

M E R N

Criando sua primeira aplicação (M)ongo, (E)xpress, (R)eact e (N)ode



JEFFERSON HENRIQUE

Criando sua primeira aplicação

(M)ongoDB, (E)xpress, (R)eact e (N)ode.

Por Jefferson Henrique 1a Edição, 13/04/2020.

Nenhuma parte deste livro pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da DevDelivery Community, exceto para resumos breves em revisões e análises.

DevDelivery Community

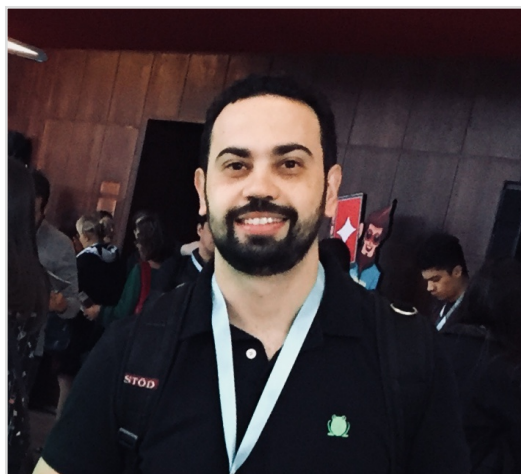
www.dev.delivery

contato@dev.delivery

Siga-nos nas redes sociais e fique por dentro de tudo!



Sobre o Autor



Jefferson Henrique

Arquiteto de Software e co fundador da [uebile](#).

Atua há mais de 15 anos no mercado de desenvolvimento de Software em geral.

Aficionado por tecnologia, música e comportamento humano.

Adora botar em prática desafios usando processos ágeis e inovação como uma mistura mágica para resolver problemas.

LinkedIn: [linkedin.com/jeffersonoh](https://www.linkedin.com/jeffersonoh)

Twitter: [@jeffersonoh](https://twitter.com/jeffersonoh)

Antes de começar...

Antes que você comece a ler esse livro, eu gostaria de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

O que você precisa saber?

Você só conseguirá absorver o conteúdo desse livro se já conhecer pelo menos o básico de JavaScript, HTML e CSS. Caso você ainda não domine esses assuntos avalie estudar um pouco mais esse conteúdo antes de iniciar a leitura deste livro.

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Eu gostaria muito de te ajudar pessoalmente nesses problemas, mas infelizmente não consigo fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando e produzindo conteúdo para nossa comunidade.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, nossa recomendação é que você poste na nossa comunidade no DevDelivery é só acessar:

dev.delivery/member-login

Como sugerir melhorias ou reportar erros sobre este livro?

Se você encontrar algum erro no conteúdo desse livro ou se tiver alguma sugestão para melhorar a próxima edição, vamos ficar muito felizes se você puder nos dizer.

Envie um e-mail para contato@dev.delivery

Onde encontrar o código-fonte do projeto?

Neste livro, nós vamos desenvolver um pequeno projeto passo a passo.

O código fonte deste projeto está armazenado no repositório da DevDelivery Community no GitHub.

É só acessar: github.com/DevDelivery

Lá voce encontrará outros projetos também .

Ajude na continuidade desse trabalho

Escrever um livro (mesmo que pequeno, como esse) dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livro para seus amigos que também se interessam por programação em JavaScript e compartilhe em suas redes sociais.

Sumário

Criando sua primeira aplicação	2
1 Introdução	8
1.1 (M)ongoDB	8
1.2 (E)xpress JS	9
1.3 (R)eact JS	9
1.4 (N)ode JS	10
2 Configurando nosso projeto	10
2.1 Configurando o backend	10
2.1.1 MongoDB	13
2.1.2 Criando nosso esquema	16
2.1.3 Criando rotas	17
2.1.4 Testando manualmente nosso backend	21
2.2 Configurando o frontend	25
2.2.1 Criando nosso projeto	26
2.2.2 Instalando as dependências e ajustando o projeto	28
3 Codificando	29
3.1 Integrando backend e frontend	33
3.2 Implementando a consulta de filmes	37
3.2 Implementando a exclusão de filmes	40
3.3 Implementando a inserção de filmes	46
3.3 Implementando a edição de filmes	50
4 Conclusão	55
4.1 Próximos passos	55

Criando sua primeira aplicação

(M)ongoDB, (E)xpress, (R)eact e (N)ode.

1 Introdução

Às vezes a parte mais difícil para o programador que está começando uma nova aplicação em JavaScript, mesmo se este programador já conheça a linguagem, é justamente começar!

Você precisa criar a estrutura de diretórios para os arquivos, além de criar e configurar o *build file* com as dependências.

Se você já programa em JavaScript, sempre que precisa criar um novo projeto, o que você faz? É possível que sua resposta seja: “eu crio um novo projeto e vou copiando as configurações de outro que já está funcionando”.

Se você é iniciante, seu primeiro passo será procurar algum tutorial que te ensine a criar o projeto do zero, e então copiar e colar todas as configurações no seu ambiente de desenvolvimento até ter o “hello world” estar funcionando.

Esses são cenários comuns no desenvolvimento web quando estamos usando as diversas ferramentas que existem, mas aqui vamos usar uma alternativa a este “castigo inicial” da criação de um novo projeto, e esse é um dos objetivos desse livro, mostrar um caminho mais fácil e prazeroso para criar um projeto web feito em JavaScript.

Vamos utilizar um conjunto de tecnologias aqui chamadas de **MERN** para criar um *CRUD* básico totalmente feito em **JavaScript**.

1.1 (M)ongoDB

[MongoDB](#) é um poderoso banco de dados NoSQL flexível e escalável. Combina a habilidade de escalar com os muitos recursos presentes nos bancos de dados relacionais como índices, ordenação etc.

Neste nosso livro, o MongoDB vai nos permitir armazenar e reaver um dado em um formato muito similar ao JSON (*JavaScript Object Notation*) que veremos mais à frente. É um banco fracamente tipado, o que ajuda muito pouco com validação de dados, de modo que boa parte da responsabilidade das regras de validação fica nas mãos dos desenvolvedores.

1.2 (E)xpress JS

O [Express JS](#), ou simplesmente Express, é uma estrutura de aplicativo da Web para o Node.js, lançada como software livre e de código aberto sob a Licença MIT. Ele foi projetado para criar aplicativos da Web e APIs. Foi chamado de estrutura de servidor padrão de fato para o Node.js.

Neste nosso livro, o Express fornecerá a estrutura necessária para a criação de nossos endpoints RESTful, as chamadas API (Application Programming Interface).

1.3 (R)eact JS

[React JS](#) é uma biblioteca JavaScript de código aberto com foco em criar interfaces de usuário (frontend) em páginas web. É mantido pelo Facebook, Instagram e outras empresas de comunidade de desenvolvedores individuais. É utilizado nos sites da Netflix, Feedly, Airbnb e outros.

Neste nosso livro, o ReactJs será utilizado para a criação de nosso frontend nossas páginas web.

1.4 (N)ode JS

[Node JS](#) é uma plataforma para aplicações JavaScript criada por Ryan Dahl sob o ambiente de execução JavaScript do Chrome. É possível utilizar bibliotecas desenvolvidas pela comunidade através de seu gerenciador de pacotes chamado **npm**.

Neste nosso livro, o Node.js será usado para implementar nosso *backend* onde serão armazenada nossa camada de negócio.

2 Configurando nosso projeto

2.1 Configurando o backend

Aqui, criaremos o lado do servidor do nosso aplicativo, onde criaremos um serviço *RESTful* seguindo algumas etapas.

Primeiro, vamos criar um diretório vazio que será a raiz do nosso sistema.

```
$ mkdir mern-sample  
$ cd mern-sample
```

Depois disso, vamos criar outra pasta vazia chamada **server** que será nosso backend.

```
$ mkdir server  
$ cd mern-backend
```

Aqui, criaremos nosso **package.json** .

O arquivo *package.json* é apenas um manifesto para o seu projeto Node.js. Ele contém os metadados. Você pode gerenciar as dependências do seu projeto e criar scripts que o ajudarão a instalar dependências, gerar builds, executar testes e outras coisas.

Para criar um *package.json*, você precisa de um gerenciador de pacotes, pode escolher [NPM](#) (Node Package Manager) ou [YARN](#). Sinta-se livre para usar o que preferir.

Então, vamos criar nosso arquivo. Escolhemos aqui para este projeto o gerenciador de pacote **NPM**.

```
$ npm init -y
```

Depois disso, você pode encontrar o arquivo na pasta do servidor.

```
$ ls  
package.json
```

Com esse arquivo, podemos instalar nossas dependências.

```
$ npm install express body-parser cors mongoose nodemon
```

O que foi instalado ?

- **express** : É a estrutura do servidor (O E em M E R N).
- **body-parser**: Responsável por retirar o corpo da solicitação de rede.
- **nodemon**: Reinicia o servidor quando houver alterações (para uma melhor experiência do desenvolvedor).
- **cors**: Pacote para fornecer um middleware *connect/express* que pode ser usado para ativar o *CORS* com várias opções.
- **mongoose**: é uma modelagem de objeto elegante do MongoDB para node.js

Agora que você conhece as dependências básicas que precisamos em nosso projeto.

Se você listar a pasta, notará que algo mudou.

```
$ ls  
node_modules package.json package-lock.json
```

Observe a nova pasta node **node_modules** e o novo arquivo package-lock.json.

Com esses dois arquivos, seu projeto pode ser interpretado.

Agora podemos criar nosso primeiro arquivo NodeJS **index.js**.

