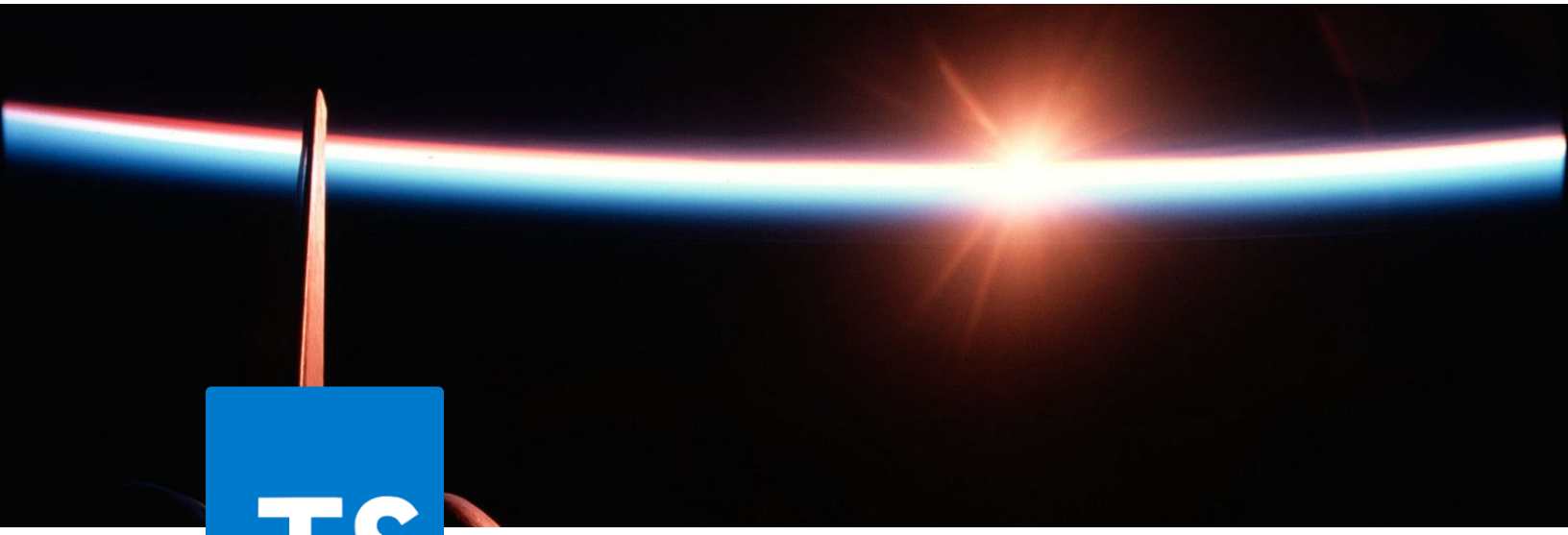


Typescript

Project Configuration (Oficial)

typescript-cheatsheet

React Typescript Cheatsheet

Intro

Fala Dev! O objetivo dessa doc é complementar a aula que disponibilizamos sobre Typescript. Aqui vamos direto ao ponto, basicamente para servir como uma referência rápida (cheatsheet) de como utilizar o **Typescript** em conjunto com o **React**. Então bora?

Create React App

Iniciando um novo projeto

Para iniciar um projeto React + TS utilizando o `create-react-app` , basta utilizar o comando:

```
npx create-react-app my-app --template typescript # ou yarn create react-app my-app --template typescript
```

Adicionando a um projeto existente

Para adicionar o Typescript a um projeto CRA existente, adicione as seguintes libs:

```
npm install --save typescript @types/node @types/react @types/react-dom @types/jest # ou yarn add typescript @types/node @types/react @types/react-dom @types/jest
```

Execute o comando abaixo para gerar o tsconfig.json (arquivo de configuração do Typescript):

```
npx tsc --init # ou yarn run tsc --init
```

Ele virá com as configuração padrão. Caso tenha interesse em saber mais sobre isso, **clique aqui**

E renomeie os arquivos Javascript para Typescript seguindo a regra:

- Caso o arquivo possua extensão `.jsx` ou conteúdo **JSX**, renomeie para `.tsx`
- Caso o arquivo não entre no critério acima, renomeie para `.ts`

Com isso, seu projeto deve começar a acusar os erros de tipagem e você está pronto para utilizar o TS.

