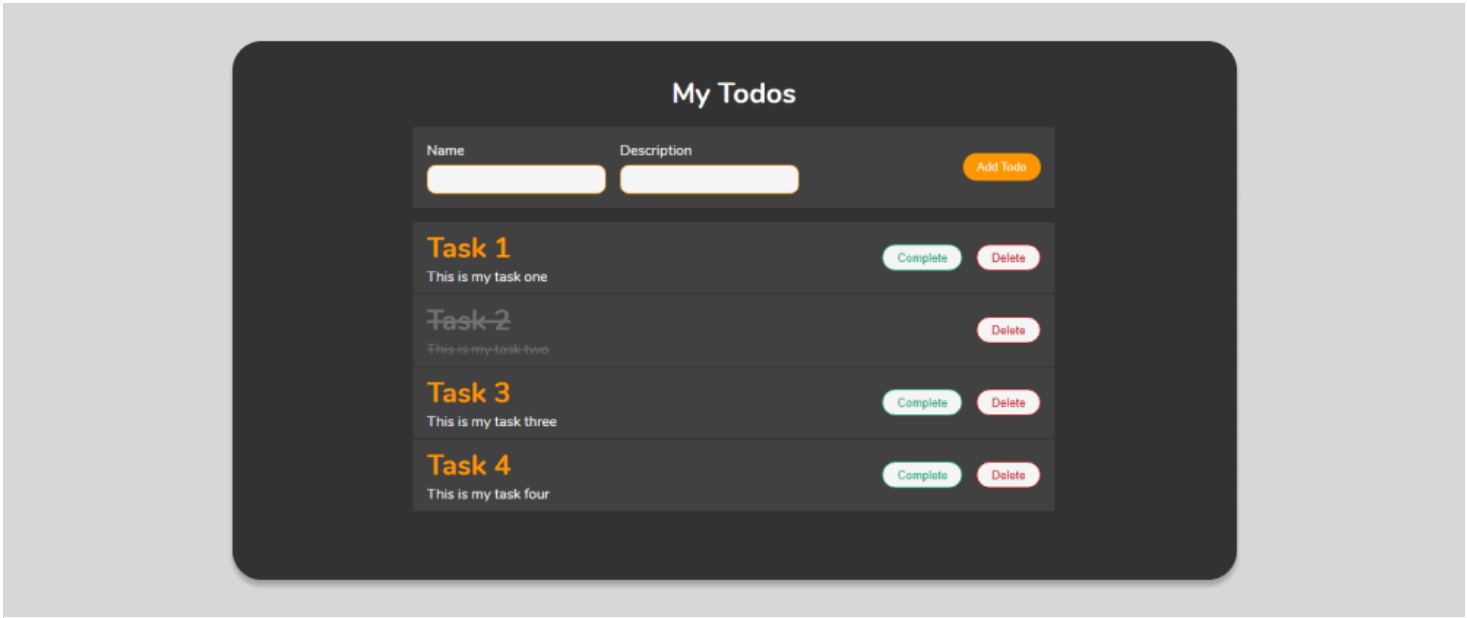
3 DE OUTUBRO DE 2023 / #TYPESCRIPT

Como criar uma aplicação de lista de tarefas com React, TypeScript, NodeJS e MongoDB

Tradutor: Keveen Tenereli

Autor: freeCodeCamp.org (em inglês)



Artigo original: How to Build a Todo App with React, TypeScript, NodeJS, and MongoDB

Neste tutorial, usaremos TypeScript em ambos os lados (*client* e servidor) para desenvolver uma aplicação de lista de tarefas do zero com React, NodeJS, Express e MongoDB.

Então, vamos começar pelo planejamento da API.

- API com NodeJS, Express, MongoDB e TypeScript
- Início
- Criar um tipo de tarefas
- Criar um modelo de tarefas
- Criar controladores de API
- Get, Add, Update e Delete de tarefas
- Criar rotas da API
- Criar um servidor

- Lado do *client* com React e TypeScript
- Configuração
- Criar um tipo de tarefas
- Buscar dados da API
- Criar os componentes
- Adicionar formulário de tarefas
- Exibir uma tarefa
- Buscar e exibir dados
- Recursos

Vamos lá.

