qtd: number;
    Page: number;
    Total: number;
    Data: any
}
```

Uma das formas de resolver a nossa demanda seria duplicar a nossa interface `INewsService`, renomeando-a para `IVideosService` e `IGaleriaService`, mas como nós já aprendemos neste livro, a duplicação de código pode gerar problemas futuros.

Então como podemos manter o nosso código organizado e criar os novos contratos sem duplicar a interface `INewsService`?

Se você respondeu "utilizando **herança**", a sua resposta está 50% correta. Para ela ficar 100% correta, vamos voltar ao capítulo 2, sobre tipagem. O nosso código não ficaria mais organizado se soubéssemos o retorno do método `get` e da propriedade `Data` da classe `Result`?

Acredito que a sua resposta tenha sido "sim", pois, dessa forma, além de o código ficar organizado, nós conseguimos utilizar um dos maiores recursos para se trabalhar com TypeScript, a tipagem.

Crie um novo arquivo dentro do diretório `contracts` chamado `iService.ts`, com o seguinte trecho de código:

```
/*iService.ts*/
import { Result } from "../infra/result";

export interface IService<T> {

    get(id: string): Promise<T>;

    getAll(page: number, qtd: number): Promise<Result<T>>;
}
```

Analisando o contrato `IService<T>`, nós criamos uma interface que recebe um parâmetro genérico que está sendo passado para os métodos `getAll` e `get`, que copiamos da interface `INewsService`.

Você se lembra dos Generics, que nós aprendemos no capítulo 6 deste livro? Essa é uma das formas como podemos trabalhar com eles no nosso dia a dia.

Caso você esteja com o seu compilador executando, no console do Visual Studio Code está aparecendo o seguinte erro:

```
/* Erro na console*/
 error TS2315: Type 'Result' is not generic.
getAll(page: number, qtd: number): Promise<Result<T>>;
```

Esse erro está informando que a classe `Result` não é genérica. Para resolvê-lo atualize o seu arquivo `result.ts`, com o seguinte trecho de código:

```
/*result.ts*/
export class Result<T> {
    Qtd: number;
    Page: number;
    Total: number;
    Data: Array<T>
}
```

Com a classe `Result<T>` OK, vamos atualizar a interface `INewsService` para que ela passe a herdar de `IService<T>` e criar os dois novos contratos `iVideosService.ts` e `iGaleriaService.ts` seguindo o exemplo de `INewsService`.

```
/*iNewsService.ts*/
import { News } from "../models/news";
import { IService } from "./iService";

export interface INewsService extends IService<News> { }
```

```
/*iVideosService.ts*/
import { Videos } from "../models/videos";
import { IService } from "./iService";

export interface IVideosService extends IService<Videos> { }
```

```
/*iGaleriaService.ts*/
import { Galeria } from "../models/galeria";
import { IService } from "./iService";

export interface IGaleriaService extends IService<Galeria> { }
```

Com a organização e a criação dos nossos contratos, o próximo passo será a atualização do arquivo `newsService.ts` e a criação das classes que devem implementar os novos contratos de vídeo e galeria de fotos.

Atualizando os serviços

Iniciando pela classe `NewsService`, atualize-a com o seguinte trecho de código:

```
/*newsService.ts*/
//outras implementações
import { News } from "../models/news";

export class NewsService implements INewsService {

    async get(_id: string): Promise<News> {
        let result = await NewsRepository.findById(_id);
        return result;
    }

    async getAll(page: number, qtd: number): Promise<Result<News>
> {
        let result = new Result<News>();
        result.Page = page;
        result.Qtd = qtd;
        result.Total = await NewsRepository.count({});
        result.Data = await NewsRepository.find({}).skip((page *
qtd) - qtd).limit(qtd);
```

```
        return result;
    }

}
```

Analisando as alterações realizadas no arquivo `newsService.ts`, nós temos:

- Alteramos o método `get` para que ele retorne uma `Promise` da model `News`.
- Em `getAll`, passamos a model `News` para a classe `Result`, que deve retornar uma `Promise` de `News` na propriedade `Data`.

Com a classe `newsService.ts` atualizada, crie dois novos arquivos dentro do diretório `services`, um chamado `videosService.ts` e um outro chamado `galeriaService.ts` e, em seguida, atualize-os com os seguintes trechos de código:

```
/*videosService.ts*/
import { IVideosService } from "../contracts/iVideosService";
import { Result } from "../infra/result";
import { Videos } from "../models/videos";
import { VideosRepository } from "../repository/videosRepository"
;

export class VideosService implements IVideosService {

    async get(_id: string): Promise<Videos> {
        let result = await VideosRepository.findById(_id);
        return result;
    }

    async getAll(page: number, qtd: number): Promise<Result<Video
s>> {
        let result = new Result<Videos>();
        result.Page = page;
        result.Qtd = qtd;
        result.Total = await VideosRepository.count({});
        result.Data = await VideosRepository.find({}).skip((page
```

```
* qtd) - qtd).limit(qtd);
        return result;
    }

}
```

```
/*galeriaService.ts*/
import { IGaleriaService } from "../contracts/iGaleriaService";
import { Result } from "../infra/result";
import { Galeria } from "../models/galeria";
import { GaleriaRepository } from "../repository/galeriaRepositor
y";

export class GaleriaService implements IGaleriaService {

    async get(_id: string): Promise<Galeria> {
        let result = await GaleriaRepository.findById(_id);
        return result;
    }

    async getAll(page: number, qtd: number): Promise<Result<Galer
ia>> {
        let result = new Result<Galeria>();
        result.Page = page;
        result.Qtd = qtd;
        result.Total = await GaleriaRepository.count({});
        result.Data = await GaleriaRepository.find({}).skip((page
 * qtd) - qtd).limit(qtd);
        return result;
    }

}
```

Com os serviços OK, o nosso próximo passo será a criação das nossas controllers e a atualização do arquivo `startUp.ts` com as novas rotas.

Criação dos arquivos de entrada

Crie dois arquivos dentro do diretório `controller`, um chamado `videosController.ts` e um outro chamado

`galeriaController.ts` e, em seguida, atualize-os com os trechos de código a seguir:

```
/*videosController.ts*/
import { Request, Response } from "express";
import { VideosService } from "../services/videosService";


class VideosController {

    private _service: VideosService;

    constructor() {
        this._service = new VideosService();
    }

    async get(request: Request, response: Response) {
        try {
            const page = request.params.page ? parseInt(request.p
arams.page) : 1;
            const qtd = request.params.qtd ? parseInt(request.par
ams.qtd) : 10;
            let result = await this._service.getAll(page, qtd);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }

    async getById(request: Request, response: Response) {
        try {
            const _id = request.params.id;
            let result = await this._service.get(_id);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }
}
```