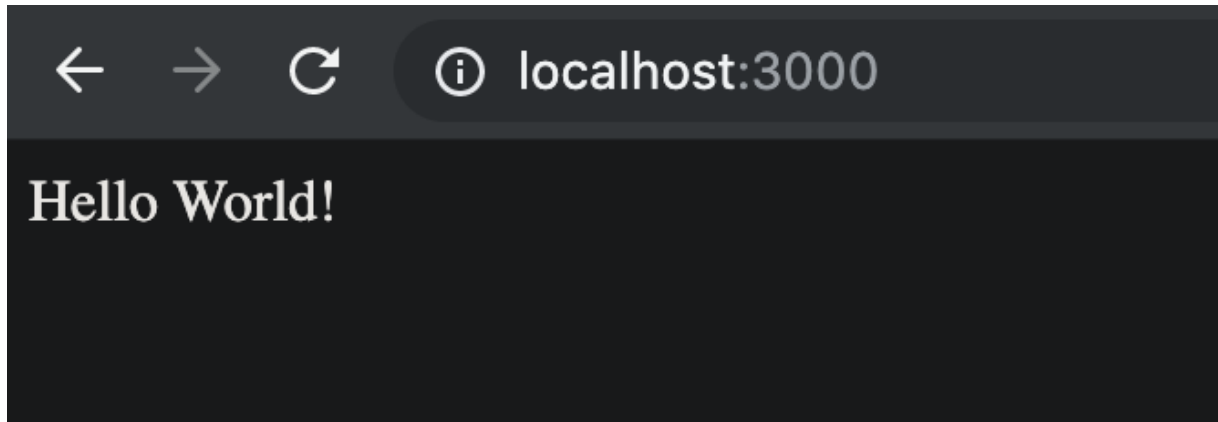
```
const express = require('express')  
const bodyParser = require('body-parser')  
const cors = require('cors')  
const app = express()  
const apiPort = 3000  
  
app.use(bodyParser.urlencoded({ extended: true }))  
app.use(cors())  
app.use(bodyParser.json())  
  
app.get('/', (req, res) => {  
  res.send('Hello World!')  
})  
  
app.listen(apiPort, () => console.log('Server running on port 3000'));
```

Para iniciar o aplicativo, você só precisa fazer:

```
$ node index.js
```

Se você puder ver a mensagem "Server running on port 3000", significa que tudo o que você fez até agora está correto.

Você pode abrir um navegador e digitar localhost:3000, você verá a mensagem "Hello World".



2.1.1 MongoDB

Por enquanto, precisamos só instalar o MongoDB. Para isso, basta digitar no seu terminal:

```
$ brew tap mongodb/brew  
$ brew install mongodb-community  
$ brew services start mongodb-community
```

Os comandos acima são para sistema operacional macOS, caso você possua outro sistema operacional basta seguir essas instruções [aqui](#).

```
$ brew services start mongod
```

O próximo passo é criar nosso banco de dados que chamamos aqui de **sample**.

```
$ mongo> use sample
```

```

jefferson@MacBook-Pro-Jefferson mern-sample % mongo
MongoDB shell version v4.2.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("d68b2d82-83ad-478b-82dc-e0ee50142268") }
MongoDB server version: 4.2.5
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
Server has startup warnings:
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten]
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten] ** WARNING: Access control is not enabled f
or the database.
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten] **           Read and write access to data a
nd configuration is unrestricted.
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten]
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten]
2020-04-07T07:20:31.670-0300 I  CONTROL  [initandlisten] ** WARNING: soft rlimits too low. Number of
files is 256, should be at least 1000
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> use sample
switched to db sample
>

```

E é isso, acabamos de criar nosso banco de dados com esses comandos.

Voltando ao nosso código JavaScript, precisamos agora criar a conexão do nosso server usando a biblioteca **Mongoose**.

Vamos criar um novo diretório chamado **db** (dentro da pasta do server) e um novo arquivo chamado **index.js** dentro dele.

```

$ mkdir db
$ touch index.js

```

```
const mongoose = require('mongoose')
```

```
mongoose
  .connect('mongodb://127.0.0.1:27017/sample', { useNewUrlParser: true })
  .catch(e => {
    console.error('Connection error', e.message)
  })
```

```
const db = mongoose.connection
```

```
module.exports = db
```

Atualizando o arquivo de *index.js* em **mern/index.js**, temos algo parecido com isto.

```
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')
```

```
const db = require('./db')
```

```
const app = express()
const apiPort = 3000
```

```
app.use(bodyParser.urlencoded({ extended: true }))
app.use(cors())
app.use(bodyParser.json())
```

```
db.on('error', console.error.bind(console, 'MongoDB connection error:'))
```

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(apiPort, () => console.log('Server running on port 3000'));
```

Se você der uma olhada no seu terminal, verá que nada mudou. Isso ocorre quando você digita o **node index.js**, o que significa que o node mantém a última configuração antes de iniciar. Se você deseja reiniciar o aplicativo, feche e abra um novo.

No entanto, no começo deste livro, eu falei sobre o **nodemon**, ele reiniciará o servidor toda vez que houver alterações em qualquer arquivo do nosso projeto. De agora em diante vamos fazer isso.

```
$ nodemon index.js
```

```
jefferson@MacBook-Pro-Jefferson mern-backend % nodemon index.js
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
(node:12804) DeprecationWarning: current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.
Server running on port 3000
```

2.1.2 Criando nosso esquema

Vamos abordar para este livro a construção de um sistema para informar horários de exibição de filmes que estão sendo exibidos em um cinema e precisaremos criar nossas entidades de relacionamento.

Dentro deste escopo precisamos criar uma entidade chamada **filmes** que deve ser composta pelos filmes de um cinema. Para isso, precisamos apenas saber o nome do filme e os horários em que o filme está passando no cinema e adicionar uma informação importante.

Vamos criar uma pasta chamada **models** e adicionar um arquivo chamado *movie.js* dentro de **server**.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema
```



```
const Movie = new Schema(  
  {  
    name: { type: String, required: true },  
    time: { type: [String], required: true },  
    rating: { type: Number, required: true },  
  },  
  { timestamps: true },  
)  
  
module.exports = mongoose.model('movies', Movie)
```

2.1.3 Criando rotas

Aqui, criaremos todas as operações **CRUD** e criaremos nossos pontos de extremidade RESTful. Vamos criar mais duas pastas em *server* são elas: *routes* e *controllers*. Na pasta *routes*, vamos criar o arquivo *movieRouter.js* e na pasta *controller*, *movieController.js*.

movieRouter.js

```
const express = require('express')  
  
const MovieCtrl = require('../controllers/movieController')  
  
const router = express.Router()  
  
router.post('/movie', MovieCtrl.createMovie)  
router.put('/movie/:id', MovieCtrl.updateMovie)  
router.delete('/movie/:id', MovieCtrl.deleteMovie)  
router.get('/movie/:id', MovieCtrl.getMovieById)  
router.get('/movies', MovieCtrl.getMovies)  
  
module.exports = router
```

movieController.js

```
const Movie = require('../models/movie')
```

```
createMovie = (req, res) => {
```

```
  const body = req.body
```

```
  if (!body) {
```

```
    return res.status(400).json({
```

```
      success: false,
```

```
      error: 'Você deve fornecer um filme',
```

```
    })
```

```
  }
```

```
  const movie = new Movie(body)
```

```
  if (!movie) {
```

```
    return res.status(400).json({ success: false, error: err })
```

```
  }
```

```
  movie
```

```
    .save()
```

```
    .then(() => {
```

```
      return res.status(201).json({
```

```
        success: true,
```

```
        id: movie._id,
```

```
        message: 'Filme criado!',
```

```
      })
```

```
    })
```

```
    .catch(error => {
```

```
      return res.status(400).json({
```

```
        error,
```

```
        message: 'Filme não criado!',
```

```
      })
```

```
    })
```

```
  }
```

```
updateMovie = async (req, res) => {
```

```
  const body = req.body
```

```

if (!body) {
  return res.status(400).json({
    success: false,
    error: 'Você deve fornecer um corpo para atualizar',
  })
}

Movie.findOne({ _id: req.params.id }, (err, movie) => {
  if (err) {
    return res.status(404).json({
      err,
      message: 'Filme não encontrado!',
    })
  }
  movie.name = body.name
  movie.time = body.time
  movie.rating = body.rating
  movie
    .save()
    .then(() => {
      return res.status(200).json({
        success: true,
        id: movie._id,
        message: 'Filme atualizado!',
      })
    })
    .catch(error => {
      return res.status(404).json({
        error,
        message: 'Filme não atualizado!',
      })
    })
})
}

deleteMovie = async (req, res) => {
  await Movie.findOneAndDelete({ _id: req.params.id }, (err, movie) => {
    if (err) {
      return res.status(400).json({ success: false, error: err })
    }
  })
}

```

```

    if (!movie) {
      return res
        .status(404)
        .json({ success: false, error: `Filme não encontrado` })
    }

    return res.status(200).json({ success: true, data: movie })
  }).catch(err => console.log(err))
}

```

```

getMovieById = async (req, res) => {
  await Movie.findOne({ _id: req.params.id }, (err, movie) => {
    if (err) {
      return res.status(400).json({ success: false, error: err })
    }

    return res.status(200).json({ success: true, data: movie })
  }).catch(err => console.log(err))
}

```

```

getMovies = async (req, res) => {
  await Movie.find({}, (err, movies) => {
    if (err) {
      return res.status(400).json({ success: false, error: err })
    }
    if (!movies.length) {
      return res
        .status(404)
        .json({ success: false, error: `Filme não encontrado` })
    }
    return res.status(200).json({ success: true, data: movies })
  }).catch(err => console.log(err))
}

```

```

module.exports = {
  createMovie,
  updateMovie,
  deleteMovie,
  getMovies,
}

```

```
    getMovieById,  
  }  
}
```

Por fim, vamos adicionar uma rota em nosso arquivo **server/index.js**.