API com NodeJS, Express, MongoDB e TypeScript

Início

Se você é novo nisso, pode começar com A Practical Guide to TypeScript ou com How to build an API from scratch with Node JS, Express, and MongoDB (textos em inglês) para tirar o máximo possível deste tutorial. Caso contrário, vamos começar.

Para criar uma aplicação do NodeJS, você precisar executar esse comando no terminal:

```
yarn init
```

Ele perguntará algumas coisas e, então, a aplicação será inicializada. Você pode pular isso adicionando uma flag `-y` ao comando.

Depois, estruture o projeto da seguinte maneira:

```
├── dist
├── node_modules
├── src
   ├── app.ts
   ├── controllers
   |  └── todos
   |     └── index.ts
   ├── models
   |  └── todo.ts
   ├── routes
   |  └── index.ts
   └── types
      └── todo.ts
├── nodemon.json
├── package.json
├── tsconfig.json
```

Como você pode observar, a estrutura desse arquivo é relativamente simples. O diretório `dist` servirá como uma pasta para os resultados uma vez que o código tenha sido compilado para JavaScript simples.

Também temos um arquivo `app.ts` que é o ponto de entrada do servidor. Os controladores, tipos e rotas também estão em suas respectivas pastas.

Agora, precisamos configurar o arquivo `tsconfig.json` para auxiliar o compilador a seguir nossas preferências.

- tsconfig.json

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "dist/js",
    "rootDir": "src",
    "strict": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["src/types/*.ts", "node_modules", ".vscode"]
}
```

Aqui, temos as quatro principais propriedades para destacar:

`outDir`: diz para o compilador colocar o código compilado dentro da pasta `dist/js`.

`rootDir`: informa ao TypeScript para compilar todos os arquivos `.ts` que se encontrarem na pasta `src`.

`include`: diz ao compilador para incluir arquivos que estão no diretório e nos sub-diretórios de `src`.

`exclude`: excluirá os arquivos ou pastas inseridos no *array* durante o tempo de compilação.

Agora, podemos instalar as dependências para habilitar o TypeScript no projeto. Por padrão, essea aplicação usaria JavaScript.

Existem duas maneiras de usar TypeScript em uma aplicação do NodeJS. Localmente, no projeto, ou globalmente, em nossa máquina. Vou optar pela última opção com base em minha preferência pessoal, mas você pode continuar com a forma local se quiser.

Vamos executar o seguinte comando no terminal e instalar o TypeScript.

```
yarn add typescript -g
```

A flag `g` permite instalar o TypeScript globalmente. Isso faz com que seja acessível de qualquer lugar no computador.

Depois, vamos adicionar algumas dependências para poder usar o Express e o MongoDB.

```
yarn add express cors mongoose
```

Também precisamos instalar tipos como depêndencias de desenvolvimento para ajudar o compilador do TypeScript a entender os pacotes.

```
yarn add -D @types/node @types/express @types/mongoose @types/cors
```

Agora, o TypeScript não vai mais gritar com você – ele usará esses tipos para definir as bibliotecas que acabamos de instalar.

Também precisamos adicionar outras dependências para poder compilar o código em TypeScript e inicializar o servidor simultaneamente.

```
yarn add -D concurrently nodemon
```

Com isso pronto, podemos atualizar o arquivo `package.json` com os scripts necessários para iniciar o servidor.

- package.json

```
"scripts": {
  "build": "tsc",
  "start": "concurrently \"tsc -w\" \"nodemon dist/js/app.js\""
}
```

`concurrently` ajudará a compilar o código TypeScript, observar mudanças e iniciar o servidor simultaneamente. Dito isso, podemos iniciar o servidor – embora, ainda não tenhamos criado algo significativo nesse sentido. Vamos corrigir isso na próxima sessão.

Criar um tipo de tarefas

- types/todo.ts

```
import { Document } from "mongoose"

export interface ITodo extends Document {
  name: string
  description: string
  status: boolean
}
```

Aqui, temos uma interface de tarefas (Todo - em português, tarefa ou afazer) que estende o tipo `Document` fornecido pelo `mongoose`. Vamos utilizar essa interface mais tarde para que interaja com o MongoDB. Dito

isso, agora, podemos definir como o modelo de tarefa deve ser.

Criar um modelo de tarefa

- models/todo.ts

