

Definição de uma rota de API usando TypeScript

André Yuji Sakuma

RA: 11201920463

O TypeScript é uma linguagem de programação que introduz tipagem estática ao JavaScript. O que permite especificar tipos para variáveis, parâmetros de função, propriedades de objetos e mais, garantindo maior segurança e prevenindo erros durante o desenvolvimento.

Por baixo dos panos, o ele é apenas uma camada acima que fica por cima do JavaScript, tanto que no processo de build ele é transpilado para JavaScript, removendo-se os tipos, portanto, para o ambiente de execução o código executado é em JavaScript.

Essa capacidade de adicionar tipos estáticos ao JavaScript torna o TypeScript uma ferramenta poderosa para o desenvolvimento de aplicativos robustos e escaláveis, especialmente no ecossistema web. Com ele, os desenvolvedores podem ter a garantia da segurança que os tipos adicionam, melhorando a legibilidade e a manutenibilidade do código.

Geralmente os Frameworks utilizados para a implementação dessas APIs (express, fastify, ...) eles lidam com as tipagens dos parâmetros e respostas das rotas de forma bem genérica, o que pode ser chato para o desenvolvedor durante o desenvolvimento ou manutenção de uma rota.

Com esse problema em mente, nesse tutorial vamos usar o TypeScript para implementar a definição de um endpoint de uma API Rest. Assim, poderemos ter a definição da rota e usá-la para inferir as tipagens de parâmetros e do retorno para a função que vai lidar com a rota(handler)

Estou definindo uma rota HTTP com as seguintes propriedades:

- Método (os mais comuns são: 'GET', 'POST', 'PUT' e 'DELETE')
- Path (string que indica qual o caminho para aquele endpoint)
- Parâmetros: esse pode ser passado de 3 maneiras
 - Body: com exceção do método GET, os outros podem receber um body que é um objeto JSON
 - Query: parâmetros passados pela query
 - ex: <https://api.teste/users?name=123>
 - Path: parâmetros passados no próprio path da rota
- Retorno: objeto JSON de resposta da API

Seguindo o Express como exemplo, podemos definir uma rota da seguinte maneira:

```
app.get('/users/:userId', (req, res) => {  
    ...  
})
```

sendo o primeiro parâmetro o `path` e o segundo a função que lidará quando a rota for executada, que iremos chamar de `handler`

Uma das dores dos desenvolvedores é que não é possível saber quais são os parâmetros que deveriam ser passados e nem qual deve ser o retorno do handler, pois no Express, com a exceção dos parâmetros passados no path (`req.params`), ele lida com a tipagem de forma bem genérica. Por exemplo:

```
app.get('/users/:userId', (req, res) => {  
    req.params.userId // funciona e me dá o valor de :userId como  
    string  
    req.query.qualquercoisa // também é permitido durante o  
    desenvolvimento, mas em execução essa propriedade não existe  
})
```

Isso não dá nenhuma segurança para o desenvolvedor de que ele não errou algum nome da propriedade, ou aquela propriedade realmente será passada para ele.

Então o objetivo desse trabalho é ao final conseguirmos definir uma rota e a partir dessa definição garantir a sua tipagem:

```
const route = defineRoute(  
    'POST',  
    '/users/:userId',  
    {  
        query: {},  
        params: {  
            userId: 'string',  
        },  
        response: {  
            id: 'string',  
        },  
        body: {  
            name: 'string',  
            age: 'number',  
            roles: 'string[]',  
            location: {  
                city: 'string',  
                country: 'string',  
            },  
        },  
    },  
    (context) => {
```

```

const query = context.query // type: {}
const params = context.params // type: {userId: string}
const body = context.body

//                                     type:
//                                     {
//                                     name:
"string";
//                                     age:
"number";
//                                     roles:
"string[]";
//                                     location:
{
//                                     city:
"string";
//                                     country:
"string";
//                                     };
//                                     }

const location = body.location
// type:
// {
//     city: "string";
//     country: "string";
// }
const city = location.city // type: string
const roles = body.roles // type: string[]

return {
    id: '123', // só é permitido retornar um objeto {
id: string } (dado a definição que foi passada), qualquer outra coisa ele
apontaria um erro de tipagem
}
}
)

```

Método

para tipar os métodos, podemos definir a tipagem da seguinte maneira:

```

type HttpMethod = 'GET' | 'POST' | 'PUT' | 'DELETE'

```

Para o TypeScript um tipo pode ser considerado um conjunto, então o operador `|` indica a união dos conjuntos, ou seja, o tipo `HttpMethod` pode ser uma string com valor `GET`, `POST`, `PUT` ou `DELETE`.

Entretanto, vale lembrar que tipos não podem ser usados como valores, então se posteriormente quisermos validar se uma string é um método válido, não iremos conseguir

já que a tipagem não interfere no runtime.

Assim, podemos ter um array com os métodos válidos e inferir a tipagem a partir deles:

```
const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE']
```

Só que ao ver a tipagem de `HTTP_METHODS` podemos ver que será `string[]`. Isso acontece, porque o ts infere a tipagem de cada um dos elementos como uma `string` e não com o valor literal.

```
type P<REST>
const HTTP_METHODS: string[]
const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE']
```

Para ele assumir o valor literal, precisamos dizer para ele que não iremos editar os valores dentro do array e usá-lo apenas como uma constante.

```
const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE'] as const
```

```
type P<REST>
const HTTP_METHODS: readonly ["GET", "POST", "PUT", "DELETE"]
const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE'] as const
```

Agora podemos criar um tipo que é a união dos possíveis elementos do array:

```
const HTTP_METHODS = ['GET', 'POST', 'PUT', 'DELETE'] as const
```

```
type HttpMethod = (typeof HTTP_METHODS)[number]
```

```
const type HttpMethod = "GET" | "POST" | "PUT" | "DELETE"
type HttpMethod = (typeof HTTP_METHODS)[number]
```

o `typeof` é um operador que retorna o tipo de um dado.

A partir do tipo podemos acessar os tipos dos elementos passando `[number]` sendo `number` o tipo das chaves, que no nosso caso é `number` por ele ser um array.

Parâmetros de rota (RouteParams)

os `RouteParams` são definidos e passados no path de uma rota como por exemplo:

```
type params = RouteParams<'/users/:userId/:name/:age'>
```

O objetivo nessa seção é a partir da string inferir quais são os parâmetros que iremos receber, assim podemos saber quais são esses parâmetros dentro do handler:

```
type params = {
  userId: string;
} & {
  name: string;
} & {
  age: string;
}

: Quick Fix... (⌘.)

type params = RouteParams<'/users/:userId/:name/:age'>
```

Antes de seguirmos, precisamos dar um passo atrás e entender alguns conceitos do typescript.

O primeiro deles é o Generics, que de forma bem simples, dado um tipo o generics permite a definição de um tipo genérico que será usado

```
type Array<T> = T[]

type NumberArray = Array<number> // number[]
```

A partir dele, podemos realizar algumas operações em cima do tipo. Com o `extends` podemos restringir um tipo genérico.

```
function getStringArray<T extends string>(str: T): T[] {
  return [str]
}
```

essa função pode receber qualquer parâmetro em que o tipo esteja dentro do conjunto `string`

Outra operação que é comum de se fazer com os tipos é a condicional, que possui a mesma sintaxe dos ternários:

```
type StringOrNumberArray<T extends string | number> = T extends string
  ? string[]
  : T extends number
  ? number[]
  : never
```

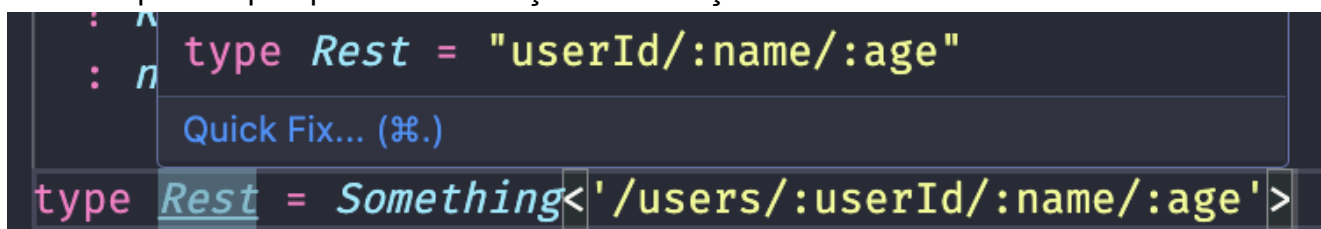
com isso podemos fazer operações mais complexas com os tipos.

o tipo `never` é um caso que não pode acontecer, caso ocorra, o TypeScript aponta como erro.