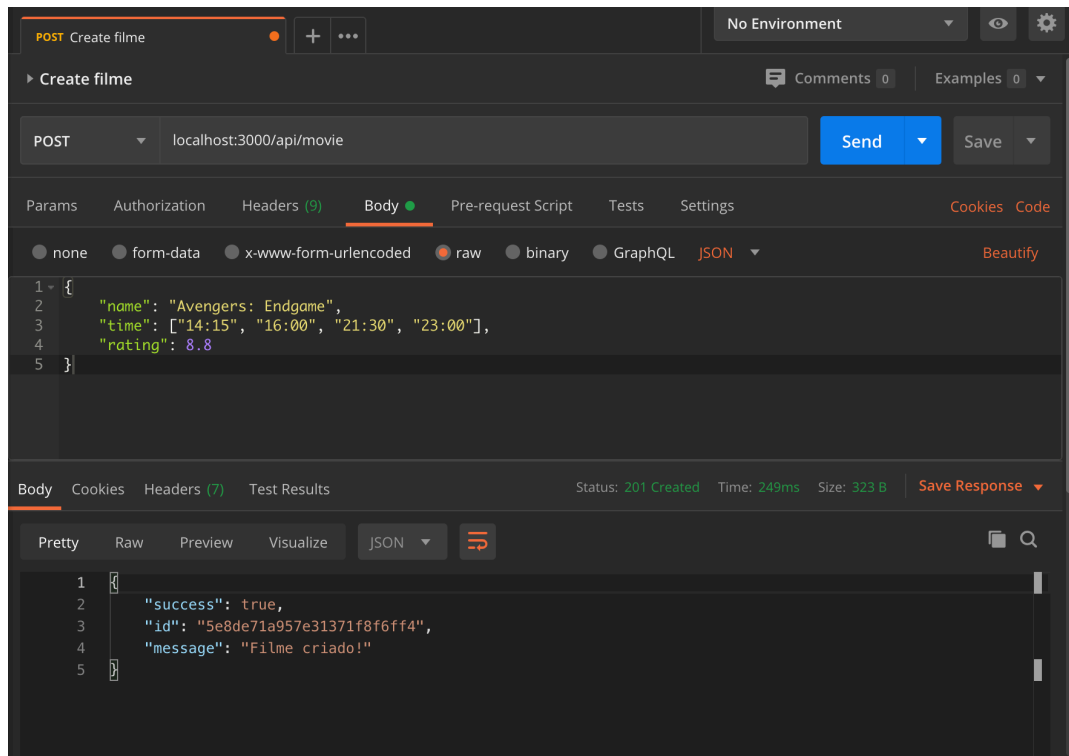
```
const app = express()  
const apiPort = 3000  
  
app.use(bodyParser.urlencoded({ extended: true })))  
app.use(cors())  
app.use(bodyParser.json())  
  
db.on('error', console.error.bind(console, 'MongoDB connection error:'))  
  
app.get('/', (req, res) => {  
  res.send('Hello World!')  
})  
  
app.use('/api', movieRouter)  
  
app.listen(apiPort, () => console.log('Server running on port 3000'));
```

2.1.4 Testando manualmente nosso backend

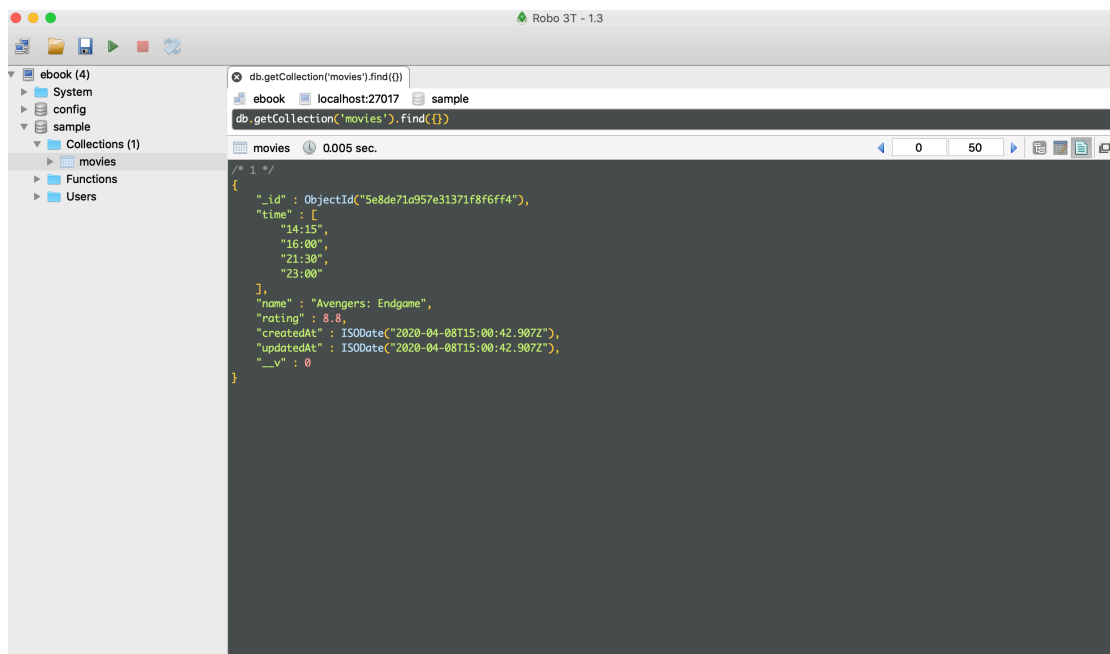
Para testar nosso aplicativo, gostaria de apresentar mais duas ferramentas: [Postman](#) e [Robo 3T](#). Essas ferramentas nos ajudarão a verificar o funcionamento do aplicativo (não explicarei como instalar porque é fácil e você pode encontrar isso documentação dos próprios sites).

Por enquanto, vamos testar adicionar um novo filme. Primeiro, devemos criar um arquivo JSON contendo as informações do filme. Para isso, precisamos inserir essas informações no banco de dados.

Vamos ao Postman e usaremos a operação **POST**.



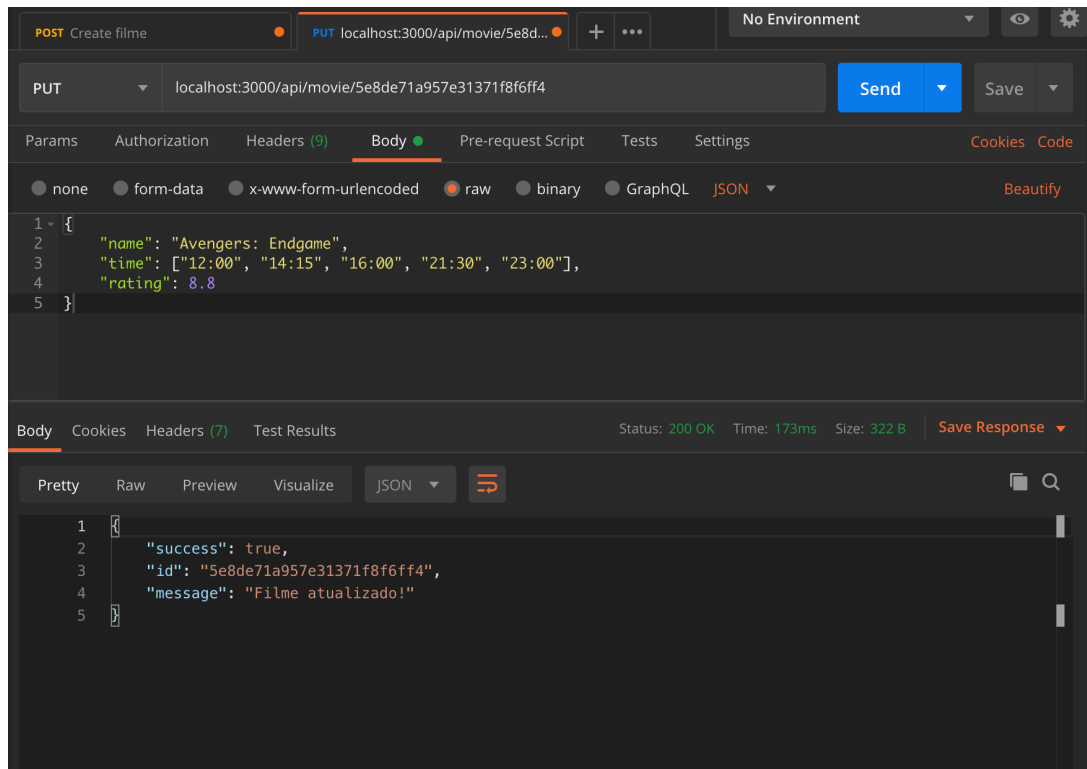
Agora vamos consultar o dado inserido pelo Robo 3T.



Imagine que o sucesso do filme foi tão grande que o cinema decidiu adicionar mais uma sessão. Temos que atualizar isso.

Usando a operação **PUT**, mas desta vez, fornecendo o **ID** do filme.

```
{
  "name": "Avengers: Endgame",
  "time": ["12:00", "14:15", "16:00", "21:30", "23:00"],
  "rating": 8.8
}
```



Vamos adicionar mais dois filmes.