```
export default new VideosController();

/*galeriaController.ts*/
import { Request, Response } from "express";
import { GaleriaService } from "../services/galeriaService";


class GaleriaController {

    private _service: GaleriaService;

    constructor() {
        this._service = new GaleriaService();
    }

    async get(request: Request, response: Response) {
        try {
            const page = request.params.page ? parseInt(request.p
arams.page) : 1;
            const qtd = request.params.qtd ? parseInt(request.par
ams.qtd) : 10;
            let result = await this._service.getAll(page, qtd);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }

    async getById(request: Request, response: Response) {
        try {
            const _id = request.params.id;
            let result = await this._service.get(_id);
            response.status(200).json({ result });

        } catch (error) {
            response.status(500).json({ error: error.message || e
rror.toString() });
        }
    }
}

export default new GaleriaController();
```

Não temos nenhuma novidade na criação desses arquivos, caso você compare com `NewsController` eles têm a mesma estrutura.

Para que possamos testar todo o fluxo desenvolvido neste capítulo, nós precisaremos criar as rotas de entrada das nossas novas controllers. Para isso, abra o seu arquivo `startUp.ts` e atualize o método `routes` dele com o seguinte trecho de código:

```
/*startUp.ts*/

//Importação das novas controllers
import VideosController from "./controller/videosController";
import GaleriaController from "./controller/galeriaController";

//outras implementações

//método routes
routes() {
        this.app.route("/").get((req, res) => {
            res.send({ versao: "0.0.2" });
        });

        /*news*/
        this.app.route("/api/v1/news/:page/:qtd").get((req: Reque
st, res: Response) => {
            return NewsController.get(req, res);
        });


        this.app.route("/api/v1/news/:id").get((req: Request, res
: Response) => {
            return NewsController.getById(req, res);
        });

        /*videos*/
        this.app.route("/api/v1/videos/:page/:qtd").get((req: Req
uest, res: Response) => {
            return VideosController.get(req, res);
        });

        this.app.route("/api/v1/videos/:id").get((req: Request, r
es: Response) => {
```

```
            return VideosController.getById(req, res);
        });

         /*galeria*/
        this.app.route("/api/v1/galeria/:page/:qtd").get((req: Re
quest, res: Response) => {
            return GaleriaController.get(req, res);
        });

        this.app.route("/api/v1/galeria/:id").get((req: Request,
res: Response) => {
            return GaleriaController.getById(req, res);
        });
    }
```

Criando as novas collections

Com o código pronto, crie duas novas collections dentro da sua base `db_portal`, uma chamada `videos` e outra chamada `galerias`, com os dados a seguir.

Caso não se recorde de como criar uma collection ou de como atualizá-la, recomendo que leia novamente os capítulos 10 e 11 deste livro.

```
/*videos*/
{
    "titulo" : "Anna Paula vence final do MasterChef 2020 e dedic
a prêmio à filha",
    "texto" : "Cozinheira preparou um prato bem brasileiro e se e
mocionou ao levar grande prêmio da temporada",
    "imagem" : "https://thumb.mais.uol.com.br/16886600-xlarge.jpg
?ver=1",
    "duracao" : "00:04:36",
    "link" : "https://entretenimento.band.uol.com.br/videos/16886
600/anna-paula-vence-final-do-masterchef-2020-e-dedica-premio-a-f
ilha",
    "url" : "https://player.mais.uol.com.br/?mediaId=16886600",
    "dataPublicacao" : "2020-12-30T03:12:00.000",
    "tags" : "masterchef, masterchef 2020",
    "ativo" : true
```

```
}
{
    "titulo" : "Muito boa essa mousse de caramelo, diz Jacquin pa
ra Marina",
    "texto" : "Estudante fez uma tartelette de maçã com mousse da
 caramelo salgado com alguns erros técnicos, mas causou uma boa i
mpressão",
    "imagem" : "https://thumb.mais.uol.com.br/16886599-xlarge.jpg
?ver=1",
    "duracao" : "00:01:27",
    "link" : "https://entretenimento.band.uol.com.br/videos/16886
599/muito-boa-essa-mousse-de-caramelo-diz-jacquin-para-marina",
    "url" : "https://player.mais.uol.com.br/?mediaId=16886599",
    "dataPublicacao" : "2020-12-30T03:12:00.000",
    "tags" : "masterchef, masterchef 2020",
    "ativo" : true
}

/*galeria*/

{
"titulo": "Marília Mendonça, Péricles, César Menotti e Maraísa em
 especial de Natal",
"texto": "Especial de natal com celebridades no MasterChef 2020",
"imagem": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc1
5aa.jpg",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16318389/marilia-mendonca-pericles-maraisa-e-cesar-menotti-par
ticipam-de-%E2%80%9Cespecial-de-natal-do-masterchef",
"dataPublicacao": "2020-08-12T10:26:00.00",
"tags": "masterchef, masterchef 2020",
"ativo": true,
"fotos": [
[
{
"thumb": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc15
aa.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/eb9e64b485e94e
fc15aa.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "No especial de Natal, mais famosos entraram na cozinh
a do MasterChef"
}
],
[
```

```
{
"thumb": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f66c93
5e.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f6
6c935e.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Foi a vez de César Menotti e Maraísa mostrarem seus d
otes culinários"
}
],
[
{
"thumb": "https://pubimg.band.uol.com.br/files/fad9cc5b705a031950
87.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/fad9cc5b705a03
195087.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Eles tiveram de enfrentar Péricles e Marília Mendonça
 no desafio"
}
]
]
}
```

12.3 TESTANDO AS NOVAS ROTAS

Para que possamos validar se tudo está funcionando corretamente, execute o comando `npm run compile` para fazer o transpile do seu projeto e depois execute o comando `npm start` .

Começando a validação pela rota de vídeos, vamos seguir o mesmo fluxo que fizemos para a validação da rota de notícias no capítulo anterior.

Abra o seguinte endereço no seu navegador: http://localhost:5000/api/v1/videos/1/2