Por fim, podemos extrair tipos de strings com o `infer`

```
type Something<SomethingBefore extends string> = SomethingBefore extends  
  `${string}:${infer Rest}`  
    ? Rest  
    : never  
  
type Rest = Something<`/users/:userId/:name/:age`>
```

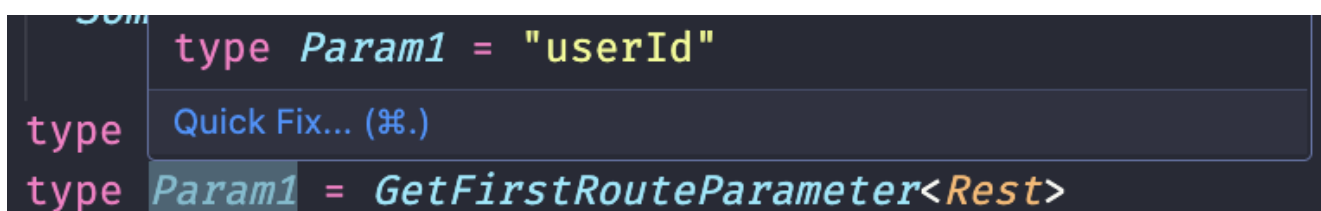
Com isso, pegamos tudo aquilo que vem depois da primeira ocorrência do `:` o que já é um caminho para o que queremos alcançar nessa seção.



```
type Rest = "userId/:name/:age"  
  
type Rest = Something<`/users/:userId/:name/:age`>
```

Então, voltando para o nosso problema, temos que remover o que vem depois do `/`

```
type RemoveTail<  
  S extends string,  
  Tail extends string  
> = S extends `${infer P}${Tail}` ? P : S  
  
type GetFirstRouteParameter<S extends string> = RemoveTail<S,  
  `/${string}`>  
type Param1 = GetFirstRouteParameter<Rest>
```



```
type Param1 = "userId"  
  
type Param1 = GetFirstRouteParameter<Rest>
```

Só que como podemos ter mais de um RouteParam, temos que aplicá-la recursivamente. Então chegamos em algo como:

```
type RouteParams<Route extends string> =  
  Route extends `${string}:${infer Rest}`  
    ? (GetFirstRouteParameter<Rest> extends never // condição de  
      parada: se não for possível pegar o parâmetro, quer dizer que no restante  
      da string não há mais parâmetros definidos  
      ? {}  
      : RouteParams<Rest>  
    : Route
```

```

: { [P in GetFirstRouteParameter<Rest>]: string } )& // o
in especifica que a chave desse objeto é do tipo
GetFirstRouteParameter<Rest> e seu valor é do tipo string
// o operador & faz a intersecção dos dois conjuntos, ou
seja, ele concatena o resultado com os próximos resultados da recursão
(Rest extends `${GetFirstRouteParameter<Rest>}${infer
Next}`
? RouteParams<Next> // chamada de recursão
: unknown)
: {}

```

```

type params = {
  userId: string;
} & {
  name: string;
} & {
  age: string;
}

type params = RouteParams<`/users/:userId/:name/:age`>

```

The screenshot shows a code editor with a dark theme. It displays a TypeScript type definition for `params` as an intersection of three objects: one with `userId: string`, and two others with `name: string` and `age: string`. Below this, the `params` type is used to define `RouteParams` for a specific route pattern: `RouteParams<`/users/:userId/:name/:age`>`. A 'Quick Fix' tooltip is visible at the bottom left of the code block.

Agora, para os parâmetros da query, do body e o tipo do retorno, iremos usar o mesmo tipo que será gerado a partir de uma definição. Gerar um tipo a partir de um dado, nesse caso, pode ser mais interessante nesse caso do que definir com o próprio tipo, pois isso pode ser usado posteriormente na runtime ou aplicado na geração da documentação da rota. Por isso, seguiremos dessa forma.

```

const typeDefinitions = [
  'string',
  'string[]',
  'number',
  'boolean',
  'number[]',
] as const
// definimos aqui como string quais serão os tipos aceitos
type TypeDefinition = (typeof typeDefinitions)[number]
// geramos o tipo

```

Usamos o generics junto com os ternários para transformar o tipo de uma string literal em um tipo do próprio ts.

```

type GetTypeFromDefinition<T extends TypeDefinition> = T extends 'string'
? string

```

```

: T extends 'string[]'
? string[]
: T extends 'number'
? number
: T extends 'number[]'
? number[]
: T extends 'boolean'
? boolean
: never

```

Como um objeto pode ter outros objetos dentro de uma propriedade dele, temos que fazer um tipo recursivo para ele, também.