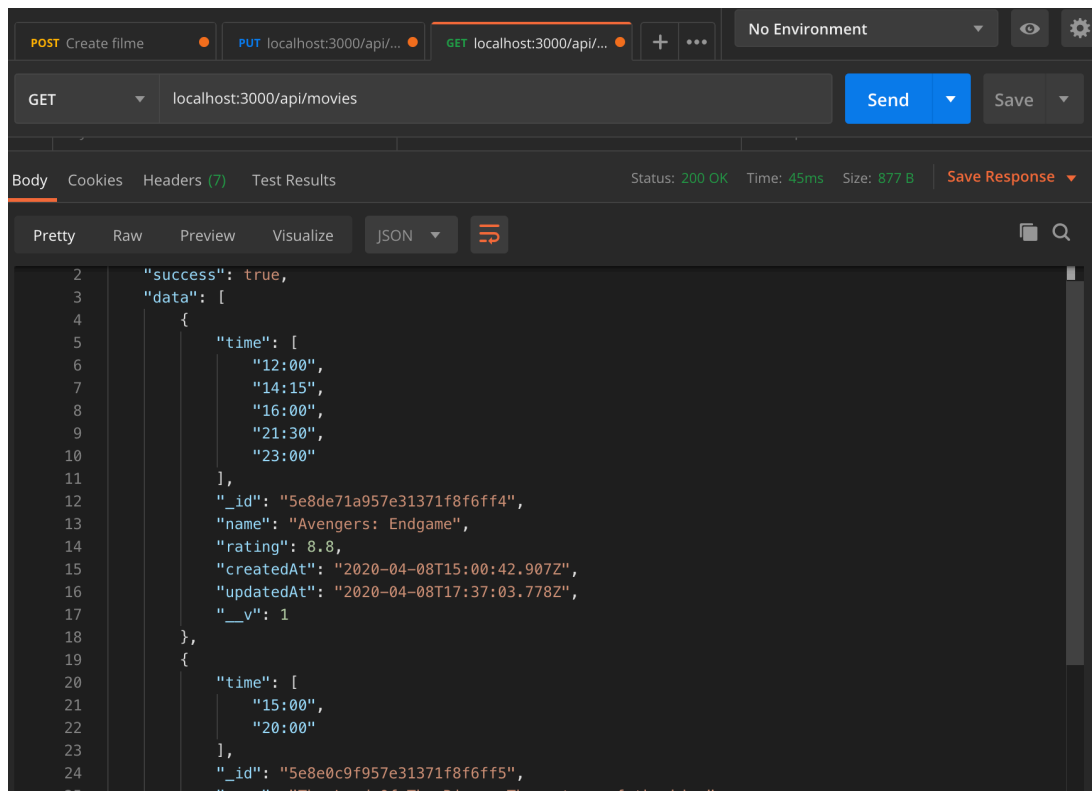
```
{
  "name": "The Lord Of The Rings: The return of the king",
  "time": ["15:00", "20:00"],
  "rating": 8.9
}
```

e

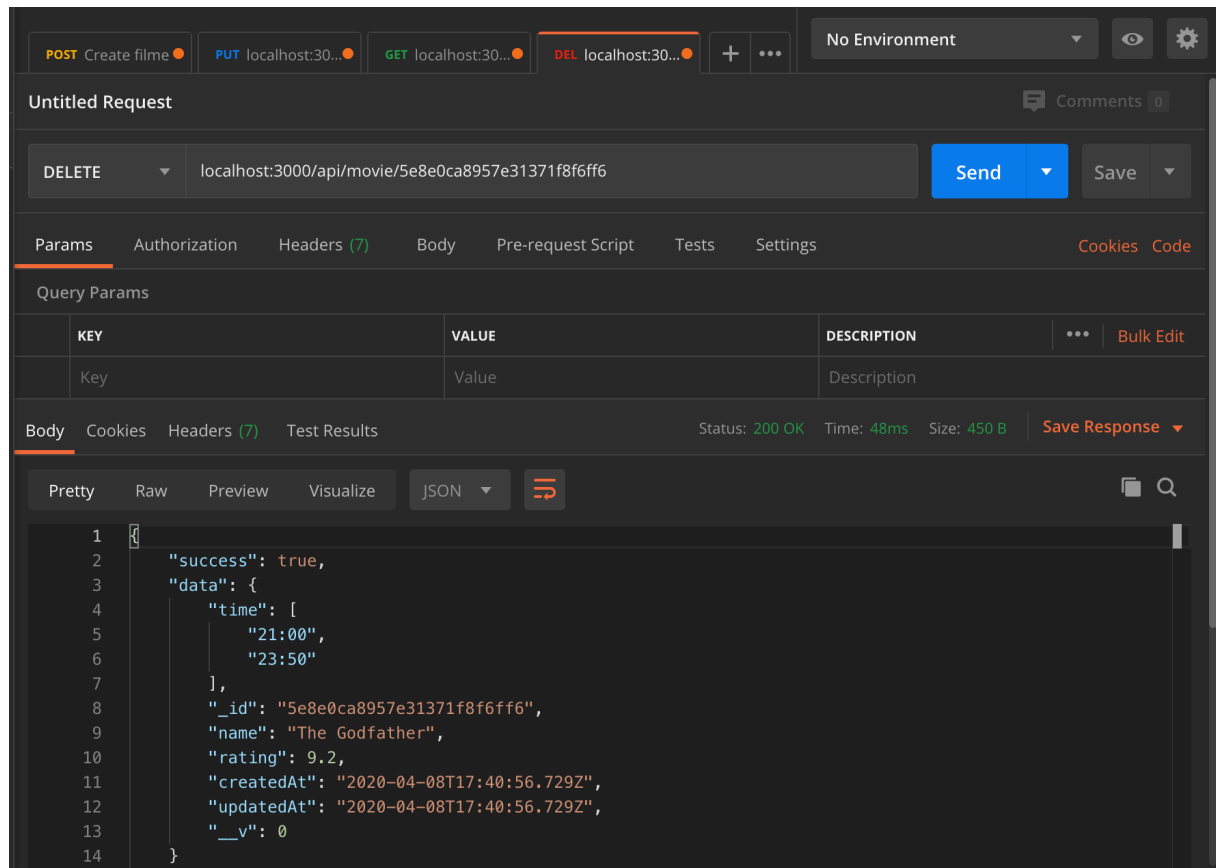
```
{
  "name": "The Godfather",
  "time": ["21:00", "23:50"],
  "rating": 9.2
}
```

}

Podemos agora consultar todos os filmes com a operação **GET**.



E para finalizar nosso backend, vamos excluir um filme utilizando a operação **DELETE**.



Terminamos a parte do nosso backend. Com todo esse conhecimento, você pode criar outras entidades, tente criar uma entidade de usuário, talvez um funcionário ou um preço de tabela, use sua imaginação. Vamos agora iniciar a configuração do nosso frontend.

2.2 Configurando o frontend

Aqui vamos criar toda a parte visual, onde o usuário irá interagir com a nossa ferramenta.

A primeira coisa que devemos fazer é ir para a raiz do nosso projeto e criar o lado do cliente. Para isso, precisamos entender um pouco sobre **NPX**.

O NPX é uma ferramenta cujo objetivo é ajudar a complementar a experiência do uso de pacotes do registro do NPM.

Como o NPM facilita a instalação e o gerenciamento de dependências hospedadas no registro, o NPX facilita o uso de ferramentas de **CLI** e outros executáveis hospedados no registro.

Agora sabendo disso, vamos criar nosso diretório de cliente pra nosso projeto. Como já temos o diretório *server* para o backend, vamos criar o diretório *client* para o lado do frontend.

2.2.1 Criando nosso projeto

Voltamos para a raiz do projeto e executamos os comandos abaixo:

```
$ npm install -g create-react-app  
$ npx create-react-app client  
$ cd client
```