```
Resultado
{
result: {
```

```
Page: 1,
Qtd: 2,
Total: 2,
Data: [
{
titulo: "Anna Paula vence final do MasterChef 2020 e dedica prêmi
o à filha",
texto: "Cozinheira preparou um prato bem brasileiro e se emociono
u ao levar grande prêmio da temporada",
imagem: "https://thumb.mais.uol.com.br/16886600-xlarge.jpg?ver=1"
,
duracao: "00:04:36",
link: "https://entretenimento.band.uol.com.br/videos/16886600/ann
a-paula-vence-final-do-masterchef-2020-e-dedica-premio-a-filha",
url: "https://player.mais.uol.com.br/?mediaId=16886600",
dataPublicacao: "2020-12-30T03:12:00.000",
tags: "masterchef, masterchef 2020",
ativo: true,
_id: "6062740c066a94dda6662c63"
},
{
titulo: "Muito boa essa mousse de caramelo, diz Jacquin para Mari
na",
texto: "Estudante fez uma tartelette de maçã com mousse da carame
lo salgado com alguns erros técnicos, mas causou uma boa impressã
o",
imagem: "https://thumb.mais.uol.com.br/16886599-xlarge.jpg?ver=1"
,
duracao: "00:01:27",
link: "https://entretenimento.band.uol.com.br/videos/16886599/mui
to-boa-essa-mousse-de-caramelo-diz-jacquin-para-marina",
url: "https://player.mais.uol.com.br/?mediaId=16886599",
dataPublicacao: "2020-12-30T03:12:00.000",
tags: "masterchef, masterchef 2020",
ativo: true,
_id: "6062740c066a94dda6662c64"
}
]
}
}
```


Agora testando a listagem de galerias, abra o seguinte endereço no seu navegador: http://localhost:5000/api/v1/galeria/1/10

Resultado


```
{
"result": {
"Page": 1,
"Qtd": 10,
"Total": 1,
"Data": [
{
"titulo": "Marília Mendonça, Péricles, César Menotti e Maraísa em
 especial de Natal",
"chapeu": "MasterChef com celebridades",
"texto": "Especial de natal com celebridades no MasterChef 2020",
"autor": "Da Redação",
"imagem": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc1
5aa.jpg",
"dataPublicacao": "2020-08-12T10:26:00.00",
"ativo": true,
"fotos": [
[
{
"thumb": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc15
aa.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/eb9e64b485e94e
fc15aa.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "No especial de Natal, mais famosos entraram na cozinh
a do MasterChef"
}
],
[
{
"thumb": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f66c93
5e.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f6
6c935e.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Foi a vez de César Menotti e Maraísa mostrarem seus d
otes culinários"
}
],
[
{
"thumb": "https://pubimg.band.uol.com.br/files/fad9cc5b705a031950
87.jpg",
```

```
"thumbNail": "https://pubimg.band.uol.com.br/files/fad9cc5b705a03
195087.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Eles tiveram de enfrentar Péricles e Marília Mendonça
 no desafio"
}
]
],
"_id": "5fe7c2bb55d856158389a162",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16318389/marilia-mendonca-pericles-maraisa-e-cesar-menotti-par
ticipam-de-%E2%80%9Cespecial-de-natal-do-masterchef"
}
]
}
}
```


Analisando os retornos, note que, através de uma organização utilizando conceitos simples como o da classe genérica `Result<T>` , nós conseguimos garantir uma estrutura de retorno padrão para todas as nossas rotas.

Neste capítulo, nós reforçamos alguns conceitos que nós aprendemos no livro, como a importância de se tipar os nossos dados, como e onde utilizar *Generics* e vimos mais alguns exemplos de como a Orientação a Objetos deixa o nosso código mais organizado.

Com isso, nós finalizamos mais este capítulo. No próximo, veremos sobre *Injeção de dependência*, o que é e como aplicá-la no nosso código.

Capítulo 13

INJEÇÃO DE DEPENDÊNCIA

Neste capítulo, nós faremos uma breve introdução à Injeção de Dependência e logo partiremos para a parte prática aplicando esse padrão no nosso projeto desenvolvido nos capítulos anteriores.

Aqui não será necessário nos aprofundar no que é Injeção de Dependência e nem nas formas que temos para trabalhar com ela. Vamos passar rapidamente por alguns conceitos básicos para que você possa entender os benefícios de utilizá-la e logo partiremos para o *hands-on*.

Bom, mas o que seria Injeção de Dependência?

Injeção de Dependência é um padrão de projeto utilizado para evitar o alto nível de acoplamento de código dentro de uma aplicação.

Para quem está tendo o seu primeiro contato com *padrão de projetos* (Design Patterns) neste livro, um padrão de projeto ou padrão de desenho é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no desenvolvimento de software. Caso você tenha interesse em saber mais sobre esse assunto, eu recomendo a leitura do seguinte link:

https://pt.wikipedia.org/wiki/Padr%C3%A3o_de_projeto_de_software.

Para ficar mais claro o que seria acoplamento, vamos analisar a nossa classe `NewsController`.

```
/*newsController.ts*/
class NewsController {

    private _service: NewsService;

    constructor() {
        this._service = new NewsService();
    }
    //outras implementações
```

Acredito que nesse momento você deve estar se perguntando: o nosso código não está funcionando? Analisando-o, vemos que estamos instanciando uma classe, conforme aprendemos no capítulo 4 deste livro, e utilizando os seus métodos. Onde está o erro?

Em uma breve análise, essa classe está OK. O código compila e ela está funcionando corretamente, mas note que ela tem uma instância de `NewsService`. Conseguiu achar o problema?

Caso não tenha encontrado, imagine o seguinte cenário: chegou uma nova demanda para adicionar um parâmetro ao construtor da classe `NewsService`. Perceba que isso afetará diretamente a classe `NewsController`, que a está instanciando sem passar nenhum parâmetro.

