CONFIGURANDO E INICIANDO PROJETO NODE.JS COM TYPESCRIPT

1. Criando a pasta para o projeto

Criei uma pasta com o comando **mkdir nodejs-typescript-project**

2. Entre na pasta **cd <nome-da-pasta>** e rodei o comando **npm init -y** para iniciar um arquivo **package.json** com algumas configurações padrões para dar início ao projeto.

3. Instalando o pacote **Typescript**

   Instale o pacote typescript, com o comando **npm install --save-dev typescrip**t, que vai ser quem vai interpretar nosso código em typescript, no modo de desenvolvimento.

4. Criando o arquivo **tsconfig.json**

   Crie o arquivo **tsconfig.json**, na raiz do projeto, para configurar as opções para compilar o projeto na linguagem typescript.

```
{
      "compilerOptions": {
          "target": "es2019",
          "moduleResolution": "node",
          "module": "commonjs",
          "lib": [
              "es2019"
          ],
          "sourceMap": true,
          "outDir": "dist",
          "strict": true,
          "noImplicitAny": true,
          "strictNullChecks": true,
          "strictFunctionTypes": true,
          "noImplicitThis": true,
          "resolveJsonModule": true,
          "alwaysStrict": true,
          "removeComments": true,
          "noImplicitReturns": true,
          "esModuleInterop": true,
          "allowSyntheticDefaultImports": true,
          "baseUrl": ".",
```

```
        "paths": {
            "@src/*": [
                "./src/*"
            ],
            "@modules/*": [
                "./src/app/modules/*"
            ],
            "@test/*": [
                "./test/*"
            ]
        },
        "rootDirs": [
            "./src",
            "./test"
        ],
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    },
    "include": [
        "./src/**/*.ts",
        "./test/**/*.ts"
    ],
    "exclude": [
        "./node_modules/*",
        "dist"
    ]
}
```

5. **Instalando o express**

   Instale o express, com o comando **npm install express**, que é um framework bastante utilizado que possui métodos utilitários HTTP e middlewares.

6. **Instalando os types do express**

   Instale também os types do express, em modo de desenvolvimento, que servirá para podermos utilizar o express com typescript, com o comando **npm install --save-dev @types/express**.

7. **Instalando o body-parser**

   Instale o body-parser, que é um middleware que faz o parse do body pra gente, com o comando **npm install body-parser**.

8. **Instalando o ts-node-dev**

   Instale o ts-node-dev, para rodar o typescript no modo de desenvolvimento, com o comando **npm install --save-dev ts-node-dev**

9. Instalando o **module-alias**

   Instale também um pacote que ajuda a diminuir a declaração dos caminhos até os arquivos, que é o module-alias, com o comando **npm install module-alias** e os types dele também, com o comando **npm install --save-dev @types/module-alias**.

Arquivo que inicia as configurações da aplicação.

10. **Criando o app.ts**

Crie uma pasta **src/** e dentro dela criei um arquivo **app.ts** com a extensão do arquivo sendo **.ts** porque é a **typescript** que vai compilar esse arquivo e os outros, para transformá-los em javascript no momento do build.

Nesse arquivo **app.ts** implementei uma classe **SetupApplication** para inicializar algumas configurações do projeto:

```
import
'./util/module-
alias';
                import express from 'express';
                import bodyParser from 'body-parser';
                import { Server } from 'http';
                import router from './routes';


                export class SetupApplication {
                  private server?: Server;


                  constructor(private port = 3000, public app = express()) { }


                  public init(): void {
                    this.setupExpress();
```

```
    this.setupRoutes();
  }


  private setupRoutes(): void {
    this.app.use(router);
  }


  private setupExpress(): void {
    this.app.use(bodyParser.json());
    this.app.use(bodyParser.urlencoded({ extended: true }));
  }


  public start(): void {
    this.server = this.app.listen(this.port, () => {
      console.log(`Server running on port ${this.port}`);
    });
  }
}
```

Com a utilização da linguagem typescript, podemos utilizar na classe, a definição do tipo de retorno dos métodos, que nesse caso o ficou **void**, pois não retornam nenhum valor, e também a definição de atributos, como foi definido o atributo privado server, com uma definição de tipo também, que no caso é do tipo Server.