A porta padrão que vem definida é a de número **3000**, mas já definimos essa porta para o nosso back-end. Então, vamos mudá-la para **4000**. Vamos alterar o arquivo *client/package.json* incluindo a variável **PORT**.

```

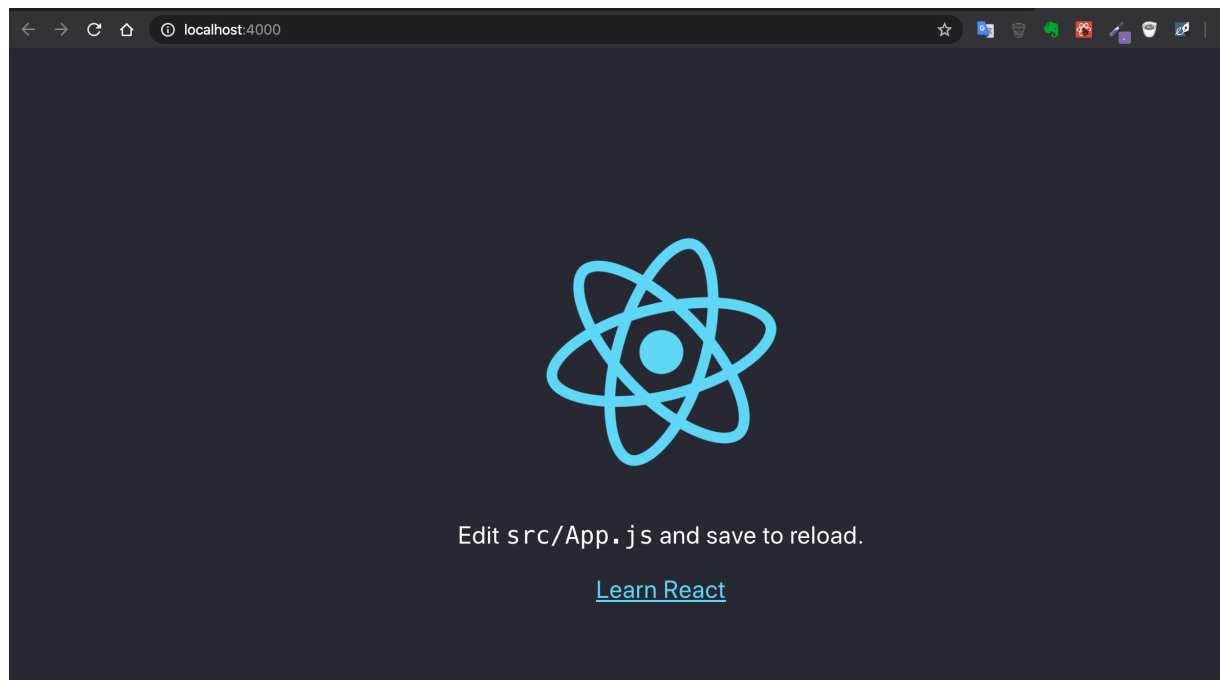
client > {} package.json > {} scripts
1  {
2    "name": "client",
3    "version": "0.1.0",
4    "private": true,
5    "dependencies": {
6      "@testing-library/jest-dom": "^4.2.4",
7      "@testing-library/react": "^9.3.2",
8      "@testing-library/user-event": "^7.1.2",
9      "react": "^16.13.1",
10     "react-dom": "^16.13.1",
11     "react-scripts": "3.4.1"
12   },
13   "scripts": {
14     "start": "PORT=4000 react-scripts start",
15     "build": "react-scripts build",
16     "test": "react-scripts test",
17     "eject": "react-scripts eject"
18   },
19   "eslintConfig": {
20     "extends": "react-app"
21   },
22   "browserslist": {
23     "production": [
24       ">0.2%",
25       "not dead",
26       "not op_mini all"
27     ],
28     "development": [
29       "last 1 chrome version",
30       "last 1 firefox version",
31       "last 1 safari version"
32     ]
33   }
34 }

```

Agora podemos iniciar nossa aplicação.

\$ yarn start

E veremos



Boa! Provavelmente você descobriu que o React cria alguns arquivos padrões. Vamos remover os arquivos desnecessários para nós.

```
$ cd src/  
$ rm App.css index.css App.test.js serviceWorker.js
```

2.2.2 Instalando as dependências e ajustando o projeto

Precisamos configurar nosso projeto instalando as dependências. Precisamos do **Axios**, **Bootstrap**, **StyledComponents** e **React Table**.

- **axios**: é uma lib para consulta baseadas em código assíncrono. É o HTTP mais popular baseado em **promise**.
- **bootstrap**: é um kit de ferramentas de código aberto e a biblioteca de componentes front-end mais popular, onde você pode desenvolver HTML, CSS e JS.
- **styled-components**: Ele permite que você escreva um código CSS real para estilizar seus componentes.
- **react-table**: É uma grade de dados leve, rápida e extensível criada para o React.
- **react-router-dom**: DOM bindings para React Routers.

```
$ yarn add styled-components react-bootstrap-table react-router-dom axios bootstrap
```

No diretório *src*, devemos criar os novos diretórios que serão a estrutura do nosso projeto. Crie um arquivo **index.js** dentro de cada diretório, exceto na *app*.

```
$ cd src  
$ mkdir api app components pages style  
$ touch api/index.js components/index.js pages/index.js style/index.js
```

Mova o arquivo *App.js* para o diretório *app*, mas renomeie para *index.js*.

3 Codificando

Primeiro de tudo, atualize nosso arquivo *client/src/index.js* para o seguinte código.

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './app'
```

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Depois disso, desenvolveremos o cabeçalho do aplicativo. Agora, vamos desenvolver os componentes do nosso projeto. Crie os novos arquivos *NavBar.jsx*, *Logo.jsx* e *Links.jsx*.

```
$ touch components/NavBar.jsx components/Logo.jsx components/Links.jsx
```

JSX é uma notação que o React escolheu para identificar a extensão de um JavaScript. É recomendável usá-lo com o React para descrever a aparência da interface do usuário.

Sabendo disso, vamos criar nossos arquivos.

Logo.jsx

```
import React, { Component } from 'react'
import styled from 'styled-components'
```

```
import logo from './logo.svg'
```

```
const Wrapper = styled.a.attrs({
  className: 'navbar-brand',
})
```

```
class Logo extends Component {
```

```

render() {
  return (
    <Wrapper href="https://dev.delivery">
      <img src={logo} width="50" height="50" alt="devDelivery" />
    </Wrapper>
  )
}
}

```

```
export default Logo
```

Links.jsx

```

import React, { Component } from 'react'
import { Link } from 'react-router-dom'
import styled from 'styled-components'

```

```

const Collapse = styled.div.attrs({
  className: 'collpase navbar-collapse',
})`

```

```

const List = styled.div.attrs({
  className: 'navbar-nav mr-auto',
})`

```

```

const Item = styled.div.attrs({
  className: 'collpase navbar-collapse',
})`

```

```

class Links extends Component {
  render() {
    return (
      <React.Fragment>
        <Link to="/" className="navbar-brand">
          MERN Sample
        </Link>
        <Collapse>
          <List>
            <Item>

```

```

        <Link to="/movies/list" className="nav- link">
            Movies
        </Link>
    </Item>
</List>
</Collapse>
</React.Fragment>
    )
}
}
export default Links

```

NavBar.jsx

```

import React, { Component } from 'react'
import styled from 'styled-components'

```

```

import Logo from './Logo'
import Links from './Links'

```

```

const Container = styled.div.attrs({
  className: 'container',
})`

```

```

const Nav = styled.nav.attrs({
  className: 'navbar navbar-expand-lg navbar-dark bg-dark',
})`
  margin-bottom: 20 px;
`

```

```

class NavBar extends Component {
  render() {
    return (
      <Container>
        <Nav>
          <Logo />
          <Links />
        </Nav>
      </Container>
    )
  }
}

```

```
}
```

```
export default NavBar
```

Você pode ver que os arquivos que criamos estão vazios. Você se lembra que criamos o arquivo ***components/index.js***?

Este arquivo que exportamos nossos componentes, nos permitirá importá-los em outros arquivos usando a notação:

```
import { xxxxx } from './components'
```

Veja como é simples.

components/index.js

```
import Links from './Links'  
import Logo from './Logo'  
import NavBar from './NavBar'
```

```
export { Links, Logo, NavBar }
```

Atualizando ***app/index.js***. Podemos já ver o crescimento do projeto.

```
import React from 'react'  
import { BrowserRouter as Router } from 'react-router-dom'
```

```
import { NavBar } from '../components'
```

```
import 'bootstrap/dist/css/bootstrap.min.css'
```

```
function App() {  
  return (  
    <Router>
```



```

    <NavBar />
  </Router>
)
}

```

```
export default App
```



3.1 Integrando backend e frontend

Agora vamos aprender como integrar o back-end com o nosso frontend. Primeiro de tudo, vamos atualizar o arquivo `api/index.js`.

```
import axios from 'axios'
```

```
const api = axios.create({
  baseURL: 'http://localhost:3000/api',
})
```

```
export const insertMovie = payload => api.post('/movie', payload)
```

```
export const getAllMovies = () => api.get('/movies')
export const updateMovieById = (id, payload) => api.put(`/movie/${id}`, payload)
export const deleteMovieById = id => api.delete(`/movie/${id}`)
export const getMovieById = id => api.get(`/movie/${id}`)
```

```
const apis = {
  insertMovie,
  getAllMovies,
  updateMovieById,
  deleteMovieById,
  getMovieById,
}
```

```
export default apis
```

Agora podemos desenvolver nossas rotas. Para isso, precisamos atualizar o arquivo *app/index.js* adicionando as rotas necessárias.

```
import React from 'react'
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom'
```

```
import { NavBar } from '../components'
import { MoviesList, MoviesInsert, MoviesUpdate } from '../pages'
```

```
import 'bootstrap/dist/css/bootstrap.min.css'
```

```
function App() {
  return (
    <Router>
      <NavBar />
      <Switch>
        <Route path="/movies/list" exact component={Movies} />
      </Switch>
    </Router>
  )
}
```

```
export default App
```

Na diretório *pages*, vamos criar os arquivos que farão o papel da página de cada aplicativo. *Movies.jsx* .

```
$ cd pages
$ touch Movies.jsx
```

Por enquanto, vamos criar arquivos simples onde você pode ver a transição da página quando clicar em cada link da *NavBar*.

Movies.jsx

```
import React, { Component } from 'react';

class MoviesList extends Component {
  render() {
    return (
      <div>
        <p>Nesta página, você verá a lista de filmes</p>
      </div>
    )
  }
}
```

```
export default MoviesList
```

Atualizamos agora o arquivo */pages/index.js*.