```
import { ITodo } from "./../types/todo"
import { model, Schema } from "mongoose"

const todoSchema: Schema = new Schema(
  {
    name: {
      type: String,
      required: true,
    },

    description: {
      type: String,
      required: true,
    },

    status: {
      type: Boolean,
      required: true,
    },
  },
  { timestamps: true }
)

export default model<ITodo>("Todo", todoSchema)
```

Como podemos ver aqui, começamos implementando a interface `ITodo` e algumas utilidades do `mongoose`. Essa última parte ajuda a definir o esquema Todo e também a passar `ITodo` como um tipo para o `model` antes de exportá-lo.

Com isso, agora podemos usar o modelo de tarefa em outros arquivos para interagir com o banco de dados.

Criar controladores API

Get, Add, Update e Delete de tarefas

- controllers/todos/index.ts

```
import { Response, Request } from "express"
import { ITodo } from "./../../types/todo"
import Todo from "../../models/todo"

const getTodos = async (req: Request, res: Response): Promise<void> => {
  try {
    const todos: ITodo[] = await Todo.find()
    res.status(200).json({ todos })
  } catch (error) {
    throw error
  }
}
```

Aqui, primeiro, precisamos importar alguns tipos do `express`, pois quero digitar os valores explicitamente. Se você quiser, pode deixar o TypeScript inferi-los para você.

Depois, usamos a função `getTodos()` para puxar os dados. Ela recebe os parâmetros `req` e `res` e retorna uma *promise*.

Com a ajuda do modelo `Todo` que criamos mais cedo, agora, podemos receber dados do MongoDB e retornar uma resposta com o *array* de tarefas.

- controllers/todos/index.ts

```
const addTodo = async (req: Request, res: Response): Promise<void> => {
  try {
    const body = req.body as Pick<ITodo, "name" | "description" | "status">

    const todo: ITodo = new Todo({
      name: body.name,
      description: body.description,
      status: body.status,
    })

    const newTodo: ITodo = await todo.save()
    const allTodos: ITodo[] = await Todo.find()

    res
      .status(201)
      .json({ message: "Todo added", todo: newTodo, todos: allTodos })
  } catch (error) {
    throw error
  }
}
```

Como você pode ver, a função `addTodo()` recebe o objeto body, que contém dados introduzidos pelo usuário.

Depois, fazemos uso de *typecasting* para evitar erros de digitação e restringir a variável `body` para que corresponda a `ITodo` e, então, criamos uma tarefa (um *Todo*) baseado no modelo.

Com isso pronto, podemos salvar a tarefa no banco de dados e retornar a resposta que contém a tarefa criada, além de atualizar o *array* de tarefas.

- controllers/todos/index.ts

```
const updateTodo = async (req: Request, res: Response): Promise<void> => {
  try {
    const {
      params: { id },
      body,
    } = req
    const updateTodo: ITodo | null = await Todo.findByIdAndUpdate(
      { _id: id },
      body
    )
    const allTodos: ITodo[] = await Todo.find()
    res.status(200).json({
      message: "Todo updated",
```

```
      todo: updateTodo,
      todos: allTodos,
    })
  } catch (error) {
    throw error
  }
}
```

Para atualizar uma tarefa, precisamos extrair a id e o body do objeto `req` e, então, passá-los para `findByIdAndUpdate()`. Essa utilidade encontrará a tarefa no banco de dados e a atualizará. Uma vez que a operação tenha sido completada, podemos retornar os dados atualizados ao usuário.

- controllers/todos/index.ts

```
const deleteTodo = async (req: Request, res: Response): Promise<void> => {
  try {
    const deletedTodo: ITodo | null = await Todo.findByIdAndRemove(
      req.params.id
    )
    const allTodos: ITodo[] = await Todo.find()
    res.status(200).json({
      message: "Todo deleted",
      todo: deletedTodo,
      todos: allTodos,
    })
  } catch (error) {
    throw error
  }
}

export { getTodos, addTodo, updateTodo, deleteTodo }
```

A função `deleteTodo()` permite que você exclua uma tarefa do banco de dados. Aqui, puxamos a id de `req` e passamos como um argumento em `findByIdAndRemove()` para accessar a tarefa correspondente e exclui-la do banco de dados.

Depois, exportamos as funções para que possamos usá-las em outros arquivos. Dito isso, podemos criar algumas rotas para a API e usar esses métodos para lidar com os pedidos.

Criar rotas da API

- routes/index.ts