Tipos

Básico

Os 3 tipos básicos mais conhecidos são:

- **boolean:** valores `true` ou `false`;

```
const isThisAGoodDoc: boolean = true;
```

- **number:** valores numéricos;

```
const fightingPower: number = 9001;
```

- **string:** valores textuais;

```
const rocketseat: string = "Are you ready for launch?";
```

Além dessas, temos outras tipagens básicas que não muito convencionais:

- **any:** aceita qualquer valor. Utilizado quando não queremos fazer a checagem do tipo;
- **void:** é basicamente o oposto de `any`, utilizado principalmente para demarcar quando não queremos retornar valores de uma função (mesmo assim, ao utilizar `void` a função irá retornar `undefined`, explicitamente ou implicitamente);
- **null:** aceita valores do tipo `null`;
- **undefined:** aceita valores do tipo `undefined`;
- **never:** não aceita nenhum tipo, utilizada principalmente para funções que **nunca** devem retornar algo (funções sem retorno retornam `undefined`, por isso usamos `void`) como loops infinitos ou excessões.

Avançado

Não se deixe enganar pelo título dessa seção, avançado não significa complexo. A partir das tipagens que vimos anteriormente, podemos utilizar alguns recursos do **Typescript** para expandir as tipagens do nosso código. As mais utilizadas são:

Arrays

Temos duas formas principais de declará-los: adicionando [] ao final do tipo ou utilizando o **generic** mais sobre isso nas próximas seções) `Array<T>`. Exemplo:

```
const educationTeam: string[] = ["Vini", "Dani", "Doge", "Claudião", "Graciano"]; const educationTeam: Array<string> = ["Vini", "Dani", "Doge", "Claudião", "Graciano"];
```

Tuples

Utilizado quando queremos trabalhar com arrays que sabemos exatamente quantos elementos ele terá, mas que não serão necessariamente do mesmo tipo. Exemplo:

```
const eitaGiovanna: [string, boolean] = ["O forninho caiu?", true]
```

Onde temos um array com 2 elementos, onde o primeiro é uma `string` e o segundo um `boolean`.

Enums

Utilizado quando queremos dar um nome mais amigável a um conjunto de valores. Exemplo:

```
enum Techs { React, Angular, Vue }; const theBest: Techs =
Techs.React; console.log(theBest) // Irá printar o valor 0
```

Objetos

Apesar de ser possível descrever um objeto utilizando simplesmente o `object`, não é recomendado pois dessa forma não conseguimos definir os campos, a sua forma (shape).

Funções

No caso das funções, precisamos definir a tipagem dos argumentos e do retorno. Exemplos:

```
function overkillConsoleLog(arg1: string, arg2: number): void {
console.log(arg1, arg2); }
```

```
function anotherCallbackExample(callback: (arg: number) => string):
string { return callback(9); }
```

No primeiro exemplo temos uma função chamada **overkillConsoleLog** que recebe dois argumentos: **arg1** é uma `string` e **arg2** é um `number`. Como não queremos retornar nenhum valor da função, atribuímos o tipo `void` **ao retorno.**

No segundo exemplo, declaramos uma função chamada **anotherCallbackExample** que recebe um parâmetro **callback** que representa uma função. Essa função recebe um argumento chamado **arg** do tipo **number** e retorna uma `string`. Como na função **anotherCallbackExample** estamos retornando diretamente o valor de **callback**, atribuímos também ao retorno dela o tipo `string`.