```
import Movies from './Movies'

export { Movies}
```

Vamos agora editar nosso arquivo para obter os filmes do banco de dados:

Para tal vamos usar a lib **react-bootstrap-table**, podemos ver todos os detalhes e exemplos de implementação em sua [documentação](#).

Neste momento vamos apenas criar a nossa *table*.

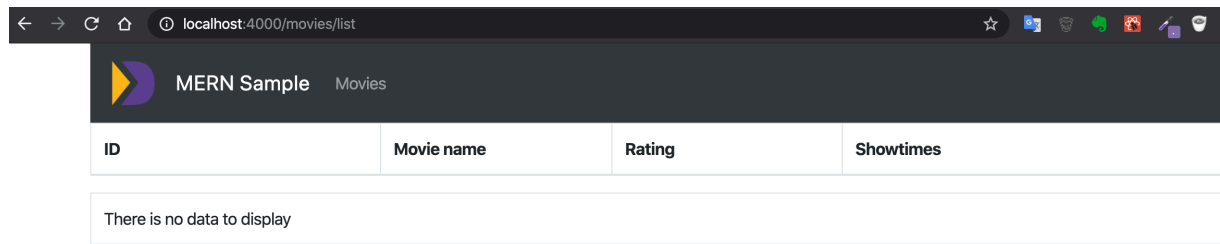
Movies.jsx.

```
import React, { Component } from "react";
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";
import styled from "styled-components";
```

```
const Container = styled.div.attrs({
  className: "container",
})``;
```

```
class Movies extends Component {
  render() {
    return (
      <div>
        <Container>
          <BootstrapTable>
            <TableHeaderColumn dataField="_id" isKey={true} width={"25%"}>
              ID
            </TableHeaderColumn>
            <TableHeaderColumn dataField="name" width={"20%"}>
              Movie name
            </TableHeaderColumn>
            <TableHeaderColumn dataField="rating" width={"20%"}>
              Rating
            </TableHeaderColumn>
            <TableHeaderColumn dataField="time" width={"40%"}>
              Showtimes
            </TableHeaderColumn>
          </BootstrapTable>
        </Container>
      </div>
    );
  }
}
```

```
export default Movies;
```



3.2 Implementando a consulta de filmes

Agora vamos inserir a chamada para nosso serviço de busca de filmes do nosso backend.

Movies.jsx.

```
import React, { Component } from "react";
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";
import styled from "styled-components";
import api from "../api";
```

```
const Container = styled.div.attrs({
  className: "container",
});
```

```
class Movies extends Component {
  constructor(props) {
    super(props);
```

```

    this.state = {
      movies: [],
    };
  }

  componentDidMount = async () => {
    await api.getAllMovies().then((movies) => {
      this.setState({
        movies: movies.data.data,
      });
    });
  };

  render() {
    const { movies } = this.state;

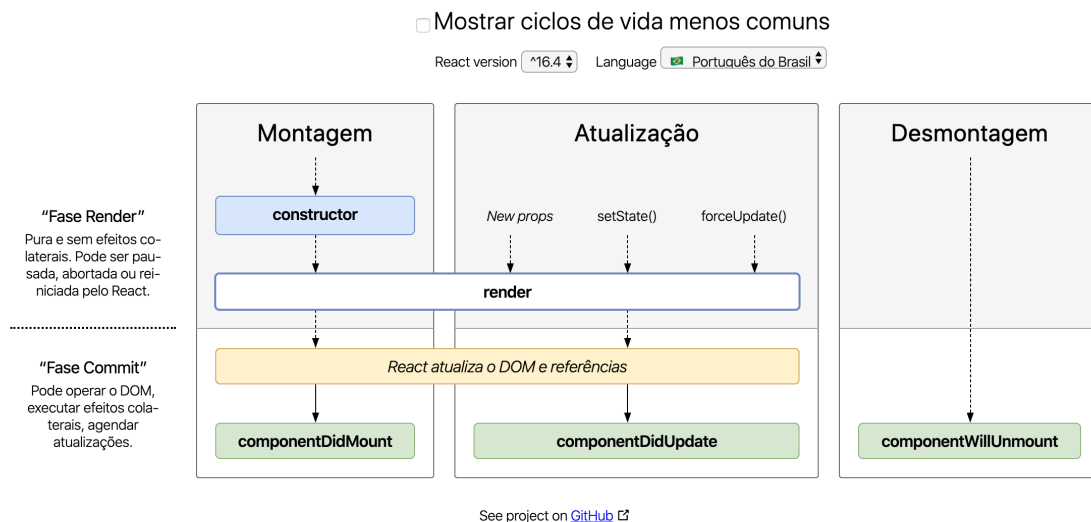
    return (
      <div>
        <Container>
          <BootstrapTable data={movies}>
            <TableHeaderColumn dataField="_id" isKey={true} width={"25%"}>
              ID
            </TableHeaderColumn>
            <TableHeaderColumn dataField="name" width={"20%"}>
              Movie name
            </TableHeaderColumn>
            <TableHeaderColumn dataField="rating" width={"20%"}>
              Rating
            </TableHeaderColumn>
            <TableHeaderColumn dataField="time" width={"40%"}>
              Showtimes
            </TableHeaderColumn>
          </BootstrapTable>
        </Container>
      </div>
    );
  }
}

export default Movies;

```

Agora aqui um ponto de **atenção** perceba que incluímos a função **componentDidMount** em nosso código.

O React possui ciclos de vidas de componentes e disponibiliza funções nativas para executar funções de acordo com cada necessidade conforme imagem abaixo:

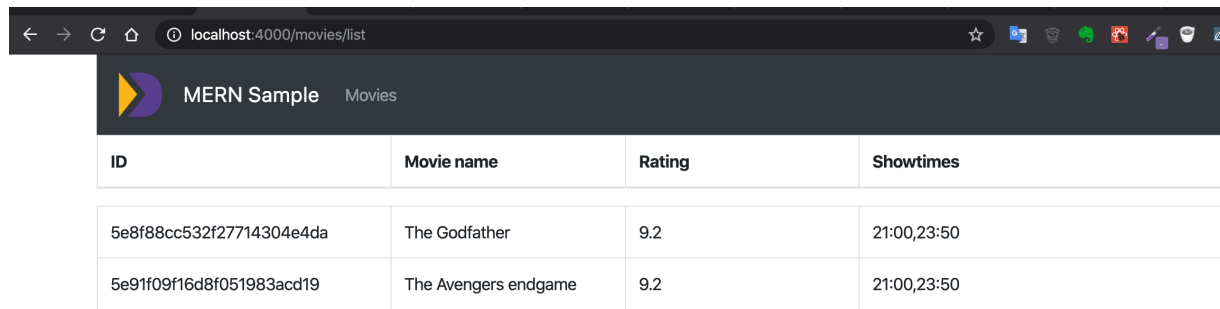


Incluímos nossa chamada ali. Também foi incluído um elemento chamado **await**. O React foi projetado para dar ênfase em performance portando explora em muito as chamadas assíncronas. O **await** faz com que a função **componentDidMount** espere o retorno da chamada a nossa **api** para prosseguir.

Armazenamos o retorno da chamada na variável *movies* que será utilizada pelo componente `<BootstrapTable>` através da propriedade

```
data={movies}
```

Podemos ver o resultado!!!



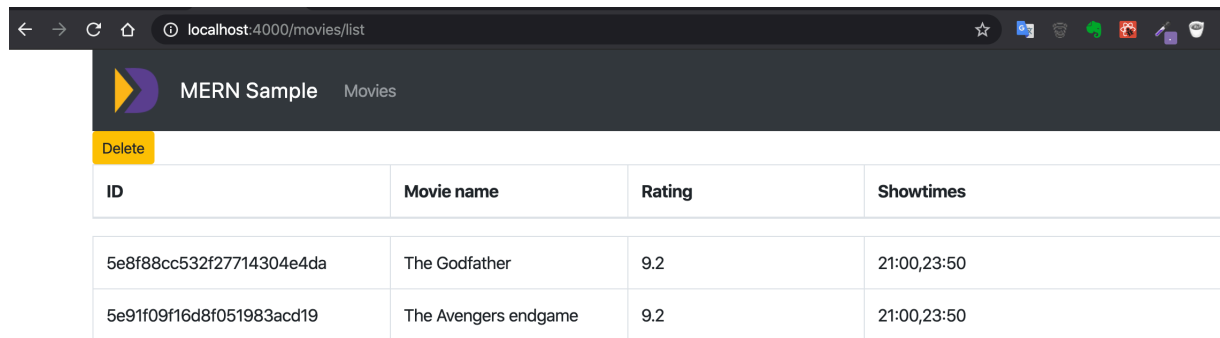
ID	Movie name	Rating	Showtimes
5e8f88cc532f27714304e4da	The Godfather	9.2	21:00,23:50
5e91f09f16d8f051983acd19	The Avengers endgame	9.2	21:00,23:50

3.2 Implementando a exclusão de filmes

Agora vamos incluir a o operação de *DELETE* em nossa página inserindo a propriedade

```
deleteRow={true}
```

No componente `<BootstrapTable>` a lib **react-bootstrap-table** inclui automaticamente um **botão** de *Delete*.



ID	Movie name	Rating	Showtimes
5e8f88cc532f27714304e4da	The Godfather	9.2	21:00,23:50
5e91f09f16d8f051983acd19	The Avengers endgame	9.2	21:00,23:50

Precisamos agora implementar a função para selecionar a linha que queremos excluir. Seguindo a [documentação](#) do **react-bootstrap-table** basta incluir a propriedade no componente `<BootstrapTable>` :

```
selectRow={ selectRowProp }
```

E incluir a *const selectRowProp*

```
const selectRowProp = {
  mode: 'radio'
};
```

Movie.jsx

```
import React, { Component } from "react";
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";
import styled from "styled-components";
import api from "../api";
```

```
const Container = styled.div.attrs({
  className: "container",
```

```
});
```

```
const selectRowProp = {  
  mode: 'radio'  
};
```

```
class Movies extends Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      movies: [],  
    };  
  }  
}
```