```
/*NewsService.ts*/
export class NewsService implements INewsService {

    constructor(acomplamento: string) {
        console.log(acomplamento);
```

```
    }
//outras implementações

/*Resultado*/
controller/newsController.ts:10:25 - error TS2554: Expected 1 arg
uments, but got 0.
10         this._service = new NewsService();

//outras implementações
```

Sistemas com alto acoplamento de código têm os seguintes problemas:

- Dificuldade no momento de manutenção;
- Dificuldade no momento de escrever os testes;
- Insegurança no momento de desenvolver novas funcionalidades.

Esses são alguns dos problemas que nós conseguimos resolver adotando a utilização de Injeção de Dependência no nosso projeto.

Para que você possa ter um melhor entendimento sobre esse assunto, vamos alterar o nosso projeto para ver na prática quais são os benefícios de se trabalhar com Injeção de Dependência do seu dia a dia.

13.1 DESACOPLANDO O PROJETO

Avançando para a parte prática, vamos alterar o nosso código para remover os acoplamentos entre as nossas classes.

O primeiro passo será baixar uma biblioteca chamada *tsyringe* e uma outra chamada *reflect-metadata*. Para isso, abra um terminal no seu computador, navegue até o seu projeto e digite o seguinte comando nele: `npm i tsyringe reflect-metadata --save .`

- `tsyringe` : esse pacote nos permite trabalhar com injeção de dependência.
- `reflect-metadata` : esse pacote nos permite trabalhar com os types em tempo de execução.

Com os pacotes importados, o próximo passo será configurá-los. Crie um novo diretório chamado `shared` e, dentro dele, um arquivo chamado `container.ts` .

O arquivo `container.ts` será responsável por registrar as interfaces do nosso projeto: `INewsService` , `IVideosService` , `IGaleriaService` e as classes `NewsService` , `VideosService` e `GaleriaService` , que estão implementando essas interfaces.

Atualize o seu arquivo `container.ts` com o seguinte trecho de código:

```
/*container.ts*/
import "reflect-metadata";
import { container } from 'tsyringe';
import { GaleriaService } from "../services/galeriaService";
import { NewsService } from "../services/newsService";
import { VideosService } from "../services/videosService";


container.register(
    "INewsService", {
    useClass: NewsService
},
);

container.register(
    "IVideosService", {
    useClass: VideosService
},
);

container.register(
    "IGaleriaService", {
```

```
    useClass: GaleriaService
},
);
```

O próximo passo será importar o contêiner, que criamos acima, no nosso arquivo `startUp.ts` , para que ele seja carregado no nosso projeto, e alterar a forma como estamos chamando nossos serviços dentro do método `routes()` .

```
/*startUp.ts*/
import "reflect-metadata";
import { NewsController } from "./controller/newsController";
import { VideosController } from "./controller/videosController";
import { GaleriaController } from "./controller/galeriaController
;
import { container } from 'tsyringe';
import './shared/container';

class StartUp {

    //outras implementações

    private news    = container.resolve(NewsController);
    private videos  = container.resolve(VideosController);
    private galeria = container.resolve(GaleriaController);

    routes() {
        //outras implementações

        /*news*/
        this.app.route("/api/v1/news/:page/:qtd").get((req: Reque
st, res: Response) => {
            return this.news.get(req, res);
        });


        this.app.route("/api/v1/news/:id").get((req: Request, res
: Response) => {
            return this.news.getById(req, res);
        });

        /*videos*/
        this.app.route("/api/v1/videos/:page/:qtd").get((req: Req
```

```
uest, res: Response) => {
            return this.videos.get(req, res);
        });

        this.app.route("/api/v1/videos/:id").get((req: Request, r
es: Response) => {
            return this.videos.getById(req, res);
        });

         /*galeria*/
        this.app.route("/api/v1/galeria/:page/:qtd").get((req: Re
quest, res: Response) => {
            return this.galeria.get(req, res);
        });

        this.app.route("/api/v1/galeria/:id").get((req: Request,
res: Response) => {
            return this.galeria.getById(req, res);
        });
    }
}
```

Analisando a classe `StartUp` nós temos:

- Na primeira linha do arquivo, estamos importando o pacote `reflect-metadata`. Esse import precisa estar no início do arquivo para que ele seja o primeiro a ser carregado;
- Depois estamos importando o contêiner em que nós registramos as nossas interfaces antes da criação da classe `StartUp`;
- Em seguida, criamos três propriedades: `news`, `videos` e `galeria`, e passamos para elas o valor da instância das controllers;
- Por fim, nós removemos a chamada direta que estava nos métodos para as três propriedades que nós criamos: `news`, `videos e galeria`.

Agora, caso você esteja executando o seu transpile, no console do Visual Studio Code, apareceram nove erros:

```
/* Erros de transpile*/
 startUp.ts:5:10 - error TS2614: Module '"./controller/newsContro
ller"' has no exported member 'NewsController'. Did you mean to u
se 'import NewsController from "./controller/newsController"' ins
tead?

5 import { NewsController } from "./controller/newsController";
           ~~~~~~~~~~~~~~

startUp.ts:6:10 - error TS2614: Module '"./controller/videosContr
oller"' has no exported member 'VideosController'. Did you mean t
o use 'import VideosController from "./controller/videosControlle
r"' instead?

6 import { VideosController } from "./controller/videosController
;
           ~~~~~~~~~~~~~~~~

startUp.ts:7:10 - error TS2614: Module '"./controller/galeriaCont
roller"' has no exported member 'GaleriaController'. Did you mean
 to use 'import GaleriaController from "./controller/galeriaContr
oller"' instead?

7 import { GaleriaController } from "./controller/galeriaControll
er";
           ~~~~~~~~~~~~~~~~~

startUp.ts:36:30 - error TS2339: Property 'get' does not exist on
 type 'unknown'.

36             return this.news.get(req, res);
                                ~~~

startUp.ts:41:30 - error TS2339: Property 'getById' does not exis
t on type 'unknown'.

41             return this.news.getById(req, res);
                                ~~~~~~~

startUp.ts:46:32 - error TS2339: Property 'get' does not exist on
 type 'unknown'.
```

```
46             return this.videos.get(req, res);
                                  ~~~

startUp.ts:50:32 - error TS2339: Property 'getById' does not exis
t on type 'unknown'.

50             return this.videos.getById(req, res);
                                  ~~~~~~~

startUp.ts:55:33 - error TS2339: Property 'get' does not exist on
 type 'unknown'.

55             return this.galeria.get(req, res);
                                   ~~~

startUp.ts:59:33 - error TS2339: Property 'getById' does not exis
t on type 'unknown'.

59             return this.galeria.getById(req, res);
```

Esses erros estão dizendo que nós não podemos mais exportar as nossas controllers instanciando-as e que devemos atualizar o nosso arquivo `tsconfig.json` para que possamos trabalhar com os decorators. Caso você não se lembre do que são os decorators, eu recomendo que volte ao capítulo 7 deste livro.

13.2 DECORATORS NA PRÁTICA

Para que possamos trabalhar com os decorators, nós precisamos adicionar as propriedades a seguir ao arquivo `tsconfig.json`:

```
  "compilerOptions": {
//outras implementações
"experimentalDecorators": true,
"emitDecoratorMetadata": true,
//outras implementações
}
```

Vamos atualizar os arquivos `newsController.ts`, `videosController.ts` e `galeriaController.ts` para resolver o problema com o `export`, que está aparecendo na console do Visual Studio Code.