Interfaces

Lembra que falamos que representar um objeto como `object` não é legal? É aí que as interfaces entram e nos ajudam (bastante). Exemplo:

```
interface EveryExampleInOne { str: string; num: number; bool: boolean;
func(arg1: string): void; arr: string[]; }
```

Onde temos uma interface **EveryExampleInOne** que possui 5 propriedades. Elas possuem, respectivamente, os seguintes tipos:

1. string
2. number
3. boolean
4. Função que recebe um argumento do tipo `string` e tem como retorno o tipo `void`
5. Array de strings

Optional Properties

Uma possibilidade interessante nas interfaces é definir uma propriedade como opcional. Exemplo:

```
interface Dog { name: string; owner?: string; }
```

Onde temos que o nome do cachorro é obrigatório, mas o nome do dono é opcional.

Dynamic Properties

Além disso, outro caso interessante é quando além das propriedades que declaramos, queremos deixar em aberto que novas propriedades de um certo tipo sejam sejam adicionadas. Exemplo:

```
interface User { name: string; email: string; [propName: string]:
string; }
```

Onde temos uma interface **User** na qual, além das 2 propriedades que definimos, deixamos em aberto a possibilidade de **N** novas propriedades de nome (propName) `string` cujo valor também é do tipo `string`. Poderíamos implementar algo do tipo:

```
const doge: User = { name: "Joseph Oliveira", email:
"doge@rocketseat.com.br", nickname: "Dogim", address: "Dogeland" }
```

Readonly Properties

Além disso, podemos também definir que uma propriedade é apenas para leitura, pode atribuir um valor a ela apenas uma vez. Segue um exemplo:

```
interface Avengers { readonly thanos: string; } let theEnd: Avengers =
{ thanos: "I'm inevitable" } theEnd.thanos = "I'm not inevitable" //
erro
```

Implements

Utilizando conceitos já comuns em linguagens tipadas como C# e Java, temos a possibilidade de reforçar que uma classe (ou uma função) atenda os critérios definidos em uma interface. Exemplo:

```
interface BalanceInterface { increment(income: number): void;
decrement(outcome: number): void; } class Balance implements
BalanceInterface { private balance: number; constructor() {
this.balance = 0; } increment(income: number): void { this.balance +=
income; } decrement(outcome: number): void { this.balance -= outcome;
} }
```

Lembrando que ao utilizar o `implements` para que a interface force a classe a seguir os padrões impostos, só conseguimos referenciar o lado público (`public`) da classe.

Extends

Outro conceito importante já apresentado nessas linguagens é a possibilidade de uma interface herdar propriedades de outra interface. Exemplo:

```
interface Aircraft { speed: number; } interface Fighter extends
Aircraft { hasMissiles: boolean; missiles?: number; } const f22:
Fighter = { speed: 2000, hasMissiles: true, missiles: 4, };
```

Union Types

Em alguns casos, queremos que uma variável/propriedade aceite mais de um tipo. Para esses casos, utilizamos os **Union Types**. Exemplo:

```
let age: number | string = 30; age = "30"; age = false; // erro
```

Generics

Vimos diversas formas até agora de como realizar a tipagem com Typescript, até mesmo em casos mais complexos como funções e objetos. Mas e se, por exemplo, não soubermos, durante o desenvolvimento, qual tipo o argumento e o retorno de uma função devem receber? Para isso utilizamos os **Generics.** Exemplo:

```
const mibr: Array<string> = ["Fallen", "Fer", "Taco", "Kng", "Trk"];
```

Nesse simples exemplo utilizamos um **generic** do próprio Typescript, o **Array**, em que o tipo informado dentro de `<>` representa o tipo dos valores do array. É o equivalente de `string[]`. Agora vamos a um exemplo mais complexo:

```
function example<T>(arg: T): T { return arg; }
```

Nesse caso, declaramos uma função **example** que recebe um argumento do tipo `T` e retorna um valor do tipo `T`. Então:

```
const value = example<string>("rocketseat"); console.log(value) // irá printar o valor "rocketseat"
```

Type assertions

As vezes, você pode saber mais de um tipo do que o próprio Typescript, principalmente ao trabalho com tipos como `any` ou `object`. Por isso, é possível atribuir manualmente um tipo utilizando **Type assertions**. Exemplo:

```
const bestDog: any = "Doge"; const dogLength: number = (bestDog as string).length;
```

Onde atribuímos manualmente o tipo `string` a variável `bestDog` utilizando o `as` (anteriormente do tipo `any`).

Utility Types

Muitas vezes, em uma mesma aplicação acabamos gerando interfaces que possuem muitas semelhanças mas que não são necessariamente iguais. Isso, além de causar um código mais verboso, também é mais trabalhoso e suscetível a erros. Por isso, o Typescript disponibiliza os **Utility Types**. Eles vêm com a missão de evitar esses problemas e gerar rapidamente interfaces a partir de outras pre-existentes. Nessa seção iremos falar de dois exemplos apenas, mas fique a vontade para olhar o restante **aqui**.

Pick<T, K>

Utilizado quando queremos pegar apenas algumas propriedades (K) de uma outra interface (T). Exemplo:

```
interface Video { title: string; description: string; fps: number;
duration: number; } type Image = Pick<Video, 'title' | 'description'>;
const picture: Image = { title: 'Profile', description: "Picture taken
for my driver's license", };
```

Omit<T, K>

Utilizando quando queremos excluir apenas algumas propriedades (K) de uma outra interface (T). Exemplo:

```
interface Video { title: string; description: string; fps: number;
duration: number; } type Image = Omit<Video, 'fps' | 'duration'>;
const picture: Image = { title: 'Profile', description: "Picture taken
for my driver's license", };
```

Extras

Acreditamos que com o que foi passado até aqui, você tem boa uma base e fonte de consulta para realizar os seus projetos utilizando Typescript. Porém, esse é um mundo vasto e apenas com a prática e estudo você vai saber melhor do que precisa saber. Por isso, deixaremos nas seções abaixo links que podem te ajudar.

DefinitelyTyped (@types)

Atualmente, é cada vez mais comum utilizarmos libs que possuem a tipagem embutida no próprio pacote (por exemplo uma pasta `types` com um arquivo `index.d.ts`), como é o caso do **Knex** . Porém, ainda existem diversas libs que não possuem tipagem embutidas no próprio pacote (como é o caso do **React**). Para muitos desses casos, temos um projeto bem legal conhecido como **DefinitelyTyped** que fornece a tipagem correta da lib (é ele que fornece os famosos pacotes do npm `@types`). Segue abaixo o link do repositório do projeto no Github:

DefinitelyTyped/DefinitelyTyped

The repository for high quality TypeScript type definitions. You can also read this README in Spanish, Korean, Russian, Chinese and

https://github.com/DefinitelyTyped/DefinitelyTyped

O que não fazer

É normal ficar em dúvida do que não se deve fazer ao utilizar os recursos que o Typescript adiciona ao Javascrpt, principalmente se for seu primeiro contato com linguagem tipada. Por isso, deixamos abaixo um link que a própria equipe do TS disponibilizou no site oficial para auxiliar nesses casos.

Function Overloads #

Don't ever use the types Number, String, Boolean, Symbol, or Object These types refer to non-primitive boxed objects that are almost never used appropriately in JavaScript code. /* WRONG */ function reverse(s: String):

https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html

Referências

Como essa é uma documentação básica e bem voltada a prática, é inviável tratar de todas as peculiaridades do Typescript. Por isso, deixaremos abaixo links que podem te ajudar a sanar eventuais dúvidas não tratadas aqui:

Handbook (Oficial)

A note about let #

https://www.typescriptlang.org/docs/handbook/basic-types.html

Declaration Files (Oficial)

Sections #

This guide is designed to teach you how to write a high-quality TypeScript Declaration File. In this guide, we'll assume basic familiarity with the TypeScript language. If you haven't already, you should read the TypeScript

https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html

Project Configuration (Oficial)

tsconfig.json

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

typescript-cheatsheet

typescript-cheatsheet

A set of TypeScript related notes used for quick reference. The cheatsheet contains references to types, classes, decorators, and

