Você já leu por aqui como[**converter React.js para TypeScript**](https://medium.com/@oieduardorabelo/convertendo-react-em-typescript-5ae43e6ff13). Falamos também sobre como [**configurar React Native e TypeScript**](https://medium.com/trainingcenter/react-native-e-typescript-6c794a158327)! Você se lembra que até cobrimos como [**utilizar TypeScript sem utilizar TypeScript**](https://medium.com/@oieduardorabelo/typescript-sem-utilizar-typescript-1323ef83b648)? Vou te dizer, é tanto <T>(a: T): U & F que dói a cabeça! 🤕🤒… Pensando nisso, nós tivemos um artigo completo de como [**entender a notação de tipos em TypeScript**](https://medium.com/@oieduardorabelo/typescript-entendendo-a-nota%C3%A7%C3%A3o-de-tipos-9e8c1c89ef62).

Falamos da sintaxe do TypeScript, front-end com React, aplicações nativas com React Native e, dentro do ecossistema JavaScript, só ficou de fora o nosso back-end. 😭

**Atualização - 19 Outubro 2018:**Após o [comentário do leitor](https://medium.com/@brunodutrafranco/boa-tarde-segui-seu-tutorial-e-s%C3%B3-estou-com-uma-d%C3%BAvida-como-eu-configuro-o-modo-debug-no-ce858b93f27a) [Bruno Dutra Franco](https://medium.com/u/fbf5793e07d6), atualizei o repositório adicionando exemplos de “Debugging”. Você pode usar debug in-editor com o VSCode ou a CLI do Node.js e abrir um aba do Google Chrome. Informações completas no README do repositório!

**Atualização - 04 Setembro 2018:** Coloquei o exemplo descrito nesse artigo em um repositório, ficando mais fácil para ver o código completo:

## [oieduardorabelo/node-typescript](https://github.com/oieduardorabelo/node-typescript)

### [Template for bootstrap projects with Node.js and TypeScript - oieduardorabelo/node-typescript](https://github.com/oieduardorabelo/node-typescript)

#### [github.com](https://github.com/oieduardorabelo/node-typescript)

# Node.js ao resgate

Iremos instalar os pacotes básicos para ter uma aplicação Node.js rodando com TypeScript e todas suas vantagens:

$ mkdir node-typescript  
$ cd node-typescript  
$ npm init -y

Vamos instalar as primeiras depêndencias:

$ npm i typescript nodemon ts-node

E rodar o comando init no recém instalado módulo do typescript:

$ ./node\_modules/.bin/tsc --init

Isso irá gerar um arquivo tsconfig.json na sua pasta. Essas são as configurações que serão passadas para o compilador do TypeScript. Ao abrir esse arquivo, ele irá conter comentários e várias opções, parecido com:

https://miro.medium.com/max/60/1*IPFbR-9MhbjrrCj2dfEEJg.png?q=20



Valores padrões do `tsconfig.json`

Vamos editar esse arquivo, ficando dessa maneira:

// node-typescript/tsconfig.json{  
 "compilerOptions": {  
 "esModuleInterop": true,  
 "lib": ["dom", "esnext"],  
 "module": "commonjs",  
 "outDir": "./dist",  
 "strict": true,  
 "target": "es5",  
 // opcionais, mas recomendados  
 // quase um "ESLint" básico  
 "noUnusedLocals": true,  
 "noUnusedParameters": true,  
 "noImplicitReturns": true,  
 "noFallthroughCasesInSwitch": true  
 }  
}

Tem alguma dúvida sobre as chaves e seus valores? Recomendo primeiro tentar pesquisar/Google. Persistindo a dúvida comente aqui embaixo :)

Agora, iremos editar nosso package.json :

// node-typescript/package.json{  
 "name": "node-typescript",  
 "version": "1.0.0",  
 "private": true,  
 "scripts": {  
 // [A]  
 "dev": "nodemon --watch "src/" --exec \"ts-node src/entry.ts\" -e ts"  
 },  
 "dependencies": {  
 "nodemon": "^1.18.4",  
 "ts-node": "^7.0.1",  
 "typescript": "^3.0.3"  
 }  
}

* **[A]: nodemon** irá observar, --watch, o diretório **src** e executar, --exec , o script, "ts-node src/entry.ts", a qualquer mudança de arquivo com extensão ts, -e ts

Antes de seguir em frente, vamos instalar o framework mais querido do Node.js, **express**:

$ npm i express @types/express

Seguindo o comando **"dev"**acima, vamos criar **src/entry.ts:**

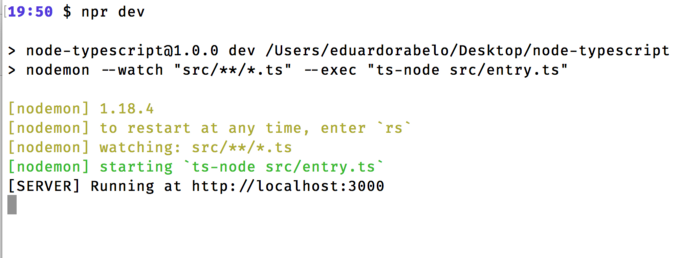
// node-typescript/src/entry.tsimport server from './server';server.listen(3000, () => {  
 console.log(`[SERVER] Running at   
[http://localhost:3000`](http://localhost:3000%60/));  
});

Opa! Esse arquivo está buscando **server** em **"./server"**. Então, vamos criar nosso **src/server.ts**:

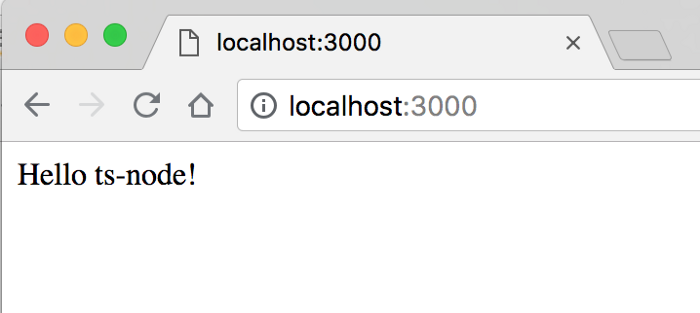
// node-typescript/src/server.tsimport express from "express";const server = express();server.get("/", (\_, res) => {  
 res.send("Hello ts-node!");  
});  
export default server;

Agora, podemos utilizar npm run dev e:

https://miro.medium.com/max/60/1*64aGicZnV1qC7jugONBc8Q.png?q=20



https://miro.medium.com/max/60/1*i4sNYCowZdZ_nqJIsyqVXg.png?q=20



Hooray! 🎉🎉

Se você usa VSCode, a integração com o editor é fantástica, auto-completando seus métodos e acessos de objetos. Ah! Você percebeu que estamos utilizando a sintaxe ES6? ;)

No cap, bro! 😎

# Adicionando testes com Jest

[Jest é a maneira mais divertida de escrever testes](https://medium.com/@oieduardorabelo/jest-escrever-testes-nunca-foi-t%C3%A3o-divertido-5f0e1950ba10). E em Node.js com TypeScript, não é diferente! Vamos instalar as depêndencias:

$ npm i @types/jest jest ts-jest

Agora precisamos dizer ao Jest para utilizar o compilador do TypeScript e também procurar por arquivos **.ts**. Vamos adicionar essa configuração extra do Jest no package.json :

// node-typescript/package.json{  
 // ...outras regras do seu `package.json`  
 "jest": {  
 "transform": {   
 "^.+\\.ts$": "ts-jest" // [A]  
 },  
 "testRegex": "\\.test\\.ts", // [B]  
 "moduleFileExtensions": [   
 "ts",  
 "tsx",  
 "js",  
 "jsx",  
 "json",  
 "node" // [C]  
 ]  
 }  
}

* **[A]:**estamos dizendo ao Jest que, ao encontrar arquivos **.ts** utilize o **ts-jest** como parte da transformação/resolução do módulo
* **[B]:**minha preferência pessoal são arquivos <**nome>.test.js**, sempre co-locados com o arquivo testado, você pode mudar essa RegEx
* **[C]:**precisamos declarar todas essas extensões (podemos omitir **.tsx/jsx**nesse exemplo) pois arquivos de módulos (dentro de **node\_modules/**) precisam ser importados pelo Jest e ou qualquer outra dependência

Para testar nossa aplicação **express**, iremos utilizar a biblioteca **supertest**:

$ npm i @types/supertest supertest

Dentro do nosso package.json , vamos adicionar um **script**chamado**test**:

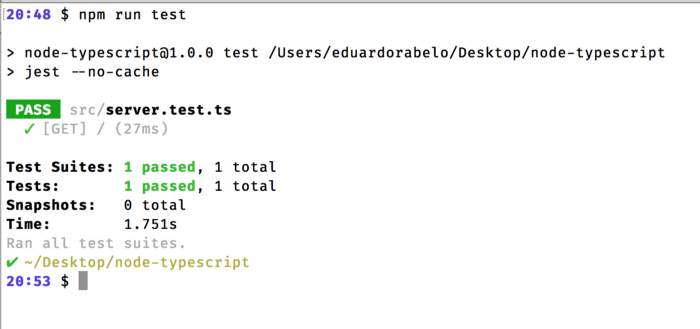
// node-typescript/package.json{  
 // ...outras regras do seu `package.json`  
 "scripts": {  
 // ...outros scripts  
 "test": "jest --no-cache"  
 }  
}

Agora já temos todas as dependências e o comando correto, vamos criar nosso arquivo **src/server.test.ts**e escrever nosso primeiro teste:

// node-typescript/src/server.test.tsimport req from "supertest";  
import server from "./server";  
test("[GET] /", async () => {  
 const res = await req(server).get("/");  
 expect(res.text).toBe("Hello ts-node!");  
});

Ao executar npm run test temos como resultado:

https://miro.medium.com/max/60/1*5XzXXu1LCPeuNU8b3R47Aw.png?q=20



Como você pode ver, ter dividido nossa aplicação **express** em **entry.ts** e **server.ts**,nos ajuda a importar toda a aplicação sem executá-la (essa abordagem é opcional). Também estamos utilizando sintaxe ES6 em nossos testes ;)

# Isso funciona em produção?

Olhando de perto nossa configuração, só estamos utilizando **ts-node**para iniciar nosso servidor. Por padrão, ele executa o compilador do TypeScript, carrega todos os tipos, faz todas análises estáticas etc.

Como comentado no repositório do **ts-node**, ele [não é recomendado](https://github.com/TypeStrong/ts-node/issues/104) para produção. Também não há **nenhum** benchmark dizendo que há problemas de performance ao usar **ts-node**, o ponto principal é [o consumo de memória](https://github.com/TypeStrong/ts-node/issues/104#issuecomment-337144823), devido ao uso conjunto com o compilador do TypeScript.

Tendo isso em mente, precisamos criar mais uma etapa, o famigerado **build**!

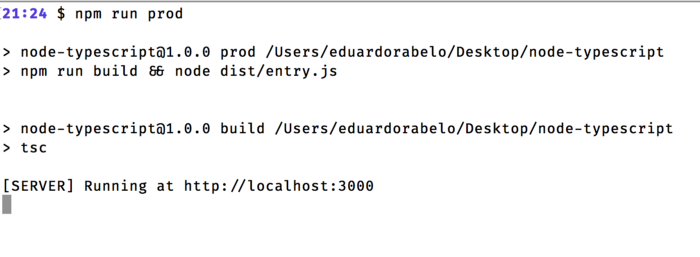
Porém, se você olhar nossa configuração do TypeScript, lá no começo, o **tsconfig.json**, nós temos a seguinte chave: "outDir": "./dist". Em outras palavras, já está tudo configurado! 🤓

Vamos adicionar mais 2 comandos ao nosso package.json , um para executar o compilador do TypeScript e imprimir arquivos **.js**e outro para executar o nosso servidor utilizando esse resultado.

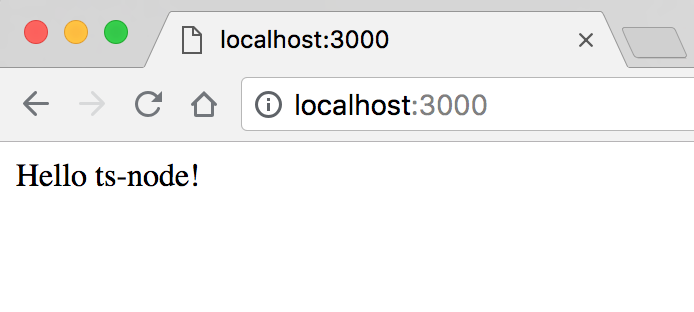
// node-typescript/package.json{  
 // ...outras regras do seu `package.json`  
 "scripts": {  
 // ...outros scripts  
 "build": "tsc",  
 "prod": "npm run build && node dist/entry.js"  
 }  
}

E ao executarmos npm run prod , teremos:

https://miro.medium.com/max/60/1*RUvOMUNosA9POGZfLb9gCw.png?q=20



https://miro.medium.com/max/60/1*BWccJNAh3gw55f3U2D_oRQ.png?q=20



Prometo que não é a mesma imagem :P

Hooray! 🎉🎉

Agora estamos compilando para puro JavaScript e utilizando Node.js para rodar o servidor.

A partir daqui você pode adicionar todas as suas práticas, módulos e pacotes preferidos. Junto da segurança de ter seu código back-end com análise estáticas e todos os benefícios do ecossistema do TypeScript.

Você que chegou até aqui, meu muito obrigado! 🤗