```
/*newsController.ts*/
import { injectable, inject } from "tsyringe";
import { INewsService } from "../contracts/iNewsService";
//outras implementações


@injectable()
export class NewsController {

    constructor(@inject('INewsService') private _service: INewsSe
rvice) { }

    //outras implementações
   //Remover export default new NewsController();
}

/*videosController.ts*/
import { injectable, inject } from "tsyringe";
import { IVideosService } from "../contracts/iVideosService";
//outras implementações


@injectable()
export class VideosController {

    constructor(@inject('IVideosService') private _service: IVide
osService) {}
   //outras implementações
   //Remover export default new VideosController();
}

/*galeriaController.ts*/
import { injectable, inject } from "tsyringe";
import { IGaleriaService } from "../contracts/iGaleriaService";

@injectable()
export class GaleriaController {
```

```
    constructor(@inject('IGaleriaService') private _service: IGal
eriaService) {}
   //outras implementações
   //Remover export default new GaleriaController();
}
```

Vamos analisar os três arquivos que nós atualizamos:

- Removemos a última linha que os estava exportando como uma nova instância e passamos a exportá-los direto, deixando a parte da instância para a Injeção de Dependência;
- Em seguida, nós adicionamos dois decorators, um de classe, o `@injectable()` , e um outro de parâmetro no construtor, o `@inject('')` .

13.3 TESTANDO O PROJETO

Para que possamos testar se o nosso projeto continua funcionando corretamente, vamos validar todas as rotas desenvolvidas nos capítulos anteriores.

Vamos começar pelas rotas de notícias:

```
/*News
rota: http://localhost:5000/api/v1/news/1/1
Resultado:
*/

{
"result": {
"Page": 1,
"Qtd": 1,
"Total": 2,
"Data": [
{
"_id": "5fe93c5555d856158389a165",
"chapeu": "vitória no MasterChef",
```

```
"titulo": "'O grito saiu da garganta', conta Danielle após rever
vitória no MasterChef",
"texto": "Vencedora do sétimo episódio do MasterChef Brasil 2020,
 a economista Danielle comentou os melhores momentos de sua passa
gem pela cozinha mais famosa do país. A campeã disse que a vitóri
a coroou sua dedicação à Gastronomia e que vencer o programa mudo
u a sua vida profissional. “O grito saiu da garganta”, contou após
s rever a decisão dos jurados.",
"autor": "Da Redação",
"imagem": "https://imagem.band.com.br/novahome/451a72ca-e766-4422
-81c2-fa2f0a09e2d2.jpg",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16308993/--o-grito-saiu-da-garganta---conta-danielle-apos-reve
r-vitoria-no-masterchef",
"dataPublicacao": "2020-08-28T17:50:43.653Z",
"ativo": true
}
]
}
}

/*Videos
http://localhost:5000/api/v1/videos/1/1
Resultado:
*/

{
"result": {
"Page": 1,
"Qtd": 1,
"Total": 2,
"Data": [
{
"titulo": "Péricles e Marília Mendonça vencem MasterChef especial
 de Natal",
"chapeu": "Masterchef 2020",
"texto": "Amigos de longa data, os dois cantores mostraram que fo
rmam uma boa dupla também na cozinha. Eles prepararam lombo suíno
, risoto de uva-passa e manjar de coco.",
"autor": "Da Redação",
"imagem": "http://thumb.mais.uol.com.br/16885455-xlarge.jpg?ver=0
,
"dataPublicacao": "2020-12-24T03:12:00.000",
"duracao": "00:02:39",
"tags": "masterchef, masterchef 2020",
```

```
"ativo": true,
"_id": "5fe943e755d856158389a167",
"link": "https://player.mais.uol.com.br/?mediaId=16885455&amp;aut
oplay=false&amp;share=false&amp;startHd=720p&amp;related=false"
}
]
}
}
```


```
/*Galeria
http://localhost:5000/api/v1/galeria/1/1
Resultado:
*/
```


```
{
"result": {
"Page": 1,
"Qtd": 1,
"Total": 1,
"Data": [
{
"titulo": "Marília Mendonça, Péricles, César Menotti e Maraísa em
 especial de Natal",
"chapeu": "MasterChef com celebridades edição de Natal",
"texto": "Especial de natal com celebridades no MasterChef 2020",
"autor": "Da Redação",
"imagem": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc1
5aa.jpg",
"dataPublicacao": "2020-08-12T10:26:00.00",
"ativo": true,
"fotos": [
[
{
"thumb": "https://pubimg.band.uol.com.br/files/eb9e64b485e94efc15
aa.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/eb9e64b485e94e
fc15aa.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "No especial de Natal, mais famosos entraram na cozinh
a do MasterChef"
}
],
[
{
"thumb": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f66c93
```