Então, se ele for uma definição de tipo, aplicamos o `GetTypeFromDefinition`, caso contrário, ele deve ser tratado como objeto, então passamos ele recursivamente para cada uma das propriedades desse objeto.

```

type GetFieldType<T extends FieldType | TypeDefinition> =
  T extends TypeDefinition
  ? GetTypeFromDefinition<T>
  : T extends FieldType
  ? ObjectDefinitionType<T>
  : never

type FieldType = {
  [key in string]: TypeDefinition | FieldType
}

type ObjectDefinitionType<TSchema extends FieldType> = {
  [key in keyof TSchema]: GetFieldType<TSchema[key]>
}

```



```

    ) = type test = {
    ) {   name: string;
    ret  age: number;
    p    roles: string[];
    m    location: ObjectDefinitionType<{
    h      city: "string";
    s      country: "string";
    }>;
  }
}

```

Quick Fix... (⌘.)

```

type test = ObjectDefinitionType<{
  name: 'string'
  age: 'number'
  roles: 'string[]'
  location: {
    city: 'string'
    country: 'string'
  }
}>

```

Por fim, foram definidos alguns outros métodos auxiliares

```

type HasBody<TMethod extends HttpMethod> = TMethod extends 'POST' | 'PUT'
  ? true
  : false

type GetRequestBody<
  TMethod extends HttpMethod,
  T extends FieldType
> = HasBody<TMethod> extends true ? ObjectDefinitionType<T> : never

type ResponseBody<T extends FieldType | void> = T extends FieldType
  ? ObjectDefinitionType<T>
  : void

type Context<RouteParams = {}, QueryParams = {}, RequestBody = {}> = {
  params: RouteParams
  query: QueryParams
  body: RequestBody
}

```

Assim, finalmente, podemos definir o nosso `defineRoute`

```
function defineRoute<
    // aqui é importante definir cada um dos parâmetros e aplicar o
    // extends, se não ele generaliza o tipo para um FieldType e não exatamente o
    // que foi passado como parâmetro
    TMethod extends HttpMethod,
    TPath extends string,
    TQuery extends FieldType,
    TResponse extends FieldType,
    TBody extends FieldType
>(
    method: TMethod,
    path: TPath,
    schema: {
        query: TQuery
        params: RouteParams<TPath> // com isso podemos validar que
        // o path está condizente com os params definidos no schema
        response: TResponse
        body: HasBody<TMethod> extends true ? TBody : never //
        // aqui validamos que se o método não tiver body na requisição, ele não pode
        // ser definido no schema
    },
    handler: (
        context: Context<
            RouteParams<TPath>,
            ObjectDefinitionType<TQuery>,
            GetRequestBody<TMethod, TBody>
        >
    ) => ResponseBody<TResponse>
) {
    return {
        path,
        method,
        handler,
        schema,
    }
}
```

Com ela podemos definir essa rota, por exemplo:

```
const route = defineRoute(
    'POST',
    '/users/:userId',
    {
        query: {},
        params: {
            userId: 'string',
        },
    },
    (context) => {
        // ...
    }
)
```

```

    },
    response: {
        id: 'string',
    },
    body: {
        name: 'string',
        age: 'number',
        roles: 'string[]',
        location: {
            city: 'string',
            country: 'string',
        },
    },
},
(context) => {

    return {
        id: '123', // só é permitido retornar um objeto {
id: string } (dado a definição que foi passada), qualquer outra coisa ele
apontaria um erro de tipagem
    }

}
)

```

E todos os tipos passados no context ficam condizentes dentro do handler.

```

    const body: ObjectDefinitionType<{
        name: "string";
        age: "number";
        roles: "string[]";
        location: {
            city: "string";
            country: "string";
        };
    }>

    const
    const
    const body = context.body

```

Código fonte

<https://github.com/andre-sakuma/ts-define-route/tree/master>