```
componentDidMount = async () => {  
  await api.getAllMovies().then((movies) => {  
    this.setState({  
      movies: movies.data.data,  
    });  
  });  
};
```

```
render() {  
  const { movies } = this.state;
```

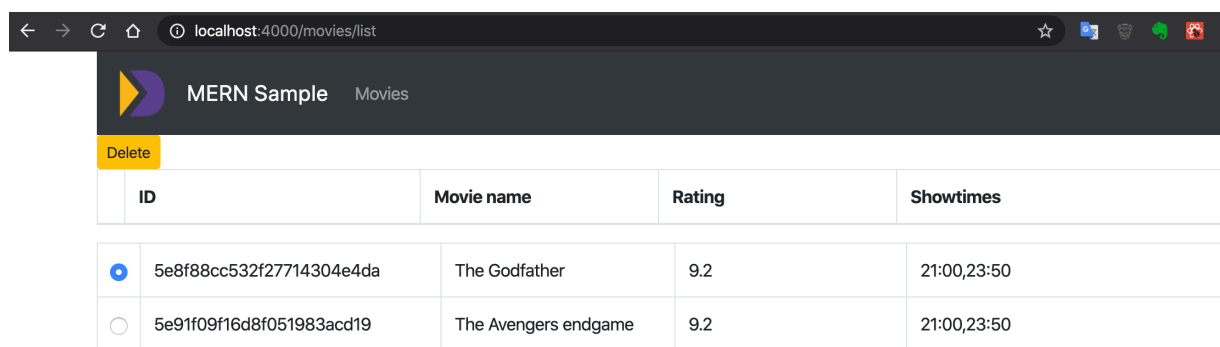
```
  return (  
    <div>  
      <Container>  
        <BootstrapTable data={movies} deleteRow={true} selectRow={selectRowProp}>  
          <TableHeaderColumn dataField="_id" isKey={true} width={"25%"}>  
            ID  
          </TableHeaderColumn>  
          <TableHeaderColumn dataField="name" width={"20%"}>  
            Movie name  
          </TableHeaderColumn>  
          <TableHeaderColumn dataField="rating" width={"20%"}>  
            Rating  
          </TableHeaderColumn>  
          <TableHeaderColumn dataField="time" width={"40%"}>  
            Showtimes  
          </TableHeaderColumn>  
        </BootstrapTable>  
      </Container>  
    </div>  
  );  
}
```

```

        </TableHeaderColumn>
      </BootstrapTable>
    </Container>
  </div>
);
}
}

```

```
export default Movies;
```



	ID	Movie name	Rating	Showtimes
<input checked="" type="radio"/>	5e8f88cc532f27714304e4da	The Godfather	9.2	21:00,23:50
<input type="radio"/>	5e91f09f16d8f051983acd19	The Avengers endgame	9.2	21:00,23:50

Agora que já conseguimos *selecionar* a linha que queremos excluir devemos implementar a chamada para a **api** do nosso backend para realizar a exclusão.

Seguindo a [documentação](#) do **react-bootstrap-table** basta incluir as propriedades no componente `<BootstrapTable>`

```
deleteRow={ true } options={ options }
```

E implementar as funções *options* e *onAfterDeleteRow*

```
const options = {  
  afterDeleteRow: onAfterDeleteRow  
};
```

```
async function onAfterDeleteRow(rowKeys, rows) {  
  alert("The rowkey you drop: " + rowKeys);  
  await api.deleteMovieById(rowKeys);  
}
```

Movie.jsx

```
import React, { Component } from "react";  
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";  
import styled from "styled-components";  
import api from "../api";
```

```
const Container = styled.div.attrs({  
  className: "container",  
})``;
```

```
const selectRowProp = {  
  mode: "radio",  
};
```

```
const options = {  
  afterDeleteRow: onAfterDeleteRow  
};
```

```
async function onAfterDeleteRow(rowKeys, rows) {  
  alert("The rowkey you drop: " + rowKeys);  
  await api.deleteMovieById(rowKeys);  
}
```

```
class Movies extends Component {  
  constructor(props) {  
    super(props);  
    this.state = {
```

```

    movies: [],
  };
}

componentDidMount = async () => {
  await api.getAllMovies().then((movies) => {
    this.setState({
      movies: movies.data.data,
    });
  });
};

render() {
  const { movies } = this.state;

  return (
    <div>
      <Container>
        <BootstrapTable
          data={movies}
          deleteRow={true}
          selectRow={selectRowProp}
          options={options}
        >
          <TableHeaderColumn dataField="_id" isKey={true} width={"25%"}>
            ID
          </TableHeaderColumn>
          <TableHeaderColumn dataField="name" width={"20%"}>
            Movie name
          </TableHeaderColumn>
          <TableHeaderColumn dataField="rating" width={"20%"}>
            Rating
          </TableHeaderColumn>
          <TableHeaderColumn dataField="time" width={"40%"}>
            Showtimes
          </TableHeaderColumn>
        </BootstrapTable>
      </Container>
    </div>
  );
}

```

```
}
```

```
export default Movies;
```

3.3 Implementando a inserção de filmes

Agora vamos incluir a operação de *INSERT*. Seguindo a [documentação](#) do **react-bootstrap-table** basta incluir as propriedades no componente `<BootstrapTable>` a lib **react-bootstrap-table** inclui automaticamente um **botão** de *New* .

```
insertRow={ true } options={ options }
```

E implementar as funções *options* e *onAfterDeleteRow*

```
const options = {  
  afterInsertRow: onAfterInsertRow  
};
```

```
async function onAfterInsertRow(row) {  
  const payload = {  
    name: row["name"],  
    rating: row["rating"],  
    time: row["time"],  
  };  
  
  await api.insertMovie(payload).then((res) => {  
    window.alert('Movie inserted successfully');  
  });  
}
```

Movie.jsx

```
import React, { Component } from "react";  
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";  
import styled from "styled-components";  
import api from "../api";
```

```

const Container = styled.div.attrs({
  className: "container",
})``;

const selectRowProp = {
  mode: "radio",
};

const options = {
  afterDeleteRow: onAfterDeleteRow,
  afterInsertRow: onAfterInsertRow,
};

async function onAfterDeleteRow(rowKeys, rows) {
  alert("The rowkey you drop: " + rowKeys);
  await api.deleteMovieById(rowKeys);
}

async function onAfterInsertRow(row) {
  const payload = {
    name: row["name"],
    rating: row["rating"],
    time: row["time"],
  };

  await api.insertMovie(payload).then((res) => {
    window.alert('Movie inserted successfully');
  });
}

class Movies extends Component {
  constructor(props) {
    super(props);
    this.state = {
      movies: [],
    };
  }

  componentDidMount = async () => {

```

```

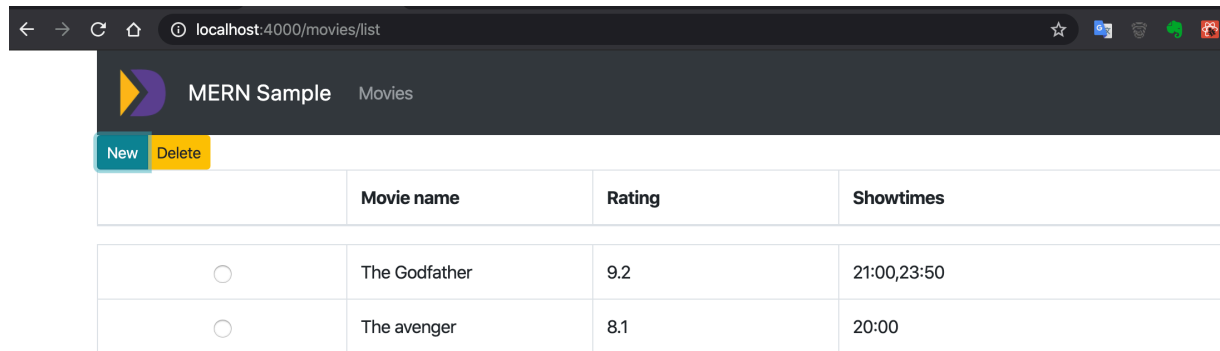
    await api.getAllMovies().then((movies) => {
      this.setState({
        movies: movies.data.data,
      });
    });
  };

  render() {
    const { movies } = this.state;

    return (
      <div>
        <Container>
          <BootstrapTable
            data={movies}
            deleteRow={true}
            selectRow={selectRowProp}
            options={options}
            insertRow={true}
          >
            <TableHeaderColumn dataField="_id" isKey={true} hidden
              autoValue={true} width={"25%"}>
              ID
            </TableHeaderColumn>
            <TableHeaderColumn dataField="name" width={"20%"}>
              Movie name
            </TableHeaderColumn>
            <TableHeaderColumn dataField="rating" width={"20%"}>
              Rating
            </TableHeaderColumn>
            <TableHeaderColumn dataField="time" width={"40%"}>
              Showtimes
            </TableHeaderColumn>
          </BootstrapTable>
        </Container>
      </div>
    );
  }
}