```
5e.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/c41a8db03aa1f6
6c935e.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Foi a vez de César Menotti e Maraísa mostrarem seus d
otes culinários"
}
],
[
{
"thumb": "https://pubimg.band.uol.com.br/files/fad9cc5b705a031950
87.jpg",
"thumbNail": "https://pubimg.band.uol.com.br/files/fad9cc5b705a03
195087.jpg",
"credito": "Carlos Reinis/Band",
"legenda": "Eles tiveram de enfrentar Péricles e Marília Mendonça
 no desafio"
}
]
],
"_id": "5fe9442055d856158389a169",
"link": "https://entretenimento.band.uol.com.br/masterchef/notici
as/16318389/marilia-mendonca-pericles-maraisa-e-cesar-menotti-par
ticipam-de-%E2%80%9Cespecial-de-natal-do-masterchef"
}
]
}
}
```


Antes de finalizar este capítulo, eu gostaria de sugerir um desafio. :)

Imagine o seguinte cenário: chegou uma nova demanda para criarmos uma nova rota de listagem de podcasts. Tendo em mente o que desenvolvemos até aqui, quais seriam os passos para a criação dessa nova rota? (Caso fique em dúvida referente aos atributos de um podcast, ele tem os mesmos atributos da model de vídeos).

A seguir, você tem o link do projeto no meu GitHub com todo

o código que nós desenvolvemos até este capítulo do livro mais a implementação da nossa nova demanda de podcast: https://github.com/programadriano/api_mc_livro/tree/capitulo-13

Com isso, finalizamos mais este capítulo. No próximo, nós documentaremos o nosso projeto.

Capítulo 14

DOCUMENTANDO O PROJETO

Para finalizar o fluxo de desenvolvimento do nosso projeto, o próximo passo será documentá-lo. Para isso, nós utilizaremos o JSDoc.

Caso não se recorde do JSDoc, trata-se de uma linguagem de marcação que nós podemos utilizar para documentar código JavaScript. Ele nos permite usar comentários para fornecer informações sobre elementos de código, como funções, campos e variáveis.

14.1 ORGANIZANDO O PROJETO

Antes de documentar o nosso projeto, que tal fazer um ajuste nele? Analisando a nossa classe `StartUp`, observe que o método `routes()` está com quase 40 linhas de código e, caso chegue uma nova demanda para criarmos uma nova rota no nosso projeto, esse método só tende a aumentar.

Para resolver esse problema de quantidade de linhas, nós podemos migrar as chamadas das rotas de dentro desse método para uma outra camada do nosso projeto. Esse processo, além de

diminuir a quantidade de linhas de código do nosso método, deixará o nosso projeto mais organizado.

Ajustando a classe StartUp

Crie um novo diretório na raiz do seu projeto chamado `router` e dentro dele crie quatro arquivos: `newsRouter.ts`, `videosRouter.ts`, `galeriaRouter.ts` e a nova rota `podcastRouter.ts`, que criamos no capítulo anterior para podcasts.

Iniciando pelo arquivo `newsRouter.ts`, vamos copiar para ele as dependências da nossa rota `/news` que estão dentro do arquivo `startUp.ts`:

```
//newsRouter.ts
import "reflect-metadata";
import express, { Request, Response } from "express";
import { container } from "tsyringe";
import { NewsController } from "../controller/newsController";

const newsRouter = express();
const news = container.resolve(NewsController);

newsRouter.route("/api/v1/news/:page/:qtd").get((req: Request, re
s: Response) => {
    return news.get(req, res);
});


newsRouter.route("/api/v1/news/:id").get((req: Request, res: Resp
onse) => {
    return news.getById(req, res);
});

export default newsRouter;
```

Analisando o arquivo `newsRouter.ts`, temos:

- No início dele nós copiamos os pacotes da classe `StartUp`: `reflect-metadata`, `express`, `tsyringe` e a classe `NewsController`;
- Em seguida, nós criamos uma constante chamada `newsRouter`, recebendo o pacote `express()`;
- Depois, criamos uma constante chamada `news` e passamos a instância de `NewsController` para ela;
- Por fim, copiamos as chamadas das nossas rotas e exportamos a constante `newsRouter` para que possamos importá-la na classe `StartUp`.

Agora que já sabemos como criar o nosso arquivo de rotas, atualize os arquivos `videosRouter.ts`, `galeriaRouter.ts` e `podcastRouter.ts` com os seguintes trechos de código:

```
//videosRouter.ts
import "reflect-metadata";
import express, { Request, Response } from "express";
import { container } from "tsyringe";
import { VideosController } from "../controller/videosController"
;

const videosRouter = express();
const videos = container.resolve(VideosController);

videosRouter.route("/api/v1/videos/:page/:qtd").get((req: Request
, res: Response) => {
    return videos.get(req, res);
});

videosRouter.route("/api/v1/videos/:id").get((req: Request, res:
Response) => {
    return videos.getById(req, res);
});

export default videosRouter;

//galeriaRouter.ts
import "reflect-metadata";
```

```
import express, { Request, Response } from "express";
import { container } from "tsyringe";
import { GaleriaController } from "../controller/galeriaControlle
r";

const galeriaRouter = express();
const galeria = container.resolve(GaleriaController);

galeriaRouter.route("/api/v1/galeria/:page/:qtd").get((req: Reque
st, res: Response) => {
    return galeria.get(req, res);
});

galeriaRouter.route("/api/v1/galeria/:id").get((req: Request, res
: Response) => {
    return galeria.getById(req, res);
});

export default galeriaRouter;

//podcastRouter.ts
import "reflect-metadata";
import express, { Request, Response } from "express";
import { container } from "tsyringe";
import { PodcastController } from "../controller/podcastControlle
r";

const podcastRouter = express();
const podcast = container.resolve(PodcastController);

podcastRouter.route("/api/v1/podcast/:page/:qtd").get((req: Reque
st, res: Response) => {
    return podcast.get(req, res);
});

podcastRouter.route("/api/v1/podcast/:id").get((req: Request, res
: Response) => {
    return podcast.getById(req, res);
});

export default podcastRouter;
```

O próximo passo será remover as chamadas da classe `StartUp` que nós copiamos para os arquivos `newsRouter.ts` ,

`videosRouter.ts`, `galeriaRouter.ts` e `podcastRouter.ts` e adicionar essas novas chamadas dentro do método `routes()`.

A seguir você tem o arquivo `startUp.ts` atualizado:

```
//startUp.ts atualizada
import express, { Application } from "express";
import database from "./infra/db";

import './shared/container';
import newsRouter from "./router/newsRouter";
import videosRouter from "./router/videosRouter";
import galeriaRouter from "./router/galeriaRouter";
import podcastRouter from "./router/podcastRouter";

class StartUp {

    public app: Application;
    private _db: database = new database();

    constructor() {
        this.app = express();
        this._db.createConnection();
        this.routes();
    }

    routes() {
        this.app.route("/").get((req, res) => {
            res.send({ versao: "0.0.2" });
        });

        this.app.use("/", newsRouter);
        this.app.use("/", videosRouter);
        this.app.use("/", galeriaRouter);
        this.app.use("/", podcastRouter);
    }
}