11. **Criando o server.ts**

Nessa mesma pasta **/src** eu crie um arquivo **server.ts** para ser responsável por inicializar o servidor da aplicação para nós. Essa separação entre o setup da aplicação e do server ajuda bastante na implementação dos testes automatizados.

```
import {
SetupApplication
} from './app';


        class Server {
            static start(): void {
                const application = new SetupApplication(3333);
                application.init();
```

```
                        application.start();
                    }
                }



                Server.start();
```

12. Criando o arquivo **routes.ts**

Crie um arquivo para gerenciar as rotas da aplicação, dentro de **/src** também, o arquivo **routes.ts**, que possui uma classe **Routes** e um método estático **define()** que recebe um parâmetro do tipo **Router** e o seu retorno também é do tipo **Router**

```
import {
Router }
from
'express';
          import ProductRouter from '@modules/Product/Router';
          class Routes {
              static define(router: Router): Router {
                  router.use('/products', ProductRouter);



                  return router;
              }
          }



          export default Routes.define(Router());
```

No método define implementei somente uma rota /products que servirá como prefixo para as rotas que serão definidas dentro do arquivo em @modules/Product/Router.
Falando nisso, tem ali a utilização do pacote module-aliasno segundo import import Product from '@modules/Product/Router', que ajuda na redução do caminho até o arquivo Router lá dentro de app/modules/Product/Router

13. **Criando o arquivo de configuração para o module-alias.ts**

Primeiro vou mostrar a configuração que precisa ser implementada para que o module-alias interprete os nomes definidos para os caminhos até as pastas.

Crie um arquivo em **src/util/module-alias.ts** com o seguinte código:

```
import
* as
path
from
'path';
       import moduleAlias from 'module-alias';


       const files = path.resolve(__dirname, '../..');


       moduleAlias.addAliases({
         '@src': path.join(files, 'src'),
         '@modules': path.join(files, 'src/app/modules'),
         '@test': path.join(files, 'test'),
       });
```

*Onde basicamente são adicionados os alias para os caminhos até as pastas, no formato **json**, sendo a chave o nome para o **alias**, e o valor o caminho até a pasta desejada*

Vamos para o arquivo onde deve-se criar uma rota simples, que retorna algo simples, só para deixar funcionando o fluxo de uma requisição.

14. **Criando a rota para /products**

Crie o arquivo **Router.js**, dentro de **src/app/modules/Product/** para ficar separado por módulo e organizado, que será responsável por definir as rotas para esse módulo **Product**.

```
import {
Router,
request,
response }
from
'express';


        const router = Router();


        router.get('/', (request, response) => {
            response.json({
                _id: 'ABC123',
                name: 'Product Name',
                price: 28.90
            });
        });


        export default router;
```

15. **Adicionando scripts para executar o server**

Chegamos ao final, agora para rodar o projeto e poder obter esse json como resposta para a requisição ao **endpoint** que criamos que ficou /products, vamos adicionar os scripts no arquivo **package.json** para depois podermos rodar o projeto no modo desenvolvimento e também fazer o build de **typescript** para **javascript**

16. Então adicionei esses scripts no arquivo **package.json:**

```
"scripts":
{
              "build": "tsc",
              "start": "npm run build && node dist/server.js",
              "dev": "ts-node-dev 'src/server.ts'"
          },
```

*Onde o comando **npm run start** vai executar o build do projeto, que irá criar uma pasta /dist na raiz do projeto, que irá conter o código compilado em javascript de todo o projeto, e depois, nesse mesmo comando, é executado o servidor da aplicação a partir do arquivo já compilado em javascript dentro da pasta /dist, o arquivo server.js.*

*Agora, rodando no terminal o comando **npm run dev** vai iniciar o servidor da aplicação em modo de desenvolvimento, que está definido para rodar na **porta 3333**.*

*Utilize o navegador de sua preferência ou uma plataforma API para desenvolvedores, onde é possível fazer a requisição do tipo GET para a url **http://localhost:3333/products** e, se tudo der certo (se não der, tudo bem, faz parte).*

É para retornar o json que definimos como resposta dessa rota, mais ou menos assim: