1 DE NOVEMBRO DE 2022 / #TYPESCRIPT

Como configurar um projeto em TypeScript

Tradutor: Allynaell

Autor: freeCodeCamp.org (em inglês)

Artigo original: How to set up a TypeScript project

Escrito por: David Piepgrass

Um guia completo para iniciantes que criam aplicações para a web com React

Em todos os meus anos como desenvolvedor, nunca encontrei uma selva tão avassaladora quanto o mundo do JavaScript. É um mundo de complexidade desconcertante, onde fazer um simples projeto parece exigir a instalação de inúmeras ferramentas, a edição de vários arquivos de texto que conectam todas essas ferramentas e a execução de muitos comandos de terminal.

Há algumas ferramentas que tentam esconder a complexidade de você, com vários níveis de sucesso. Porém, enquanto essas ferramentas não tiverem adoção universal, elas me parecem ainda mais coisas que você precisa aprender além de tudo.

Para mim, a maior fonte de irritação é que a maioria dos tutoriais assume que você **já está** familiarizado com o ecossistema. Então, não há uma preocupação com explicar o básico. Para piorar, muitos tutoriais tentam empurrar diversas ferramentas extras para você — como Webpack, Bower, NVM e Redux — com pouca explicação.

É irônico, porque o próprio JavaScript já está instalado em praticamente todos os computadores do mundo, incluindo celulares. Por que escrever um aplicativo da maneira "profissional" tem que ser tão complexo em comparação com escrever um arquivo HTML com algum código JavaScript nele?

Se, como eu, você tem a **necessidade inata** de entender o que está acontecendo e:

- se você não suporta copiar comandos cegamente em terminais e arquivos de texto
- se você quiser ter certeza de que precisa de uma ferramenta antes de instalá-la
- se você está se perguntando por que seu projeto baseado em npm tem 50MB antes de escrever sua primeira linha de código

então, boas-vindas! Você veio ao lugar certo.

Por outro lado, se você quiser começar a programar em 5 minutos, eu conheço um truque para isso: pule a introdução aqui e comece a ler sobre a Abordagem A na seção 2. Se você acha que estou dando muita informação, apenas pule as partes que você não quer aprender à medida que avança.

Neste tutorial, assumirei que você tem **alguma** experiência de programação com HTML, CSS e JavaScript, mas **nenhuma** experiência com TypeScript, React ou Node.js.

Darei aqui uma visão geral do ecossistema JavaScript como eu o entendo. Vou explicar por que acho que TypeScript e React (ou Preact) são sua melhor aposta para criar aplicativos da web. Também vou ajudar você a iniciar um projeto sem extras desnecessários.

Na seção 2, discutiremos como e por que adicionar extras ao seu projeto, **se** você decidir que deseja.

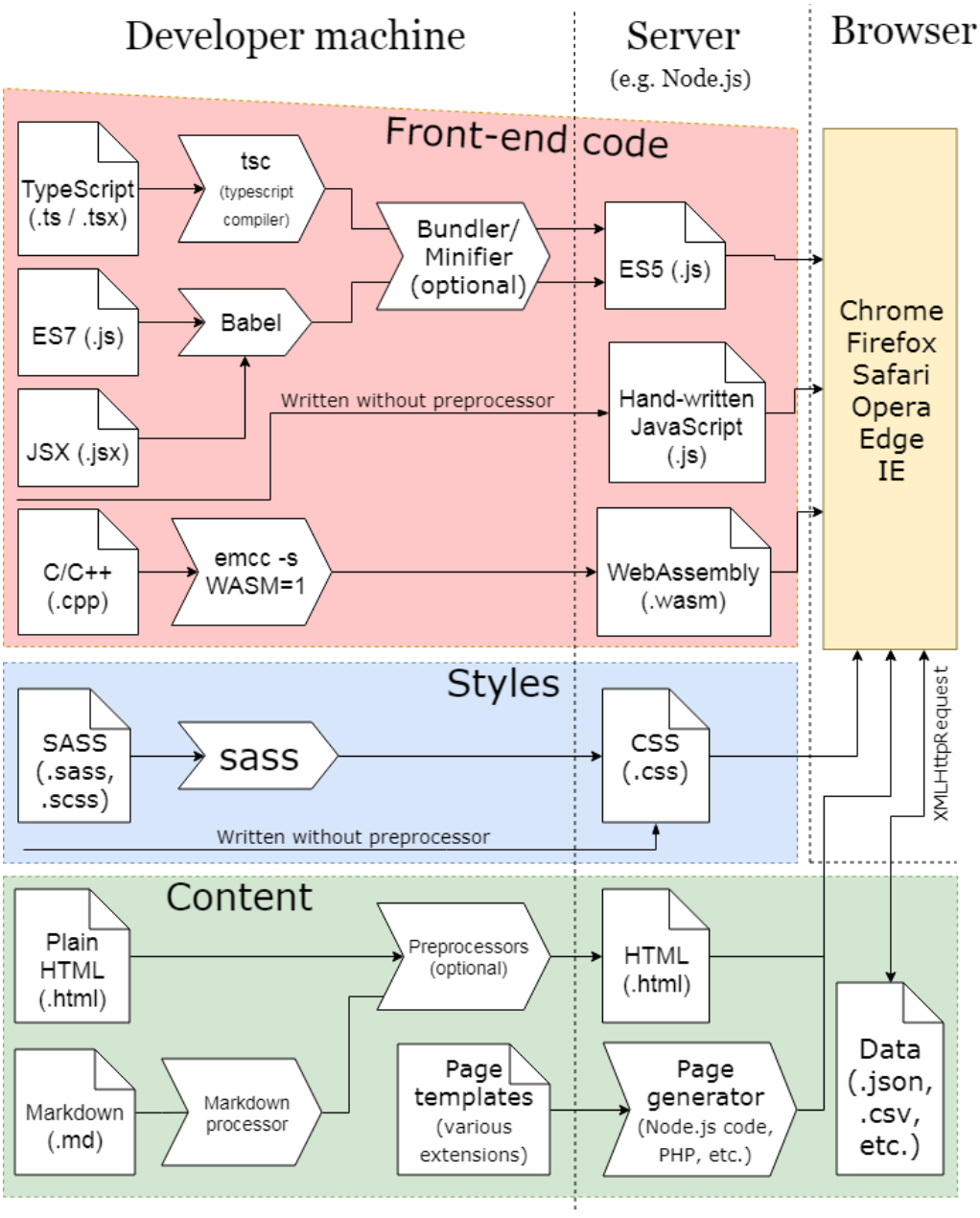
Índice

Seção 1: visão geral do ecossistema JavaScript

Para muitas linguagens de programação, há uma certa maneira de fazer coisas que todo mundo conhece.

Por exemplo, se você quiser criar um aplicativo C#, instale o Visual Studio, crie um projeto do Windows Forms com alguns cliques do mouse, clique no botão verde "reproduzir" para executar seu novo programa e comece a escrever código para ele. O gerenciador de pacotes (NuGet) é interno e o depurador simplesmente funciona. Claro, pode levar algumas horas para instalar o IDE e o WPF é tão divertido quanto bater a cabeça contra uma parede de tijolos, mas, pelo menos, é fácil **começar** (exceto se você não estiver usando o Windows, então é totalmente diferente, mas isso já é divagação).

Em JavaScript, por outro lado, existem muitas bibliotecas e ferramentas concorrentes para quase todos os aspectos do processo de desenvolvimento. Essa enxurrada de ferramentas pode se tornar assustadora antes de você escrever sua primeira linha de código! Quando você busca no Google "como criar uma aplicação para a web", cada site que você visita parece dar conselhos diferentes.



Obrigado, draw.io, pela ferramenta de diagramação!

A única coisa com a qual a maioria das pessoas parece concordar é usar o Node Package Manager (NPM) para baixar as bibliotecas do JavaScript (tanto do lado do servidor quanto do lado do navegador). Mesmo

aqui, porém, algumas pessoas usam o Yarn, que é compatível com o npm, ou, possivelmente, o Bower.

O NPM é fornecido com o Node.js, um servidor da web que você controla inteiramente com código em JavaScript. O NPM é totalmente integrado ao Node. Por exemplo, o comando `npm start` executa `node server.js` por padrão.

Mesmo se você estivesse planejando usar um servidor da web diferente (ou não usar **nenhum** servidor web e apenas clicar duas vezes em um arquivo HTML), todo mundo parece supor que você terá o Node.js instalado. Então, você também pode ir em frente e instalar o Node.js, que dá a você o `npm` como um efeito colateral.

O Node.js não é apenas um servidor da web — ele também pode executar aplicativos de linha de comando escritos em JavaScript. Nesse sentido, o compilador do TypeScript é um aplicativo Node.js!

Além do NPM, você tem várias opções:

Qual "sabor" de JavaScript você quer?

O nome oficial do JavaScript é, na verdade, ECMAScript, e a versão mais amplamente implementada é a ECMAScript 6 ou a ES6. Navegadores antigos, principalmente o Internet Explorer, suportam apenas a ES5.

A ES6 acrescenta muitos novos recursos úteis e importantes como módulos, let, const, arrow functions (ou funções lambda), classes e atribuição de desestruturação.

A ES7 adiciona mais alguns recursos, principalmente algo chamado async/await.

Se você não precisa usar navegadores antigos e se o seu código não é muito grande, executá-lo diretamente no navegador é uma opção atraente, pois você não precisa "compilar" seu JavaScript antes de abri-lo no navegador.

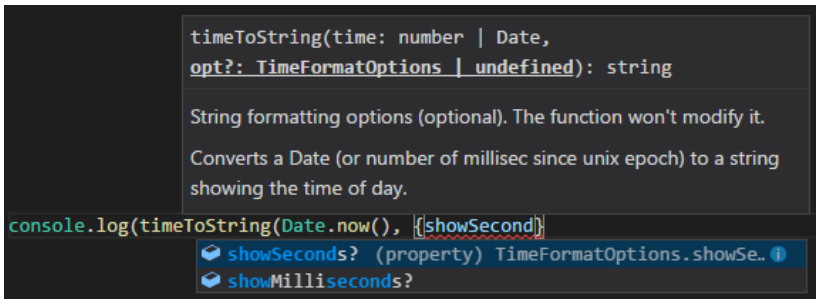
Há, **porém**, muitas razões para se usar uma etapa de compilação:

- Se você precisar dar suporte a navegadores antigos, precisará de um "transpilador" para poder usar novos recursos de JavaScript em navegadores antigos. Um transpilador é um compilador cujo código de saída é uma linguagem de alto nível – neste caso, JavaScript. Eu diria que o transpilador mais popular é o Babel, com o TypeScript em segundo lugar.
- Se você quiser usar o framework popular React (mas sem o TypeScript), provavelmente escreverá código em "JSX" — fragmentos de XML dentro de código JavaScript. O JSX não é suportado por navegadores e, portanto, requer um pré-processador (normalmente o Babel).
- Se você quiser "minificar" seu código para que ele use menos largura de banda (ou esteja ofuscado), você precisará de um pré-processador "minificador". Minificadores populares incluem o UglifyJS, o JSMin e o o Closure Compiler.
- Se você quiser verificação de tipo ou autopreenchimento de código de alta qualidade (também conhecido como IntelliSense), você vai querer usar o TypeScript, um superset do JavaScript (o que significa que todo arquivo JavaScript também é um arquivo TypeScript… ostensivamente). O TypeScript suporta recursos do ES7 e do JSX. Sua saída é o código da ES5 ou da ES6. Quando o código TypeScript e JSX são usados juntos, a extensão do arquivo deve ser `.tsx`. Algumas pessoas estão usando uma linguagem diferente, semelhante em conceito ao TypeScript, chamada Flow.
- Se você não gosta de JavaScript, pode tentar uma linguagem totalmente diferente que transpile para JavaScript, como Elm, ClojureScript ou Dart.

Felizmente, é possível automatizar a compilação para que seu código seja recompilado sempre que você salvar um arquivo.

Este tutorial usa TypeScript, um superset de JavaScript com um sistema de tipos abrangente. Os benefícios do TypeScript são:

1. Você recebe mensagens de erro do compilador quando comete erros relacionados ao tipo (ao invés de descobrir erros indiretamente quando seu programa se comporta mal). Em IDEs, como o Visual Studio Code, seus erros são sublinhados em vermelho.
2. Você pode obter recursos de refatoração. Por exemplo, no Visual Studio Code, pressione `F2` para renomear uma função ou variável através de múltiplos arquivos, sem afetar outras coisas que tenham o mesmo nome.
3. Os tipos permitem que os IDEs forneçam pop-ups de conclusão de código, também conhecidos como IntelliSense, o que torna a programação muito mais fácil, pois você não precisa memorizar todos os nomes e argumentos esperados das funções que você chama:



IntelliSense™ no Visual Studio Code

Dica: para testar o TypeScript sem instalar nada, visite o playground.

Client x servidor

Você pode executar o código em um client (navegador em front-end), em um servidor (Node.js no back-end) ou em ambos. O client não está sob seu controle. O usuário pode usar Firefox, Chrome, Safari, Opera, Edge ou, no pior caso, o Internet Explorer.

Por questões de segurança, mantenha em mente que o usuário pode modificar o comportamento do navegador usando extensões ou as ferramentas do desenvolvedor (`F12`). Você nem pode ter certeza de que o seu código está sendo executado em um navegador real.

Os desenvolvedores costumavam confiar na biblioteca jQuery para obter um comportamento consistente em diferentes navegadores, mas, hoje em dia, você pode contar com o fato de os navegadores diferentes se comportarem da mesma maneira na **maioria** dos casos (exceto, talvez, o Internet Explorer).

Neste tutorial, executaremos todos os códigos importantes no navegador, mas também configuraremos um servidor Node.js simples para fornecer o aplicativo ao navegador. Muitos outros servidores estão disponíveis, como Apache, IIS ou um servidor estático, como o Jekyll.

O Node.js, porém, se tornou uma espécie de padrão, provavelmente porque o Node.js e o NPM estão empacotados juntos.

Frameworks de interface do usuário

O HTML e o CSS, por si só, são ótimos para os velhos artigos com imagens ou formulários simples. Se isso é tudo o que você está fazendo, provavelmente não há necessidade de usar o JavaScript. O CSS pode até fazer algumas coisas que antes exigiam JavaScript, como menus suspensos, páginas que se reformatam completamente para navegadores pequenos/de dispositivos móveis ou impressão e animações.

Se você precisar de algo mais complexo do que isso ou se suas páginas forem geradas dinamicamente a partir de dados brutos, você provavelmente desejará usar JavaScript com uma biblioteca ou framework de interface de usuário opcional. Mostrarei mais tarde (texto em inglês) como usar o React, que conquistou uma posição como o framework de interface do usuário mais popular, e seu primo, o Preact.

As alternativas populares "grandes" (texto em inglês) incluem Angular e Vue.js, enquanto as "pequenas" incluem D3, Mithril e um antigo clássico chamado jQuery.

Se o seu servidor da web executa JavaScript (Node.js), você pode executar o React no servidor para pré-gerar a aparência inicial da página.

Ferramentas de construção

Várias ferramentas para "construir" e "empacotar" seu código (texto em inglês) estão disponíveis — Webpack, Grunt, Browserify, Gulp, Parcel — mas todas essas coisas são opcionais. Eu mostrarei a você como fazer isso apenas com o `npm` e, se você quiser, Parcel ou Webpack.

"Sabores" do CSS

Neste artigo, usaremos CSS simples. Se você vai ter uma etapa de compilação de qualquer maneira, pode querer experimentar o SCSS, um derivado "melhorado" do CSS com recursos extras. Você também pode usar o SASS, que é, conceitualmente, idêntico ao SCSS, mas tem uma sintaxe mais concisa.

De qualquer forma, você precisará do pré-processador SASS. E, como sempre no mundo JavaScript, existem várias alternativas (texto em inglês), especialmente o LESS.

Teste unitário

As bibliotecas de teste unitário populares são Mocha, Jasmine e Jest. Saiba mais aqui (texto em inglês). O NPM tem um comando especial para teste, `npm test` (que é a abreviação para `npm run test`).

Outras bibliotecas

Além do Redux, outras bibliotecas populares de JavaScript (texto em inglês) incluem Lodash, Ramda, Underscore e GraphQL.

O utilitário de linting mais popular é 0 ESLint.

Bootstrap é uma biblioteca CSS popular mas requer uma parte de JavaScript (e é realmente SASS, não CSS).

Quando você vê `$` no código JavaScript, normalmente se refere ao jQuery. Quando você vê `_`, normalmente se refere a Lodash ou Underscore.

Talvez valha a pena mencionar bibliotecas populares de templates: Jade, Pug, Mustache e Handlebars.

Aplicativos que não sejam da web

Não direi mais nada sobre isso, mas TypeScript e JavaScript podem ser usados fora da web.

Com Electron, você pode escrever aplicativos de desktop multiplataforma. Com React Native, você pode escrever aplicativos JavaScript para dispositivos Android/iOS que tenham uma interface de usuário "nativa". Você também pode escrever aplicativos de linha de comando com Node.js.

Tipos de módulo

Por muito tempo, todo o código JavaScript foi executado em um único namespace global. Isso causou conflitos entre bibliotecas de código não relacionadas, então vários tipos de "definições de módulo" foram inventados para **simular** o que outras linguagens chamam de pacotes ou módulos.

O Node.js usa módulos CommonJS, que envolvem uma função mágica chamada `require('module-name')` para importar módulos e uma variável mágica chamada `module.exports` para criar módulos. Para escrever módulos que funcionem tanto em navegadores quanto em Node.js, pode-se usar o Universal Module Definition (módulos UMD). Módulos que podem ser carregados de forma assíncrona usam AMD.

O ES6 introduziu um sistema de módulo envolvendo as palavras-chave `import` e `export`, mas o Node.js e alguns navegadores ainda não o suportam. Aqui está uma cartilha sobre os vários tipos de módulos (texto em inglês).

Polyfills e protótipos

Como um desenvolvedor experiente, consigo pensar em apenas duas palavras (além dos nomes de bibliotecas e ferramentas) que são usadas apenas na terra do JavaScript: **polyfill** e **prototype**.

Polyfills são auxiliares de compatibilidade com versões anteriores. Eles são pedaços do código escritos em JavaScript que permitem usar novos recursos em navegadores antigos. Por exemplo, a expressão `"food".startsWith('F')` testa se a string `'food'` começa com F (para constar, isso é `false` - ela começa com `f`, não `F`.) `startsWith`, porém, é um recurso novo do JavaScript que não está disponível em navegadores mais antigos.

Você pode "polipreenchê-lo" com este código:

```
String.prototype.startsWith = String.prototype.startsWith ||  function(search, pos) {    return search
```

Ele tem a forma `X = X || function(...) {...}`, que significa "se X estiver definido, defina X para si mesmo (não o altere), caso contrário, defina X para ser esta função". A função mostrada aqui se comporta da maneira que `startsWith` deveria.

O código refere-se a uma das outras coisas únicas sobre JavaScript, a ideia de protótipos. Os protótipos correspondem **aproximadamente** a classes em outras linguagens. Portanto, o que esse código está fazendo é, na verdade, alterar a definição dos tipos de dados `String`. Depois, quando você escrever `'string'.startsWith()`, ele chamará esse polyfill (se `String.prototype.startsWith` ainda não estiver definido). Há vários artigos por aí para ensinar sobre protótipos e herança prototípica, como este (texto em inglês).

Até mesmo alguns recursos avançados do navegador têm polyfills. Você já ouviu falar do WebAssembly, que permite executar código C e C++ no navegador? Existe um polyfill JavaScript para isso!

Crédito

Gostaria de agradecer às pesquisas State of Javascript e State of JavaScript Frameworks (textos em inglês) por muitas das informações acima! Para alguns itens, usei npm-stat para medir popularidade. Veja também esta outra nova pesquisa (texto em inglês).

Seção 2: configurando o projeto de verdade

Ei! Ainda acordado? Agora, faremos uma tour pelo ecossistema de ferramentas JavaScript. Esta parte não é sobre o React (falaremos disso mais tarde - novamente, texto em inglês), mas inclui um componente do React simples.

Esta é uma tour **grande**. Então, falaremos sobre como escrever seu aplicativo de três maneiras diferentes (com um resumo depois):

- A. a forma mais fácil (com Parcel)
- B. a forma com menos ferramentas (ou o modo faça-você-mesmo)
- C. a forma do Webpack

As primeiras seis etapas são as mesmas em todas as três abordagens. Então, vamos começar!

Etapa 1: instalar o Node.js/npm

Se você ainda não o fez, instale o Node.js, que também instalará o gerenciador de pacotes de linha de comando, `npm`.

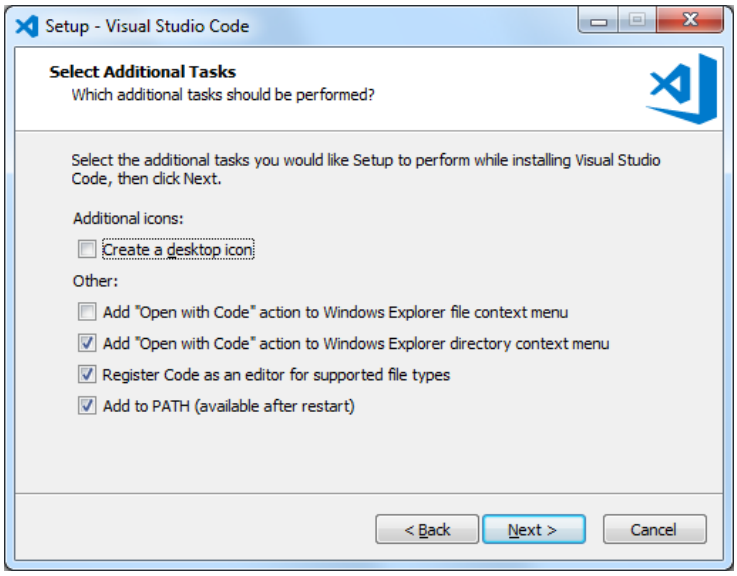
Se você deseja implantar seu aplicativo em algum outro servidor da web, recomendo se preocupar em como fazer isso mais tarde.

Etapa 2: instalar o Visual Studio Code ou outro editor

Uma das principais razões para usar o TypeScript ao invés do JavaScript é que ele suporta recursos de conclusão de código.

Para aproveitar esse benefício, você precisará editar seus arquivos TypeScript `.ts` em um editor compatível, como o Visual Studio Code — que é gratuito e multiplataforma. É também o editor de texto mais popular para aplicativos JavaScript. As alternativas incluem o Atom e o Sublime Text.

O Visual Studio Code (VS Code) é orientado a pastas: você abre uma pasta no VS Code e essa pasta será tratada como o "projeto" atual. Durante a instalação (no Windows, pelo menos) ele oferecerá uma caixa de seleção para adicionar uma ação "Abrir com o Code" para pastas (diretórios). Eu recomendo usar essa opção como uma maneira fácil de iniciar o VS Code de qualquer pasta:



Crie uma pasta vazia para seu aplicativo e abra essa pasta no VS Code. Observe que o VS Code possui um terminal integrado. Portanto, você não precisará de uma janela de terminal separada.

Etapa 3: configurar o package.json

O arquivo `package.json` representará a configuração do seu projeto. Isso inclui seu nome, comandos de compilação, a lista de módulos do npm usados pelo seu projeto e muito mais.

Se você ainda não fez isso, crie uma pasta vazia para seu aplicativo e abra uma janela de terminal nessa pasta.

No terminal, execute `npm init`.

O `npm init` fará algumas perguntas para produzir o `package.json`. Deixe um campo em branco para aceitar a sugestão padrão.

Eu queria fazer um pequeno aplicativo educacional para desenhar alguns gráficos demonstrando como a ciência climática explica o registro de temperatura do século 20.

Então, eu chamei meu aplicativo de `climate-app`:

```
C:\Dev\climate-app>npm initThis utility will walk you through creating a package.json file.It only co
```

```
package name: (climate-app)version: (1.0.0)description: Demo to visualize climate dataentry point: (i
```

```
About to write to C:\Dev\climate-app\package.json:{  "name": "climate-app",  "version": "1.0.0",  "de
```

```
Is this ok? (yes)
```

Observe a referência a `index.js`. Estranhamente, esse arquivo não precisa existir e não o usaremos. Ele é usado apenas se você compartilhar seu código via npm.

Etapa 4: instalar o Typescript

O VS Code supostamente (texto em inglês) tem "suporte à linguagem" TypeScript em vez de um **compilador** TypeScript. Então, agora, precisamos instalar o compilador.

Existem duas maneiras de instalar o TypeScript com npm. Use

```
npm install --global typescript
```

ou

```
npm install --save-dev typescript
```

Se você usar a opção `--global`, o compilador de TypeScript `tsc` estará disponível em todos os projetos na mesma máquina. Ele também estará disponível como um comando de terminal, mas não será adicionado ao seu arquivo `package.json`. Portanto, se você compartilhar seu código com outras pessoas, o TypeScript **não** será instalado quando outra pessoa obtiver seu código e executar `npm install`.

Se você usar `--save-dev`, o TypeScript será adicionado ao `package.json` e instalado na pasta `node_modules` do seu projeto (tamanho atual: 34,2 MB), mas **não** estará disponível diretamente como um comando de terminal.

Você ainda pode executá-lo a partir do terminal como `./node_modules/.bin/tsc`, e ainda pode usar `tsc` dentro da seção `npm` `"scripts"` do `package.json`.

Curiosidade: o compilador TypeScript é multiplatforma porque é escrito em TypeScript — e compilado para JavaScript.

Etapa 5: instalar React ou Preact

Para adicionar o React ao seu projeto:

```
npm install react react-domnpm install --save-dev @types/react @types/react-dom
```

Observação: `--save-dev` marca as coisas como "usadas para desenvolvimento", enquanto `--save` (que é o padrão e, portanto, opcional) significa "usado pelo programa quando é implantado".

Pacotes `@types` fornecem informações de tipo para o TypeScript, mas não são usados quando seu código está em execução/implantado.

Se você esquecer `--save-dev` ou se você o utilizar no pacote errado, **seu projeto ainda funcionará**. A distinção só é importante se você compartilhar seu projeto como um pacote do npm.

Como alternativa, você pode usar o Preact, que é quase o mesmo que o React (texto em inglês), mas mais de 10 vezes menor. O Preact possui definições de tipo TypeScript incorporadas. Portanto, você só precisa de um único comando para instalá-lo:

```
npm install preact
```

Dica: `npm i` é um atalho para `npm install`, e `-D` é uma abreviação para `--save-dev`.

Observação: não instale `preact` e `@types/react` no mesmo projeto, ou o `tsc` vai enlouquecer e dar cerca de 150 erros (veja a preact issue #639 - texto em inglês). Se isso acontecer, desinstale os tipos do React com `npm uninstall @types/react @types/react-dom`

Etapa 6: escrever código em React

Faça um arquivo chamado `app.tsx` com este pequeno programa em React:

Observação: para que o JSX incorporado (HTML/XML) funcione, a extensão do arquivo deve ser `tsx`, não `ts`.

Se você tiver algum problema para fazer seu código funcionar, tente este código — é o programa React mais simples possível:

```
import * as ReactDOM from 'react-dom';import * as React from 'react';
```

```
ReactDOM.render(React.createElement("h2", null, "Hello, world!"),                document.body);
```

Discutiremos como o código funciona mais tarde. Por enquanto, vamos nos concentrar em fazê-lo funcionar.

Se você estiver usando o Preact, altere as duas primeiras linhas assim:

```
import * as React from 'preact';import * as ReactDOM from 'preact';
```

Algumas observações sobre o Preact:

- Existe uma biblioteca preact-compat que permite que você use o preact com zero alterações no seu código React. Existem instruções de uso para usuários do Webpack/Browserify/Babel/Brunch, e essa página mostra como usar preact-compat com Parcel.

- Há rumores de que, no Preact, você deve escrever `/** @jsx h */` no topo do arquivo, que diz ao TypeScript para chamar `h()` ao invés do `React.createElement` padrão. Nesse caso, você **não deve** fazer isso ou receberá um erro no seu navegador, dizendo que `h` não está definido (`React.h`, muito embora esteja). Na verdade, Preact define `createElement` como um alias para `h`. Como nossa declaração `import` atribui `'preact'` para `React`, `React.createElement` existe e funciona muito bem.

Opcional: executando scripts em TypeScript

Este tutorial está focado em criar **páginas da Web** que executam código em TypeScript. Se você deseja executar um arquivo TypeScript diretamente do prompt de comando, a maneira mais fácil é usar o `ts-node`:

```
npm install --global ts-node
```

Depois de instalar o `ts-node`, execute `ts-node X.ts`, onde `X.ts` é o nome de um script que você deseja executar. No script, você pode chamar `console.log("Hello")` para escrever texto no terminal (ler texto de um usuário é mais complicado - texto em inglês). Em sistemas Linux, você pode colocar um "shebang" no topo do script se quiser executar `./X.ts` diretamente (sem mencionar o `ts-node`):

```
#!/usr/bin/env ts-node
```

Observação: se você não precisar executar arquivos `.ts` de um terminal, não precisará instalar o `ts-node`.

Executando seu projeto – Abordagem A: a maneira fácil

Descobri o Parcel quando terminei de escrever este artigo. Honestamente, se eu soubesse do Parcel desde o início, talvez não tivesse me incomodado em escrever sobre as outras abordagens. É impressionante a facilidade de usar o Parcel! Merece uma medalha!

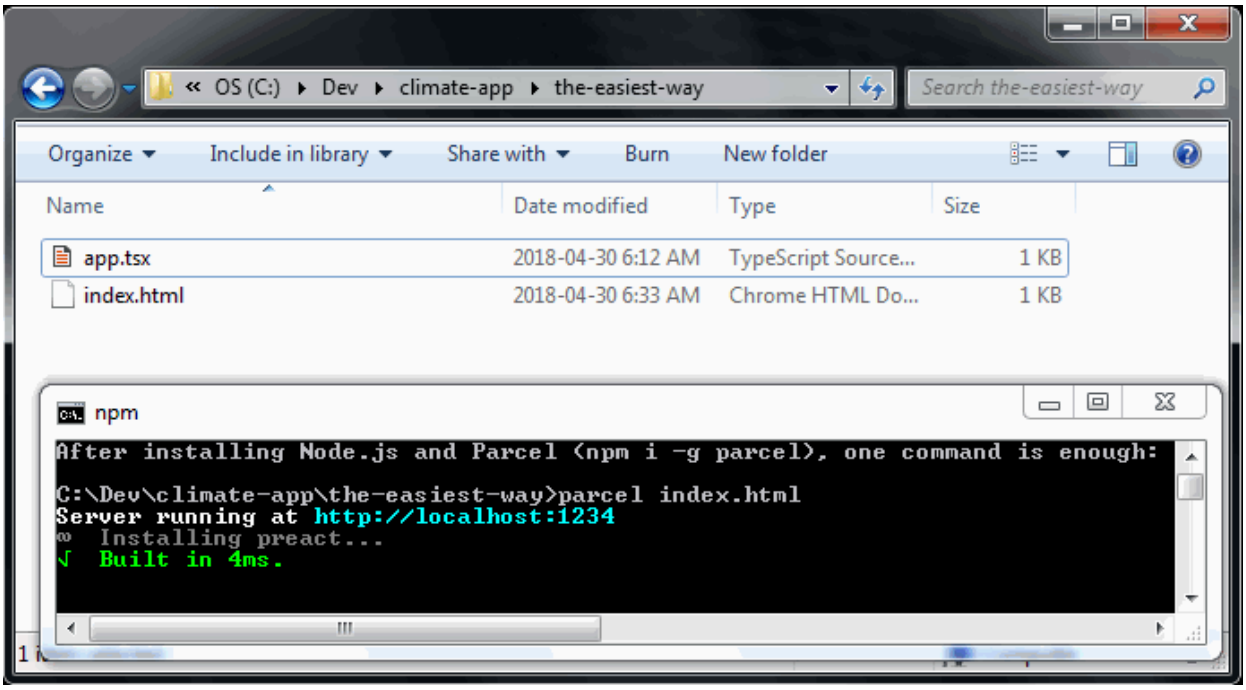
No entanto, ele é muito grande (81,9 MB). Então, você deve instalá-lo como global:

```
npm install --global parcel-bundler
```

A verdade é que eu tenho mentido para você. Parcel é **tão** fácil que você nem precisa de todos os seis passos acima! Você realmente só precisa das etapas 1, 2 e 6 (instalar o Node, instalar um editor e escrever código) porque o Parcel fará as etapas 3, 4 e 5 para você automaticamente.

Então, tudo o que temos que fazer agora é criar um arquivo `index.html` que se refira ao nosso arquivo `app.tsx`, como esse:

Em seguida, simplesmente abra um terminal na mesma pasta e execute o comando `parcel index.html`.



Isso não pode ser executado diretamente em um navegador, é claro. Então, o Parcel:

1. Compila o `app.tsx` automaticamente
2. Instala o React ou Preact se ainda não estiver instalado, pois percebe que você está usando
3. Agrupa seu aplicativo com suas dependências em um único arquivo chamado `app.dd451710.js` (ou algum outro nome parecido e engraçado)
4. Cria um `index.html` modificado que se refere ao aplicativo compilado e empacotado
5. Coloca esses novos arquivos em uma pasta chamada `dist`.

Depois, ele faz todo o resto para você:

1. Ele executa seu aplicativo em um minisservidor da web em `http://127.0.0.1:1234` — visível em um navegador na mesma máquina
2. Ele observa as alterações no seu código (`app.tsx` e `index.html`) e recompila quando você as altera
3. Como se isso não bastasse, quando seus arquivos forem alterados, ele enviará um comando ao seu navegador para **atualizar automaticamente!**
4. Melhor ainda, ele atualiza a página sem recarregá-la totalmente usando seu recurso Hot Module Replacement

Pode ser um desafio configurar uma construção convencional que faça todas essas coisas. Este tutorial cobre apenas como fazer o nº 1 e o nº 2 em uma compilação convencional, apenas com recompilação de código (não em HTML).

Para saber mais sobre os recursos do Parcel, consulte a documentação do Parcel.

Uma limitação do Parcel é que ele não realiza verificação de tipo (seu código é traduzido para JavaScript, mas erros de tipo não são detectados).

Para projetos pequenos, isso não é um grande problema, pois o Visual Studio Code executa sua própria verificação de tipo. Ele fornece sublinhados ondulados vermelhos para indicar erros e todos os erros são

listados no painel "Problemas" (pressione `Ctrl` + `Shift` + `M` para mostrá-lo). Se você quiser, porém, pode executar `npm install parcel-plugin-typescript` para obter suporte aprimorado ao TypeScript, incluindo checagem de tipo (não está funcionando atualmente para mim).

Outras abordagens

As outras abordagens são mais conhecidas e são práticas padrão na comunidade JavaScript. Estaremos criando uma pasta com os seguintes arquivos dentro dela:

- `app/index.html`
- `app/app.tsx`
- `package.json`
- `tsconfig.json`
- `server.js`
- `webpack.config.js` (opcional)

Por uma questão de comunicação com outras pessoas que examinam seu código posteriormente, é útil separar o **código do front-end** do seu programa de sua **configuração de compilação** e **servidor de aplicativos**.

A pasta raiz de um projeto tende a ficar cheia de arquivos adicionais com o passar do tempo (assim como `.gitignore` se você usa git, arquivos `README` e `LICENSE`, arquivos `appveyor` / `travis` se você usa integração contínua.) Portanto, devemos separar o código do nosso front-end em uma pasta diferente.

Em adição aos arquivos que **nós** criamos, o TypeScript compilará `app.tsx` em `app.js` e `app.js.map`, enquanto o `npm` criará uma pasta chamada `node_modules` e um arquivo chamado `package-lock.json`. Não consigo imaginar a razão de se chama "lock", mas esta página explica porque isso existe (em inglês).

Então, comece criando uma pasta `app` e colocando seu `app.tsx` lá.

Executando seu projeto – Abordagem B: o caminho de poucas ferramentas

Parece que o projeto JavaScript de todo mundo usa uma dúzia de ferramentas além da pia da cozinha. É possível fazer um pequeno programa sem ferramentas extras? Certamente é! Aqui está como.

Etapa B1: criar o tsconfig.json

Crie um arquivo de texto chamado `tsconfig.json` na sua pasta raiz.

Este arquivo marca a pasta como um projeto TypeScript e habilita comandos de compilação no VSCode com `Ctrl` + `Shift` + `B` (o comando `tsc: watch` é útil — ele recompilará automaticamente seu código sempre que você o salvar).

Detalhe: o `tsc` permite comentários em arquivos `.json`, mas o `npm` não.

Este arquivo é muito importante, pois se as configurações não estiverem corretas, algo pode dar errado e erros misteriosos ocorrer. Aqui está a documentação do tsconfig.json, mas as opções do compilador são documentadas separadamente.

Etapa B2: adicionar um script de compilação

Para permitir o `npm` compilar seu código TypeScript, você também deve adicionar entradas na parte `scripts` do `package.json`. Modifique essa seção para que fique assim:

```
"scripts": {  "test": "echo \"Error: no tests installed\" && exit 1",  "build": "tsc",  "start": "node
```

O `build` script simplesmente executa `tsc`, que compila seu código de acordo com as opções em `tsconfig.json.` Para invocar este script, você escreve `npm run build` na linha de comando.

"Espere aí!", você deve estar pensando. "É muito mais fácil digitar `tsc` do que `npm run build`!" Isso é verdade, mas existem duas razões para definir um `build` script:

1. Se você instalou o TypeScript com `--save-dev` mas não `--global`, você não pode executar `tsc` diretamente da linha de comando porque ele não está no `PATH`.
2. Há uma boa chance de seu processo de compilação se tornar mais complicado posteriormente. Ao criar um script de compilação, você pode adicionar facilmente outros comandos ao processo de compilação posteriormente.

Observação: o `npm` executa o script `prestart` automaticamente sempre que alguém executa o script `start`. Então, você **pode** acrescentar um script adicional:

```
"prestart": "npm run build",
```

Isso compilaria seu projeto sempre que você iniciasse seu servidor com `npm start` ou `npm run start`.

Há, porém, duas desvantagens:

1. o `tsc` é um pouco lento
2. se o `tsc` encontrar erros de tipo, seu servidor não iniciará

Quando o TypeScript detecta erros de tipo, isso não o impede de gravar arquivos de saída JavaScript, e você pode achar isso útil de vez em quando para executar seu código mesmo com erros de tipo.

O comportamento padrão do `npm start` é executar `node server.js`. Então, parece redundante incluir `"start": "node server.js"`. No entanto, se o seu servidor estiver escrito em TypeScript, você precisará dessa linha, pois o arquivo `server.js` não existe até que o `server.ts` seja compilado. Se o `server.js` não existir, o `npm start` dará o erro `missing script: start`, a não ser que você inclua essa linha.

Etapa B3: fazer um servidor simples

Para garantir que o Node.js esteja funcionando, crie um arquivo de texto chamado `server.js` e coloque este código nele:

```
const http = require('http');
```

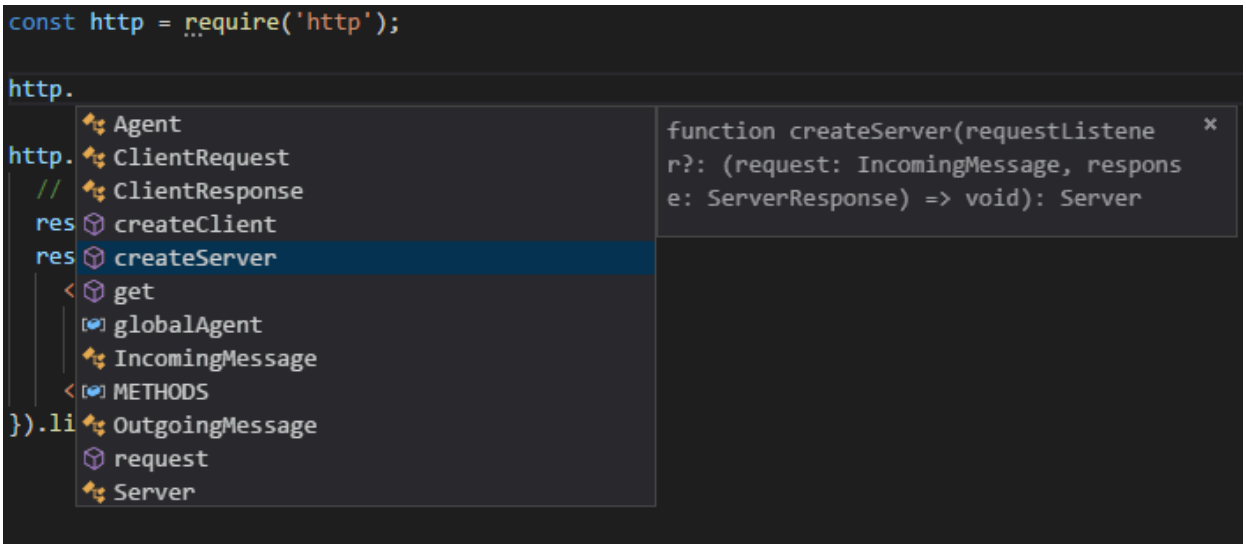
```
http.createServer(function (request, response) {  // Send HTTP headers and body with status 200 (meani
```

Execute `npm start` para iniciá-lo, visite `http://127.0.0.1:1234/index.html` para se certificar de que está funcionando e, então, pressione `Ctrl` + `C` para parar o servidor.

Para obter o IntelliSense para Node.js, você precisa instalar as informações de tipo para ele com este comando:

```
npm install @types/node --save-dev
```

Em seguida, no VS Code, digite `http.` para ter certeza de que funciona:



Nos bastidores, o VS Code usa o mecanismo do TypeScript para isso. No entanto, se você renomear seu arquivo para `server.ts`, o **IntelliSense não funcionará**! O TypeScript está quebrado no Node.js? Na verdade, não. O TypeScript ainda pode compilá-lo, apenas não compreenderá `require` em um contexto `.ts`. Portanto, em arquivos TypeScript, você deve usar `import` ao invés de `require`:

```
import * as http from 'http';
```

Observação: isso é confusamente diferente dos arquivos `.mjs` do Node, que requerem `import http from 'http';` (detalhes)

O TypeScript então converte `import` para `require` em sua saída (por causa da opção `"module": "umd"` em `tsconfig.json`).

Agora, vamos alterar nosso servidor para que ele possa servir qualquer arquivo da nossa pasta `/app`:

Você notará que esse código tem alguns... "aninhamentos" engraçados. Isso porque as funções do Node.js são normalmente assíncronas. Quando você chama funções em `fs`, ao invés de **retornar** um resultado, elas pausam seu programa até que elas terminem e, então, **chamam** uma função fornecida por você, enviando a essa função um erro (`err`) ou alguma informação (`fileInfo`).

Por exemplo, em vez de **retornar** informações sobre um arquivo, `fs.stat`, **envia** informações para uma função de callback.

Uma coisa suspeita sobre este servidor da web é que ele ignora `request.method`, tratando cada solicitação como se fosse um arquivo `GET`. No entanto, ele funciona bem o suficiente para começar.

Etapa B4 (opcional): usar o Express

Se você quiser que seu lado do servidor faça qualquer "roteamento" que seja mais complicado do que servir alguns arquivos, você provavelmente deve aprender sobre a estrutura do servidor do Node.js mais popular: o Express.

Se usarmos o Express, nosso código de servidor será muito mais curto.

Basta instalá-lo com `npm install express` e colocar o seguinte código no `server.js`:

```
const express = require('express');const app = express();
```

```
app.use('/node_modules', express.static('node_modules'));app.use('/', express.static('app'));app.liste
```

Etapa B5: criar uma página web para manter seu aplicativo

Finalmente, em sua pasta `app`, crie um arquivo `index.html` para carregar seu aplicativo:

Esta página inclui o React (`react.development.js` e `react-dom.development.js`) e o Preact (`preact.dev.js`), então não preciso dar instruções separadas para cada um. Você pode remover qualquer um que não esteja usando, mas a página ainda pode carregar com elementos de script não resolvidos.

Neste ponto, você deve ser capaz de construir seu código (`npm run build`), iniciar seu servidor (`npm start`) e visitar `http://127.0.0.1:1234` para visualizar seu aplicativo!

Lembre-se: você pode recompilar seu código automaticamente no VS Code. Pressione `Ctrl` + `Shift` + `B` e escolha `tsc: watch`.

Observação: é importante carregar `app.js` no final do `body`, ou o React dirá `Error: Target container is not a DOM element`, porque `app.js` estaria chamando `document.getElementById('app')` antes de o elemento app existir.

Neste ponto, vale a pena notar que este código é um pouco "hackeado". Principalmente esta parte:

```
<script>    module = {exports:{}}; exports = {};    window.require = function(name) { return window[na
```

Para que isso? A resposta curta é que: se o seu código contiver `import`, o TypeScript **não poderá** produzir código que "simplesmente funcione" em um navegador e essa é uma das muitas soluções possíveis para esse problema.

A resposta longa? Antes de tudo, lembre-se de que o ecossistema JavaScript possui vários sistemas de módulos. No momento, seu arquivo `tsconfig.json` usa a opção `"module": "umd"`, porque `"module": "umd"` e `"module": "commonjs"` são os únicos modos que podem ser usados no Node.js e em um navegador da web.

Eu pedi para você fazer um arquivo `server.js` (não `server.ts`), mas usando `"module": "umd"` você poderia escrever seu código de servidor em TypeScript se você quiser.

O UMD é a escolha natural já que ele deve fazer uma definição de módulo "universal", mas o TypeScript não tenta realmente ser universal — ele simplesmente não tenta funcionar em um navegador da web sem ajuda.

Em vez disso, ele espera encontrar símbolos predefinidos para um sistema de módulos AMD ou um sistema de módulo CommonJS (ou Node.js). Se nenhum desses estiver definido, o módulo sai sem sequer registrar uma mensagem de erro.

Mesmo se **pudéssemos** usar a opção `"module": "es6"`, que mantém os comandos `import` inalterados no arquivo de saída, não funcionaria porque o Chrome de alguma forma **ainda** não suporta o `import` no momento de escrita desse texto. Além disso, os URLs de nossos módulos têm pouco em comum com a string em nossas declarações `import` e aprendi que os aliases de mapeamento de caminho do TypeScript (texto em inglês) não podem resolver o problema porque não alteram a saída do compilador.

A implementação do CommonJS do TypeScript requer que o `require` seja definido, claro — é usado para importar módulos. Ele, porém, também procura por `exports` e `module.exports`, mesmo que o nosso módulo não exporte nada. Então, nosso pequeno "hack" deve definir todos os três.

As versões UMD do React, ReactDOM e Preact definem variáveis globais chamadas `React`, `ReactDOM` e `preact`, respectivamente. Variáveis "globais" em um navegador, porém, são, na verdade, membros de um objeto especial chamado `window`. Em JavaScript, `window.algumacoisa` significa exatamente a mesma coisa que `window['alguma_coisa']` exceto que você pode usar caracteres especiais, como traços, na última forma. Portanto, `window['preact']` e/ou `window['React']` já existem. Então, definir uma função `require` que simplesmente retorna `window[nome]` permite que React ou Preact sejam importados.

No entanto, também precisamos criar aliases em letras minúsculas `'react'` e `'react-dom'` porque esses são os nomes que devemos usar em nosso código TypeScript (esses nomes são reconhecidos pelo compilador TypeScript, porque são os nomes das pastas em `node_modules`).

Há outra coisa em nosso index.html que é um pouco... infeliz:

```
<script src="node_modules/react/umd/react.development.js"></script><script src="node_modules/react-dom
```

O que faz com que esse código não seja o ideal?

1. Já temos declarações `import` em nosso arquivo `app.tsx`. Então, é um infortúnio que precisemos de um comando **separado** para carregar os módulos em nosso `index.html`.
2. Estamos nos referindo especificamente às versões de **desenvolvimento** do código, que incluem comentários e são muito mais legíveis do que as versões minificadas. Se, contudo, lançarmos nosso site para um grande público, vamos querer mudar para as versões reduzidas para que as páginas carreguem mais rápido. Seria bom se pudéssemos fazer isso sem perder os benefícios de depuração das versões de desenvolvimento.
3. Ele assume que podemos acessar arquivos em `node_modules`, que é uma maneira incomum de configurar um servidor.

Todas as desvantagens descritas aqui nos levam a querer algum tipo de ferramenta adicional para nos ajudar a implantar código em nosso navegador da web. Já discutimos o Parcel, mas o mais popular é o Webpack.

Executando seu projeto – Abordagem C: a maneira do Webpack

A coisa mais popular a se fazer com aplicativos front-end é “empacotar” todos os módulos (React + seu código + qualquer outra coisa que você precise) em um único arquivo. Isso é comparável ao que é chamado de "linking" em algumas outras linguagens, como o C++. Isso é, basicamente, o que o Parcel e o Webpack são feitos para fazer (o Gulp, não — ele requer ferramentas extras instaladas separadamente).

Etapas C1 e C2: criar os arquivos tsconfig.json e server.js

Se você pulou a abordagem B, faça as etapas B1 e B4 agora.

Etapa C3: instalar o webpack

Você **poderia** instalá-lo assim:

```
npm install --save-dev webpack webpack-cli
```

Infelizmente, o Webpack é muito grande: esses dois pacotes têm 735 dependências, pesando at 50.9 MB (13.198 arquivos em 1.868 pastas). Por alguma razão, o `webpack-cli` requer o pacote Webpack, mas não o marca como uma dependência. Portanto, você deve instalar os dois explicitamente.

Embora o `webpack-cli` seja, em grande parte, "apenas" a interface de linha de comando para as APIs do Webpack, ela é desproporcionalmente grande por algum motivo (o Webpack sozinho tem apenas 13,6 MB).

Devido ao seu tamanho, provavelmente faz mais sentido instalá-lo como global:

```
npm install --global webpack webpack-cli
```

Quando usar `--global`, tenha em mente que se você compartilhar seu código com outra pessoa, ela não obterá o Webpack automaticamente quando digitar `npm install`. Assim, você deve explicar como instalar em seu arquivo `README`.

Se você mudar de ideia e quiser mudar de `--save-dev` para `--global`, apenas execute o comando de instalação global `--global` e use `npm uninstall webpack webpack-cli` para excluir a cópia local.

Etapa C4: adicionar scripts de compilação

Para permitir que o `npm` construa e sirva seu projeto, adicione entradas na seção `"scripts"` do `package.json`.

Você **poderia** modificar essa seção para que fique assim:

```
"scripts": {  "test": "echo \"Error: no tests installed\" && exit 1",  "build": "tsc && webpack app/ap
```

Com esses scripts, você usaria ou `npm run build` para compilar uma versão de produção minificada ou `npm run build:dev` para compilar uma versão de desenvolvimento com símbolos e comentários completos. No entanto, isso é inconveniente, pois quando você altera seu código, precisa repetir manualmente o comando `npm run build:dev`.

Na Abordagem B, poderíamos usar `tsc: watch` no VS Code, mas isso não funcionará dessa vez, porque nós **também** precisamos executar o Webpack — e o `tsc` não conhece isso.

Podemos configurá-lo para reconstruir automaticamente quando nosso código for alterado? Sim, mas precisaremos de um plugin do Webpack para fazer isso. Um dos plugins que podem fazer o trabalho é chamado de `awesome-typescript-loader`. Instale assim:

```
npm install awesome-typescript-loader --save-dev
```

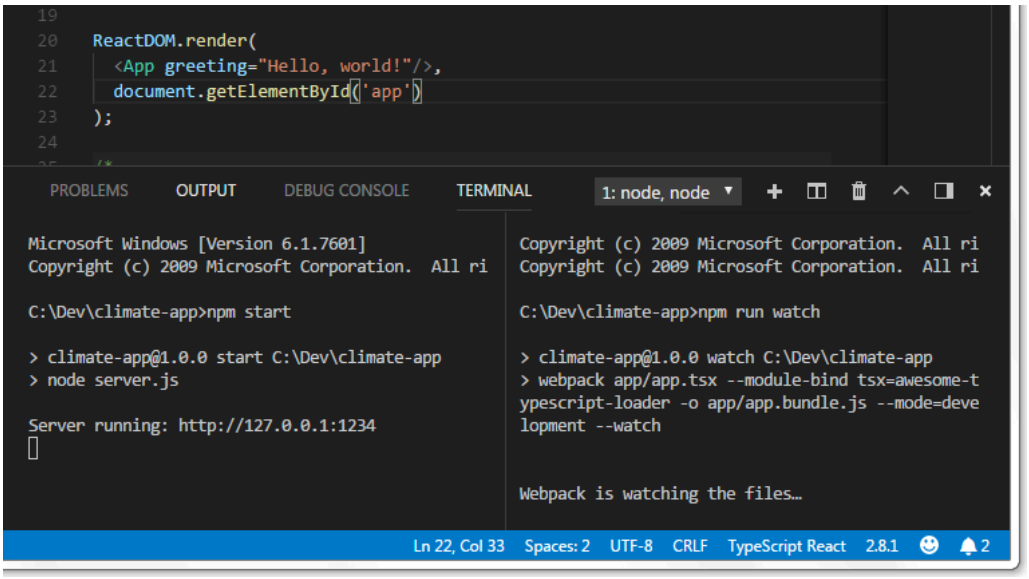
Então, em `package.json`, altere sua seção `"scripts"` para parecer com isso:

Isso torna o `webpack` completamente responsável por compilar nosso código TypeScript e, portanto, podemos usar sua opção `--watch` para observar as mudanças no código. O comando para compilar e observar as alterações de código é `npm run watch`.

Etapa C5: iniciar o servidor e o Webpack

Você precisará de dois terminais separados, um para seu sistema de compilação (`npm run watch`) e outro para seu servidor (`npm start`). Se o seu servidor estiver escrito em TypeScript, você precisará executar `npm run watch` primeiro. Caso contrário, não importa qual você inicie primeiro.

Vale a pena notar que o VS Code pode acompanhar vários terminais. Você pode criar dois terminais e executar um comando em cada um, assim:



Etapa C6: criar um index.html e carregá-lo

Na Abordagem C, seu arquivo `index.html` é muito mais simples do que na Abordagem B:

```
<!DOCTYPE html><html><head>  <title>App</title>  <meta charset="utf-8"/></head><body>  <h1>Mini React
```

Visite `http://127.0.0.1:1234` e a página deve carregar. Você terminou!

Etapa C7: criar um arquivo webpack.config.js (opcional)

Nosso comando de compilação está ficando bastante longo e é muito semelhante para nossos três modos. Além disso, apenas configuramos a extensão de arquivo `tsx`. Portanto, o `webpack` ainda não sabe como compilar arquivos `ts`.

A maneira mais popular de usar o Webpack é com um arquivo de configuração especial, separado do `package.json`. O script `"build"` acima se torna o seguinte arquivo `webpack.config.js`:

```
module.exports = {  entry: __dirname+'/app/app.tsx',  output: {    path: __dirname+'/app',    filename
```

Depois de criar este arquivo, altere seus `scripts` em `package.json`.

Como antes, você pode compilar e observar as alterações com o `npm run watch` ou usar `npm run build` para uma compilação de produção minificada.

Você terminou!

É isso! Clique aqui para obter um resumo de todas as etapas acima e aqui para continuar aprendendo sobre o TypeScript.