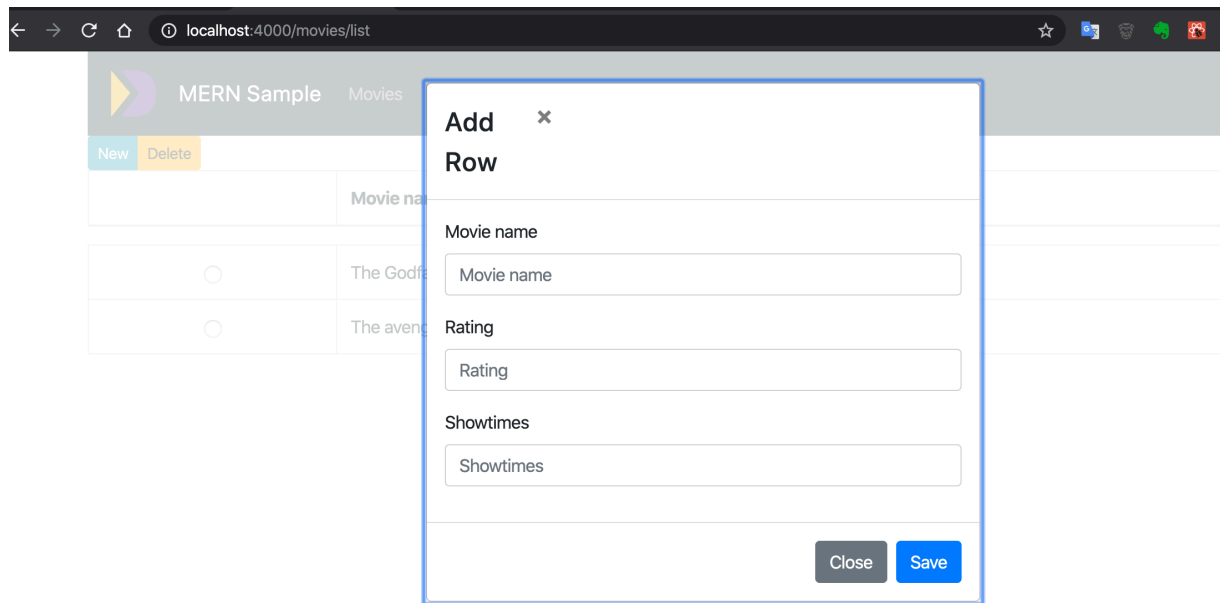
export default Movies;

```

	Movie name	Rating	Showtimes
<input type="radio"/>	The Godfather	9.2	21:00,23:50
<input type="radio"/>	The avenger	8.1	20:00

A lib **react-bootstrap-table** inclui automaticamente uma modal para inserir novos dados.



3.3 Implementando a edição de filmes

Agora vamos incluir a operação de *EDIT*. Seguindo a [documentação](#) do **react-bootstrap-table** basta incluir as propriedades no componente `<BootstrapTable>` a lib **react-bootstrap-table**.

```
cellEdit={cellEditProp}
```

E implementar as funções `cellEditProp` e `onAfterInsertRow`

```
const cellEditProp = {  
  mode: "click",  
  blurToSave: true,  
  afterSaveCell: onAfterSaveCell,  
};
```

```
async function onAfterSaveCell(row, cellName, cellValue) {  
  alert(`Save cell ${cellName} with value ${cellValue}`);  
}
```

```

let id = row["_id"];

const payload = {
  name: row["name"],
  rating: row["rating"],
  time: row["time"],
};

await api.updateMovieById(id, payload).then((res) => {
  window.alert('Movie updated successfully');
});
}

```

Movies.jsx

```

import React, { Component } from "react";
import { BootstrapTable, TableHeaderColumn } from "react-bootstrap-table";
import styled from "styled-components";
import api from "../api";

const Container = styled.div.attrs({
  className: "container",
})``;

const selectRowProp = {
  mode: "radio",
};

const options = {
  afterDeleteRow: onAfterDeleteRow,
  afterInsertRow: onAfterInsertRow,
};

const cellEditProp = {
  mode: "click",
  blurToSave: true,
  afterSaveCell: onAfterSaveCell,
};

```

```

async function onAfterDeleteRow(rowKeys, rows) {
  alert("The rowkey you drop: " + rowKeys);
  await api.deleteMovieById(rowKeys);
}

async function onAfterInsertRow(row) {
  const payload = {
    name: row["name"],
    rating: row["rating"],
    time: row["time"],
  };

  await api.insertMovie(payload).then((res) => {
    window.alert('Movie inserted successfully');
  });
}

async function onAfterSaveCell(row, cellName, cellValue) {
  alert('Save cell ${cellName} with value ${cellValue}');

  let id = row["_id"];

  const payload = {
    name: row["name"],
    rating: row["rating"],
    time: row["time"],
  };

  await api.updateMovieById(id, payload).then((res) => {
    window.alert('Movie updated successfully');
  });
}

class Movies extends Component {
  constructor(props) {
    super(props);
    this.state = {
      movies: [],
    };
  }
  componentDidMount = async () => {

```

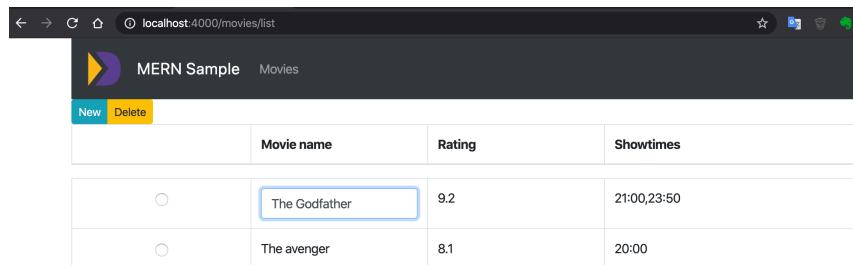
```

await api.getAllMovies().then((movies) => {
  this.setState({
    movies: movies.data.data,
  });
});

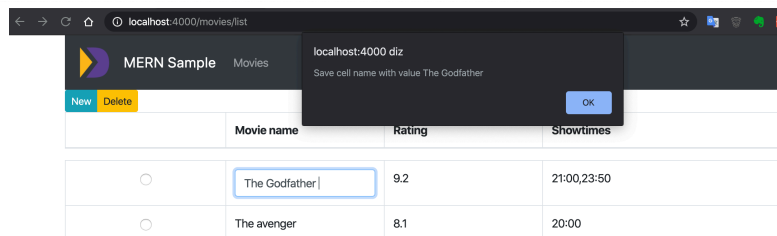
render() {
  const { movies } = this.state;

  return (
    <div>
      <Container>
        <BootstrapTable
          data={movies}
          deleteRow={true}
          selectRow={selectRowProp}
          options={options}
          insertRow={true}
          cellEdit={cellEditProp}
        >
          <TableHeaderColumn dataField="_id" isKey={true} hidden
            autoValue={true} width={"25%"}>
            ID
          </TableHeaderColumn>
          <TableHeaderColumn dataField="name" width={"20%"}>
            Movie name(img)
          </TableHeaderColumn>
          <TableHeaderColumn dataField="rating" width={"20%"}>
            Rating
          </TableHeaderColumn>
          <TableHeaderColumn dataField="time" width={"40%"}>
            Showtimes
          </TableHeaderColumn>
        </BootstrapTable>
      </Container>
    </div>
  );
}
}
export default Movies;

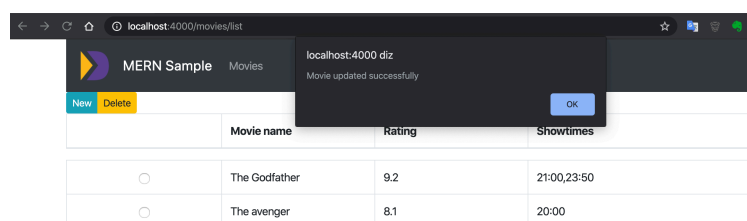
```



	Movie name	Rating	Showtimes
<input type="radio"/>	The Godfather	9.2	21:00,23:50
<input type="radio"/>	The avenger	8.1	20:00



	Movie name	Rating	Showtimes
<input type="radio"/>	The Godfather	9.2	21:00,23:50
<input type="radio"/>	The avenger	8.1	20:00



	Movie name	Rating	Showtimes
<input type="radio"/>	The Godfather	9.2	21:00,23:50
<input type="radio"/>	The avenger	8.1	20:00

E finalizamos nosso CRUD.

4 Conclusão

Que legal você ter chegado ao final desta leitura, estou feliz por você ter cumprido mais essa etapa na sua carreira.

Espero que tenha colocado em prática tudo que aprendeu. Não se contente em apenas ler esse livro. Pratique, programe, implemente cada detalhe, caso contrário, em algumas semanas já terá esquecido grande parte do conteúdo.

Se você gostou desse livro, por favor, ajude a manter esse trabalho. Recomende para seus amigos de trabalho, faculdade e/ou compartilhe no Facebook e Twitter.

E é isso. Neste livro, você viu a estrutura para criar um aplicativo **MERN** usando diversas bibliotecas fornecidas por uma comunidade incrível em todo o mundo.

Meu intuito foi manter as coisas simples para sua compreensão.

Este é um projeto que você pode fazer muitos aprimoramentos. Talvez você possa tentar, por exemplo, criar uma estrutura melhor para incluir a hora dos filmes, que tal criar uma nova entidade chamada customer e criar mais uma operação RESTful para isso? Você pode adicionar muitos novos recursos.

4.1 Próximos passos

Embora esse livro tenha te ajudado a criar uma aplicação do início ao fim com as tecnologias (M)ongo, (E)xpress, (R)eact e (N)ode de nível básico o que você aprendeu nele é só a ponta do iceberg!

É claro que você não perdeu tempo com o que acabou de estudar, o que eu quero dizer é que há muito mais coisas para se aprofundar.

Caso você tenha interesse em continuar seu aprendizado, recomendo que você faça parte da nossa comunidade e aprimore seus conhecimentos.