```
import { Router } from "express"
import { getTodos, addTodo, updateTodo, deleteTodo } from "../controllers/todos"

const router: Router = Router()

router.get("/todos", getTodos)

router.post("/add-todo", addTodo)

router.put("/edit-todo/:id", updateTodo)

router.delete("/delete-todo/:id", deleteTodo)
```

```
export default router
```

Como você pode ver aqui, temos quatro rotas para obter, adicionar, atualizar e excluir tarefas do banco de dados. Como já criamos as funções, a única coisa que temos que fazer é importar os métodos e passá-los como parâmetros para lidar com os pedidos.

Até então, cobrimos bastante. Ainda não temos, no entanto, um servidor para inicializar. Vamos corrigir isso na próxima seção.

Criar um servidor

Antes de criar o servidor, precisamos, primeiro, adicionar algumas variáveis do ambiente que guardarão as credenciais do MongoDB no arquivo `nodemon.json`.

- nodemon.json

```
{
    "env": {
        "MONGO_USER": "seu-nome-de-usuario",
        "MONGO_PASSWORD": "sua-senha",
        "MONGO_DB": "nome-do-banco-de-dados"
    }
}
```

Você pode obter as credenciais criando um cluster no MongoDB Atlas.

- app.ts

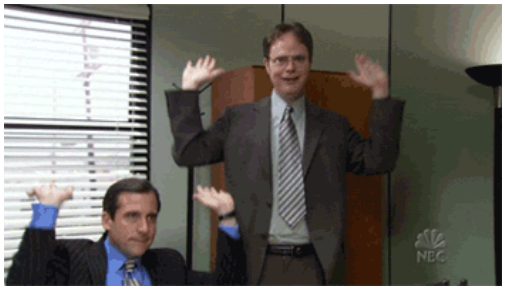
```
import express, { Express } from "express"
import mongoose from "mongoose"
import cors from "cors"
import todoRoutes from "./routes"

const app: Express = express()

const PORT: string | number = process.env.PORT || 4000

app.use(cors())
app.use(todoRoutes)

const uri: string = `mongodb+srv://${process.env.MONGO_USER}:${process.env.MONGO_PASSWORD}@clustertodo.raz9g.mo
const options = { useNewUrlParser: true, useUnifiedTopology: true }
mongoose.set("useFindAndModify", false)

mongoose
  .connect(uri, options)
  .then(() =>
    app.listen(PORT, () =>
      console.log(`Server running on http://localhost:${PORT}`)
    )
  )
  .catch(error => {
    throw error
  })
```

Aqui, começamos importando a biblioteca `express`, que permite que acessemos o método `use()`, que, por sua vez, ajuda a lidar com as rotas das tarefas.

Depois, usamos o pacote `mongoose` para conectar com o MongoDB, ao adicionar ao URL as credenciais que estão no arquivo `nodemon.json`.

Dito isso, se agora conectarmos com sucesso ao MongoDB, o servidor iniciará. Se for apropriado, um erro será mostrado.

Terminamos de criar a API com Node, Express, TypeScript e MongoDB. Vamos começar a criar a aplicação do lado do *client* com React e TypeScript.



Lado do *client* com React e TypeScript

Configuração

Para criar uma aplicação com React, usarei o create-react-app – você pode usar outros métodos, se quiser.

Então, vamos rodar o seguinte comando no terminal:

```
npx create-react-app my-app --template typescript
```

Depois, instale a biblioteca Axios para poder buscar dados remotamente.

```
yarn add axios
```

Uma vez que a instalação tenha terminado, vamos ter a estrutura do nosso projeto do seguinte modo:

```
├── node_modules
├── public
├── src
|   ├── API.ts
|   ├── App.test.tsx
|   ├── App.tsx
|   ├── components
|   |   ├── AddTodo.tsx
|   |   └── TodoItem.tsx
```

```
|  ├── index.css
|  ├── index.tsx
|  ├── react-app-env.d.ts
|  ├── setupTests.ts
|  └── type.d.ts
├── tsconfig.json
├── package.json
└── yarn.lock
```

Aqui, temos uma estrutura de arquivos relativamente simples. A principal coisa para se notar é que `src/type.d.ts` vai manter dois tipos. Como eu vou usá-los em quase todo arquivo, eu adicionei a extensão `.d.ts` para fazer com que os tipos sejam disponibilizados globalmente. Agora, não precisamos mais importá-los.

Criar um tipo de tarefa

- src/type.d.ts