https://rmolinamir.github.io/typescript-cheatsheet/

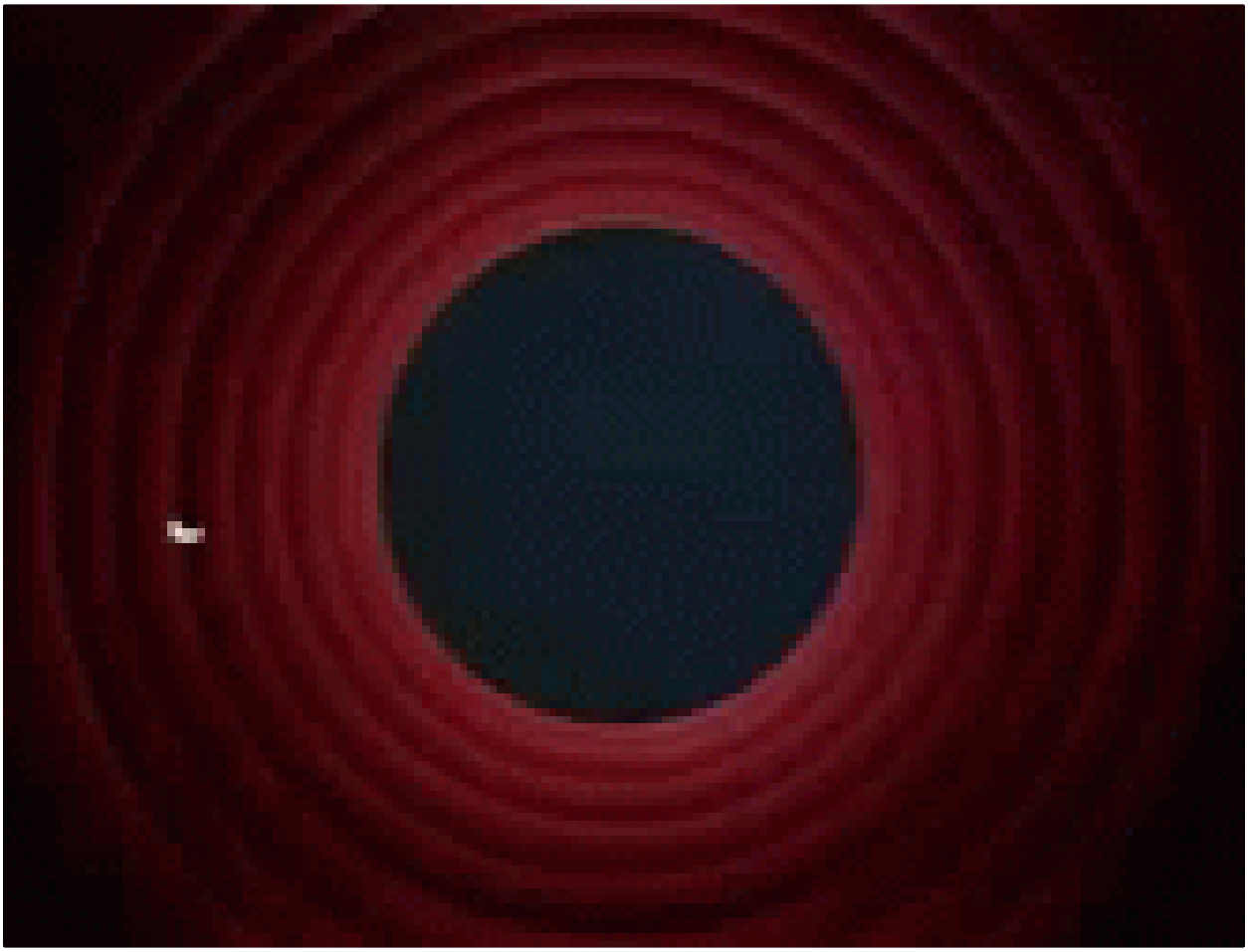
React Typescript Cheatsheet

React TypeScript Cheatsheets | React TypeScript Chea…

React TypeScript Cheatsheets

https://react-typescript-cheatsheet.netlify.app/

É isso dev, esperamos que tenha gostado da doc e que entenda o poder que o Typescript pode adicionar ao Javascript. Só não vai botar `any` em tudo hein?



https://www.notion.so/Typescript-5712aeab312d44fcba0aa88895caad36

Caíque de Oliveira Gonçalves