export default new StartUp();
```

Bem mais organizado, certo?

A nossa classe `StartUp` tinha aproximadamente 73 linhas de

código e depois do nosso ajuste ela ficou com aproximadamente 33 linhas.

14.2 DOCUMENTANDO O NOSSO CÓDIGO

A prática de adicionar comentários ao código ajuda no entendimento de outros desenvolvedores no momento de uma possível manutenção ou implementação de novas funcionalidades.

Conforme mencionado no início deste capítulo, nós utilizaremos o JSDoc para documentar o código.

A seguir, você tem algumas das marcações que ele permite adicionar ao nosso código:

- `@deprecated` : especifica uma função ou um método preterido.
- `@description` : especifica a descrição de uma função ou um método.
- `@param` : especifica informações para um parâmetro em uma função ou método. O TypeScript também dá suporte à marcação `@paramTag` .
- `@property` : especifica informações, incluindo uma descrição, para um campo ou membro definido em um objeto.
- `@returns` : especifica um valor de retorno.
- `@summary` : especifica a descrição de uma função ou método.
- `@type` : especifica o tipo para uma constante ou uma variável.
- `@typedef` : especifica um tipo personalizado.

Para que você possa reforçar tudo o que aprendeu nos capítulos 11, 12 e 13 deste livro, que tal um *overview* de todos os passos que nós demos para a construção da nossa API?

Assim conseguimos entender melhor cada etapa desenvolvida e, conforme a necessidade, podemos documentá-las.

Models

Conforme nós aprendemos no capítulo 11 deste livro, as models são a representação de um conjunto de informações sobre determinado conceito do sistema. Toda model possui atributos, que são as informações que a referenciam.

Para nossa API, nós criamos as seguintes models: `News`, `Videos`, `Galeria`, `Podcast`, `Fotos` e todas elas herdam os atributos de `Core`.

Iniciando pela classe `Core`, atualize-a com o seguinte trecho de código:

```
import { Document } from 'mongoose';

/**
 * @summary Classe abstrata para criação das models
 * @type titulo {String} título
 * @type texto  {String} texto ou descrição
 * @type imagem {String} imagem default ou avatar
 * @type dataPublicacao {Date} data de publicação
 * @type tags {String} tags relacionada a model
 * @type link {String} link do conteudo Ex.: https://conteudo.com
.br
 * @type ativo {Boolean} status
 */
export abstract class Core extends Document {
    titulo: String;
    texto: String;
    imagem: String;
```

```
    dataPublicacao: Date;
    tags: String;
    link: String;
    ativo: Boolean;
}
```

Seguindo o mesmo modelo que nós utilizamos para documentar a classe `Core`, atualize as classes `News`, `Videos`, `Galeria`, `Podcast`, `Fotos` com os seguintes trechos de código.

```
//News
import { Core } from "./core";

/**
 * Model de news
 * @type chapeu {String} titulo menor
 * @type autor {String} quem escreveu a noticia
 */
export class News extends Core {
    chapeu: String;
    autor: String;
}

//Videos
import { Core } from "./core";

/**
 * Model de video
 * @type url {String} url do video
 * @type duração {String} tempo do video
 */
export class Videos extends Core {
    url: String;
    duracao: String;
}

//Galeria
import { Core } from "./core";
import { Fotos } from "./fotos";

/**
 * Model de galeria de fotos
 * @type fotos {Array} lista da model de fotos
 */
```

```
export class Galeria extends Core {
    fotos: Array<Fotos>;
}

//Fotos
/**
 * Model de fotos
 * @type thumb {String} foto principal
 * @type thumbNail {String} foto menor
 * @type credito {String} quem tirou a foto
 * @type legenda {String} descrição da foto
 */
export class Fotos {
    thumb: String;
    thumbNail: String;
    credito: String;
    legenda: String;
}

//Podcast
/* videos.ts*/
import { Core } from "./core";

/**
 * Model de podcast
 * @type url {String} url do podcast
 * @type duração {String} tempo do audio
 */
export class Podcast extends Core {
    url: String;
    duracao: String;
}
```

No diretório Repository

No diretório `Repository`, nós mapeamos as nossas `models` com as nossas collections do MongoDB. Analisando esses arquivos, não há a necessidade de documentá-los por serem arquivos de mapeamento entre as models e as nossas collections. Mas, para reforçar o nosso conhecimento sobre essa etapa, vamos abrir um dos arquivos e entendê-lo melhor.

```
import mongoose from "mongoose";
import { News } from "../models/news";

const NewsSchema = new mongoose.Schema<News>({
  titulo: { type: String },
  chapeu: { type: String },
  texto: { type: String },
  autor: { type: String },
  imagem: { type: String },
  dataPublicacao: { type: Date },
  tags: { type: String },
  link: { type: String },
  ativo: { type: Boolean }
});

export const NewsRepository = mongoose.model<News>("news", NewsSc
hema);
```

Analisando o arquivo `NewsSchema` nós temos:

- Estamos importando no início do arquivo o pacote do `mongoose` e a classe `News`;
- Em seguida, nós criamos uma constante que está criando um schema de `News`, passando todos os atributos que a nossa collection deve ter;
- No final do arquivo, nós exportamos a constante `NewsRepository`.

Contracts

No diretório `contracts`, nós criamos as nossas interfaces, ou, como aprendemos neste livro, os nossos contratos.

Nesse diretório, nós temos os seguintes arquivos: `iGaleriaService.ts`, `iNewsService.ts`, `iPodcastService.ts`, `iVideosService.ts` e `iService.ts`, que é uma interface genérica desenvolvida com dois métodos,

`get` e `getAll`, que devem ser implementados pelas outras interfaces.

A seguir você tem o código atualizado com o JSDoc de cada uma dessas interfaces:

```
//iService.ts
import { Result } from "../infra/result";

/**
 * Interface genérica para retorno de pesquisas
 */
export interface IService<T> {
    /**
     * @summary busca por id
     * @param id {String}
     * @returns retorna o resultado de uma busca pelo seu id
     */
    get(id: string): Promise<T>;
    /**
     * @summary Realiza uma busca paginada de uma model
     * @param page {number} pagina
     * @param qtd  {Number} quantidade de itens
     * @returns retorna uma lista de T onde T é uma model
     */
    getAll(page: number, qtd: number): Promise<Result<T>>;
}

//iGaleriaService.ts
import { Galeria } from "../models/galeria";
import { IService } from "./iService";

 /**
  * Contrato IGaleriaService
  * @summary esse contrato implementa a interface IService passan
do a model de Galeria
  */
export interface IGaleriaService extends IService<Galeria> { }

//iNewsService.ts
import { News } from "../models/news";
import { IService } from "./iService";
```

```
/**
 * Contrato INewsService
 * @summary esse contrato implementa a interface IService passand
o a model de News
 */
export interface INewsService extends IService<News> { }

//iPodcastService.ts
import { Podcast } from "../models/podcast";
import { IService } from "./iService";

/**
 * Contrato IPodcastService
 * @summary esse contrato implementa a interface IService passand
o a model de Podcast
 */
export interface IPodcastService extends IService<Podcast> { }

//iVideosService.ts
import { Videos } from "../models/videos";
import { IService } from "./iService";

/**
 * Contrato IVideosService
 * @summary esse contrato implementa a interface IService passand
o a model de Podcast
 */
export interface IVideosService extends IService<Videos> { }
```

No diretório Services

No diretório `Services`, implementamos os serviços criados dentro do diretório `contracts`, chamando os nossos `repositories` para criarmos as queries *pesquisas dentro do banco de dados.*

Para ficar mais claro, vamos abrir um dos arquivos dentro desse diretório e detalhá-lo.