```
interface ITodo {
  _id: string
  name: string
  description: string
  status: boolean
  createdAt?: string
  updatedAt?: string
}

interface TodoProps {
  todo: ITodo
}

type ApiDataType = {
  message: string
  status: string
  todos: ITodo[]
  todo?: ITodo
}
```

Aqui, a interface `ITodo` precisa espelhar a forma dos dados da API. Como não temos o `mongoose` aqui, precisamos adicionar outras propriedades para combiná-las com o tipo definido na API.

Depois, usaremos essa mesma interface para o `TodoProps`, que é a anotação do tipo para a *props* que será recebida pelo componente responsável por renderizar os dados.

Agora que definimos os nossos tipos, vamos agora começar a buscar dados da API.

Buscar dados da API

- src/API.ts

```
import axios, { AxiosResponse } from "axios"

const baseUrl: string = "http://localhost:4000"

export const getTodos = async (): Promise<AxiosResponse<ApiDataType>> => {
  try {
```

```
    const todos: AxiosResponse<ApiDataType> = await axios.get(
      baseUrl + "/todos"
    )
    return todos
  } catch (error) {
    throw new Error(error)
  }
}
```

Como você pode ver, precisamos importar o `axios` para fazer solicitações de dados da API. Depois, usamos a função `getTodos()` para pegar dados do servidor. Ela retornará uma *promise* do tipo `AxiosResponse`, que contém as tarefas buscadas, as quais precisam combinar com o tipo `ApiDataType`.

- src/API.ts

```
export const addTodo = async (
  formData: ITodo
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const todo: Omit<ITodo, "_id"> = {
      name: formData.name,
      description: formData.description,
      status: false,
    }
    const saveTodo: AxiosResponse<ApiDataType> = await axios.post(
      baseUrl + "/add-todo",
      todo
    )
    return saveTodo
  } catch (error) {
    throw new Error(error)
  }
}
```

Essa função recebe os dados inseridos pelo usuário como um argumento e retorna a *promise*. Aqui, precisamos omitir a propriedade `_id`, porque o MongoDB a criará na hora.

- src/API.ts

```
export const updateTodo = async (
  todo: ITodo
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const todoUpdate: Pick<ITodo, "status"> = {
      status: true,
    }
    const updatedTodo: AxiosResponse<ApiDataType> = await axios.put(
      `${baseUrl}/edit-todo/${todo._id}`,
      todoUpdate
    )
    return updatedTodo
  } catch (error) {
    throw new Error(error)
  }
}
```

Para atualizar uma tarefa, precisamos passar os dados atualizados e a `_id` do objeto. Aqui, precisamos mudar o `status` da tarefa. É por isso que eu pego apenas a propriedade que precisamos antes de mandar o pedido ao servidor.

- src/API.ts

```
export const deleteTodo = async (
  _id: string
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const deletedTodo: AxiosResponse<ApiDataType> = await axios.delete(
      `${baseUrl}/delete-todo/${_id}`
    )
    return deletedTodo
  } catch (error) {
    throw new Error(error)
  }
}
```

Aqui, também temos a função que recebe como parâmetro a propriedade `_id` e retorna a *promise*.

Com isso pronto, agora podemos acessar a pasta `components` e adicionar algumas linhas de código importantes aos seus arquivos.

Criar os componentes

Adicionar o formulário de tarefas

- components/AddTodo.tsx

```
import React from "react"

type Props = TodoProps & {
  updateTodo: (todo: ITodo) => void
  deleteTodo: (_id: string) => void
}

const Todo: React.FC<Props> = ({ todo, updateTodo, deleteTodo }) => {
  const checkTodo: string = todo.status ? `line-through` : ""
  return (
    <div className="Card">
      <div className="Card--text">
        <h1 className={checkTodo}>{todo.name}</h1>
        <span className={checkTodo}>{todo.description}</span>
      </div>
      <div className="Card--button">
        <button
          onClick={() => updateTodo(todo)}
          className={todo.status ? `hide-button` : "Card--button__done"}
        >
          Complete
        </button>
        <button
          onClick={() => deleteTodo(todo._id)}
          className="Card--button__delete"
        >
          Delete
        </button>
      </div>
```

```
    </div>
  )
}

export default Todo
```

Como você pode ver, aqui, temos um componente funcional do tipo `React.FC` (FC significa componente funcional – do inglês, *functional component*). Ele recebe como *prop* o método `saveTodo()`, que nos permite salvar os dados no banco de dados.

Depois, temos o *state* `formData`, que precisa combinar com o tipo do `ITodo` para satisfazer o compilador. Essa é a razão para o passarmos para o *hook* `useState`. Também precisamos adicionar um tipo alternativo (`{}`), pois o *state* incial será um objeto vazio.

Com isso, podemos seguir em frente e visualizar os dados que foram buscados.

Exibir uma tarefa

- components/TodoItem.tsx

```
import React from "react"

type Props = TodoProps & {
  updateTodo: (todo: ITodo) => void
  deleteTodo: (_id: string) => void
}

const Todo: React.FC<Props> = ({ todo, updateTodo, deleteTodo }) => {
  const checkTodo: string = todo.status ? `line-through` : ""
  return (
    <div className="Card">
      <div className="Card--text">
        <h1 className={checkTodo}>{todo.name}</h1>
        <span className={checkTodo}>{todo.description}</span>
      </div>
      <div className="Card--button">
        <button
          onClick={() => updateTodo(todo)}
          className={todo.status ? `hide-button` : "Card--button__done"}
        >
          Complete
        </button>
        <button
          onClick={() => deleteTodo(todo._id)}
          className="Card--button__delete"
        >
          Delete
        </button>
      </div>
    </div>
  )
}

export default Todo
```

Aqui, precisamos estender o tipo `TodoProps` e acrescentar as funções `updateTodo` e `deleteTodo` para lidar apropriadamente com as *props* recebidas pelo componente.

Agora, uma vez que o objeto Todo (a tarefa) tenha sido passado, poderemos exibir e adicionar as funcões necessárias para atualizar ou excluir uma tarefa.

Ótimo! Podemos agora acessar o arquivo `App.tsx` e adicionar a última peça do quebra-cabeça.

Buscar e exibir dados

- App.tsx

```
import React, { useEffect, useState } from 'react'
import TodoItem from './components/TodoItem'
import AddTodo from './components/AddTodo'
import { getTodos, addTodo, updateTodo, deleteTodo } from './API'

const App: React.FC = () => {
  const [todos, setTodos] = useState<ITodo[]>([])

  useEffect(() => {
    fetchTodos()
  }, [])

  const fetchTodos = (): void => {
    getTodos()
    .then(({ data: { todos } }: ITodo[] | any) => setTodos(todos))
    .catch((err: Error) => console.log(err))
  }
```

Aqui, primeiro, precisamos importar os componentes e funções utilitárias do `API.ts`. Depois, passamos um *array* do tipo `ITodo` para o `useState` e o inicializamos com um *array* vazio.

O método `getTodos()` retorna uma *promise* – logo, podemos acessar a função `then` e atualizar o *state* com os dados buscados ou lançar um erro, se algum ocorrer.

Com isso pronto, podemos chamar a função `fetchTodos()` quando o componente é montado com sucesso.

- App.tsx

```
const handleSaveTodo = (e: React.FormEvent, formData: ITodo): void => {
  e.preventDefault()
  addTodo(formData)
    .then(({ status, data }) => {
      if (status !== 201) {
        throw new Error("Error! Todo not saved")
      }
      setTodos(data.todos)
    })
    .catch(err => console.log(err))
}
```

Uma vez que o formulário é submetido, o usamos para enviar o pedido ao servidor. Se a tarefa foi salva com sucesso, atualizamos os dados. Caso contrário, um erro será lançado.

- App.tsx

```
const handleUpdateTodo = (todo: ITodo): void => {
  updateTodo(todo)
    .then(({ status, data }) => {
      if (status !== 200) {
        throw new Error("Error! Todo not updated")
      }
      setTodos(data.todos)
    })
    .catch(err => console.log(err))
}

const handleDeleteTodo = (_id: string): void => {
  deleteTodo(_id)
    .then(({ status, data }) => {
      if (status !== 200) {
        throw new Error("Error! Todo not deleted")
      }
      setTodos(data.todos)
    })
    .catch(err => console.log(err))
}
```

As funções para excluir ou atualizar uma tarefa são bastante similares. As duas recebem um parâmetro, enviam seu pedido e recebem de volta uma resposta. Então, elas verificam se o pedido teve sucesso e lidam com ele de maneira apropriada.

- App.tsx