```
import { INewsService } from "../contracts/iNewsService";
import { Result } from "../infra/result";
import { News } from "../models/news";
```

```
import { NewsRepository } from "../repository/newsRepository";

export class NewsService implements INewsService {

    async get(_id: string): Promise<News> {
        let result = await NewsRepository.findById(_id);
        return result;
    }

    async getAll(page: number, qtd: number): Promise<Result<News>
> {
        let result = new Result<News>();
        result.Page = page;
        result.Qtd = qtd;
        result.Total = await NewsRepository.count({});
        result.Data = await NewsRepository.find({}).skip((page *
qtd) - qtd).limit(qtd);
        return result;
    }

}
```

- No início do arquivo, estamos importando os pacotes necessários para a criação do nosso serviço;
- Em seguida, nós implementamos os métodos do nosso contrato `INewsService` .

Nós não precisamos documentar esses serviços. Como eles implementam os nossos contratos, nós conseguimos pegar a descrição de cada um dos métodos apenas colocando o mouse sobre um dos métodos, `get` ou `getAll` .

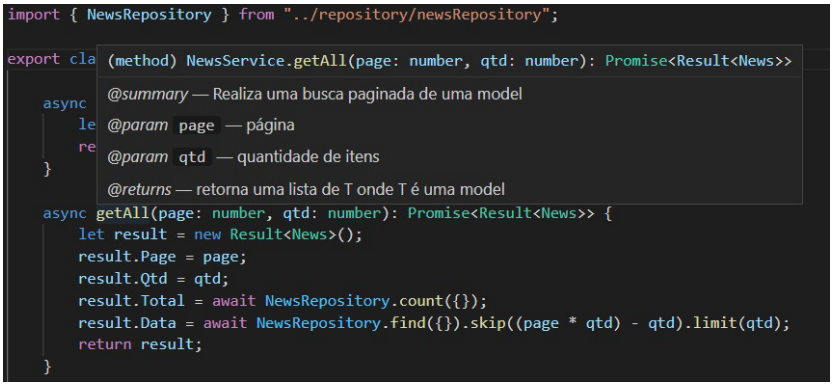

Figura 14.1: NewsService - JSDoc.

Controllers

As nossas controllers são responsáveis por receber todas as requisições dos nossos usuários, se comunicando com as outras camadas do nosso projeto e retornando o que o usuário precisa. Na nossa API, as controllers estão acessando o nosso banco de dados através dos nossos serviços e retornando os dados conforme a requisição do usuário. Mas isso não é uma regra, nós podemos ter outras controllers com outras responsabilidades, como o upload de um arquivo.

Pelo mesmo motivo pelo qual não precisamos documentar os nossos repositórios, nós não precisamos documentar as nossas controllers: elas implementam outros serviços que já estão documentados.

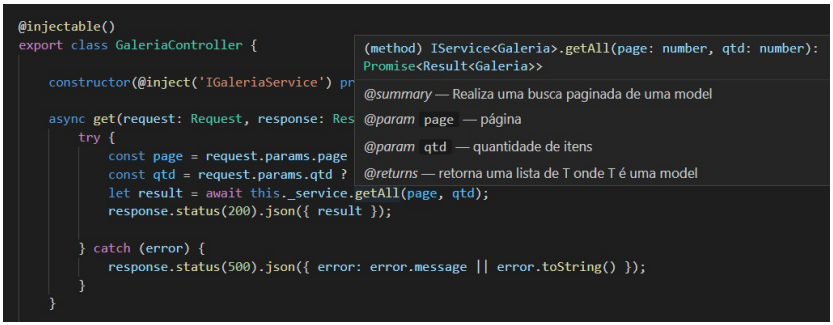

Figura 14.2: GaleriaController - JSDoc.

Arquivos de inicialização do projeto

Nós temos dois arquivos de inicialização do nosso projeto, o `startUp.ts` e o `program.ts`. O `startUp.ts` seria o arquivo configurações do nosso projeto. É nele que devemos inicializar o `express`, o nosso arquivo de conexão com o banco de dados `db.ts` e chamar as nossas rotas.

O `program.ts` é o primeiro arquivo chamado no nosso projeto, ele importa o arquivo `startUp.ts` e inicializa o `express`.

Como a finalidade desses arquivos é importar e inicializar as nossas outras classes, que estão todas documentadas, nós não precisamos documentá-los.

Caso você tenha interesse na versão final do nosso projeto documentado, ela está no meu GitHub: https://github.com/programadriano/api_mc_livro/tree/versao-final.

Capítulo 15

CONCLUSÃO

O objetivo deste livro foi abordar o que é TypeScript e auxiliar você desde a instalação até a criação de um projeto. Ao longo do trajeto, os principais pontos pelos quais passamos foram: tipagem, programação Orientada a Objetos, interfaces, Generics, decorators e Injeção de Dependência.

E para que você esteja alinhado(a) com as tecnologias que estão em alta no mercado, tivemos uma rápida introdução sobre o que é o Docker no capítulo 10 e, nos capítulos seguintes, nós montamos um ambiente de desenvolvimento com ele.

Por fim, no capítulo 14, nós utilizamos o JSDoc para documentar o nosso projeto.

A versão final do projeto aqui desenvolvido pode ser encontrada no seguinte link no meu GitHub: https://github.com/programadriano/api_mc_livro/tree/versao-final.

15.1 OBRIGADO

Espero que este livro tenha lhe ensinado bastante coisa e que você possa colocar em prática todo o aprendizado no seu dia a dia,

seja no desenvolvimento front-end utilizando somente TypeScript com algum framework, como o Angular, ou ainda no desenvolvimento back-end.

Para que possamos ampliar a nossa rede de amigos, a seguir deixo o meu e-mail e minhas redes sociais.

- tadriano.dev@gmail.com
- https://www.linkedin.com/in/tadriano-net/

Eu ficarei muito feliz de receber o seu GitHub com os exemplos deste livro ou com algum outro exemplo que você tenha desenvolvido utilizando TypeScript.