```
  return (
    <main className='App'>
      <h1>My Todos</h1>
      <AddTodo saveTodo={handleSaveTodo} />
      {todos.map((todo: ITodo) => (
        <TodoItem
          key={todo._id}
          updateTodo={handleUpdateTodo}
          deleteTodo={handleDeleteTodo}
          todo={todo}
        />
      ))}
    </main>
  )
}

export default App
```

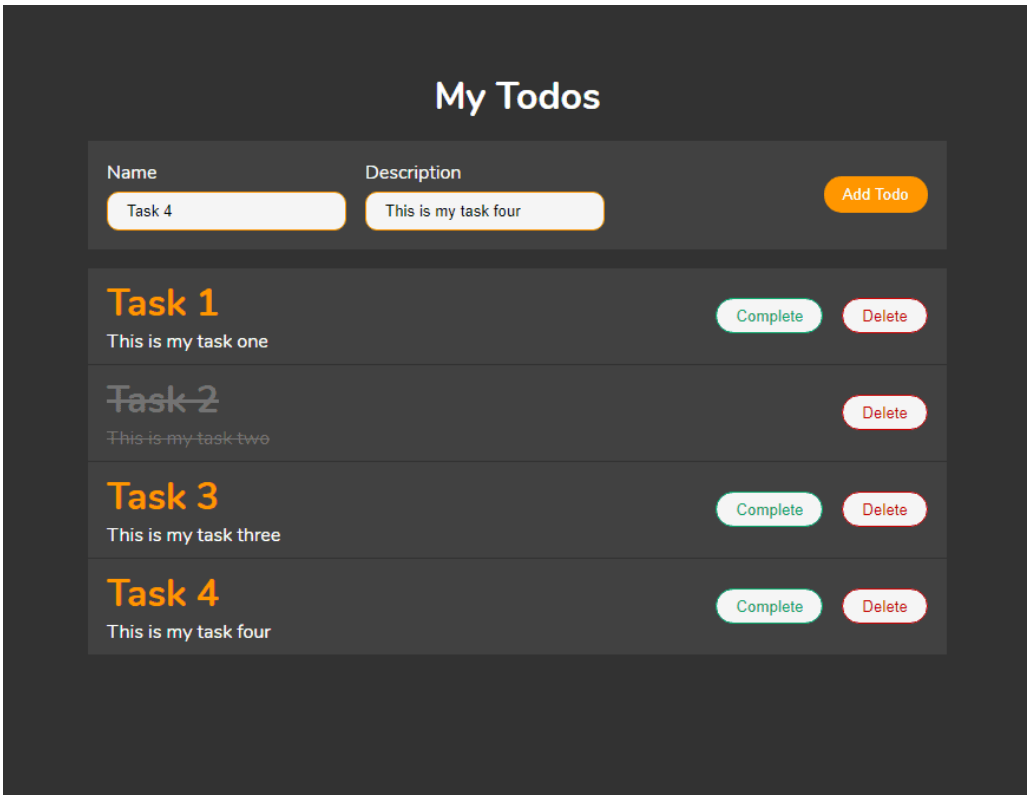
Aqui, iteramos pelo *array* `todos` e, então, passamos para o `TodoItem` os dados esperados.

Navegue até a pasta que contém o lado do servidor da aplicação e execute os seguinte comando no terminal:

```
yarn start
```

Faça o mesmo no lado do client da aplicação:

Você deve ver que nossa aplicação de lista de tarefas funciona como esperado.



Ótimo! Com esse toque final, terminamos de criar uma aplicação de lista de tarefas com TypeScript, React, NodeJs, Express e MongoDB.

Você pode encontrar o código-fonte aqui.

Você pode encontrar outros ótimos conteúdos como esse no blog do autor (em inglês) e seguir o autor no Twitter para ser notificado.

Agradecemos pela leitura.

Recursos (em inglês, no GitHub e no site do autor)

Ficha informativa de TypeScript e React

Ficha informativa de tipos avançados em TypeScript (com exemplos)

Fichas informativas de TypeScript
