Javascript Node

Usando Sequelize ORM com Node e Express

Vamos aprender a configurar uma aplicação com banco de dados MySQL usando Sequelize ORM com Node.js e Express.

Wesley Gado • há 2 anos 5 meses

e o mapeamento dos dois modelos quase que eliminando a necessidade de ter que utilizar alguma query com código SQL.

Você pode acessar nosso artigo "O que é ORM?" aqui da TreinaWeb para se aprofundar no assunto.

Agora vamos aprender a utilizar Sequelize ORM, um dos mais conhecidos se tratando de Node.js, criando uma aplicação para realizar a comunicação com o banco de dados e também manipular os registros. Você pode acessar os arquivos neste repositório do GitHub.

Configurando ambiente Node.js e Express

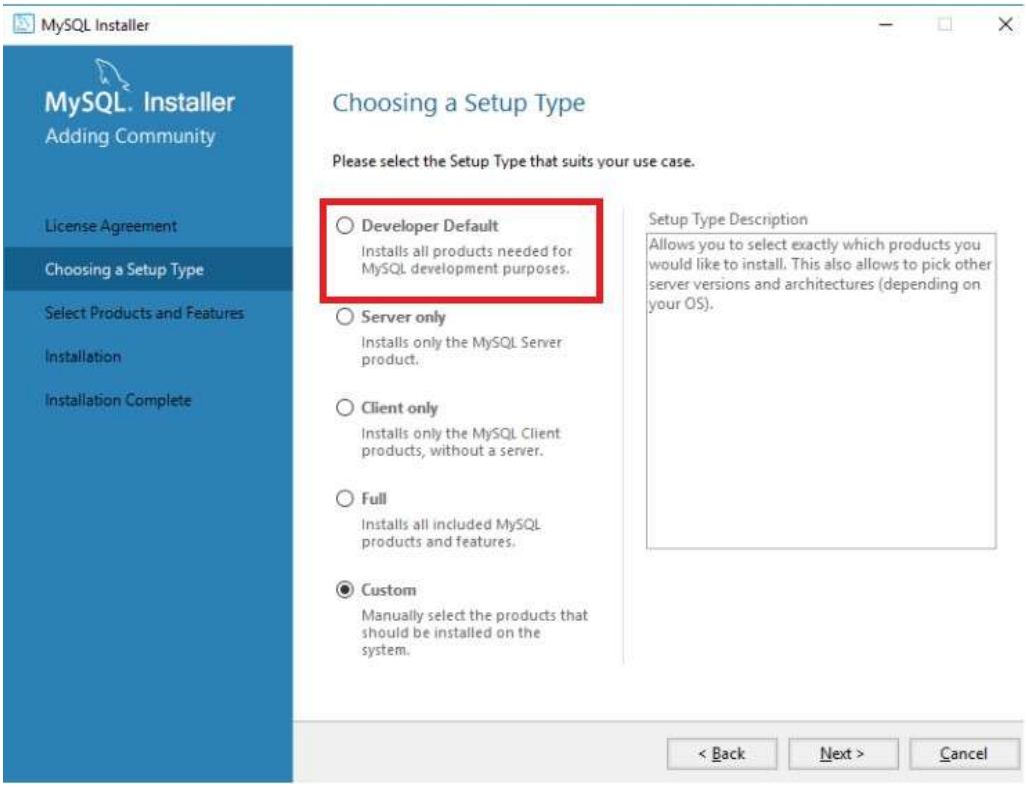
Caso você não tenha conhecimento com Node.js e Express, aconselho seguir os artigos abaixo para instalar o Node.js e configurar o ambiente que vamos utilizar neste artigo:

- Instalação do Node.js - Windows, Mac e Linux;
- Instalando e criando um servidor com Express;
- Criando rotas com Express.

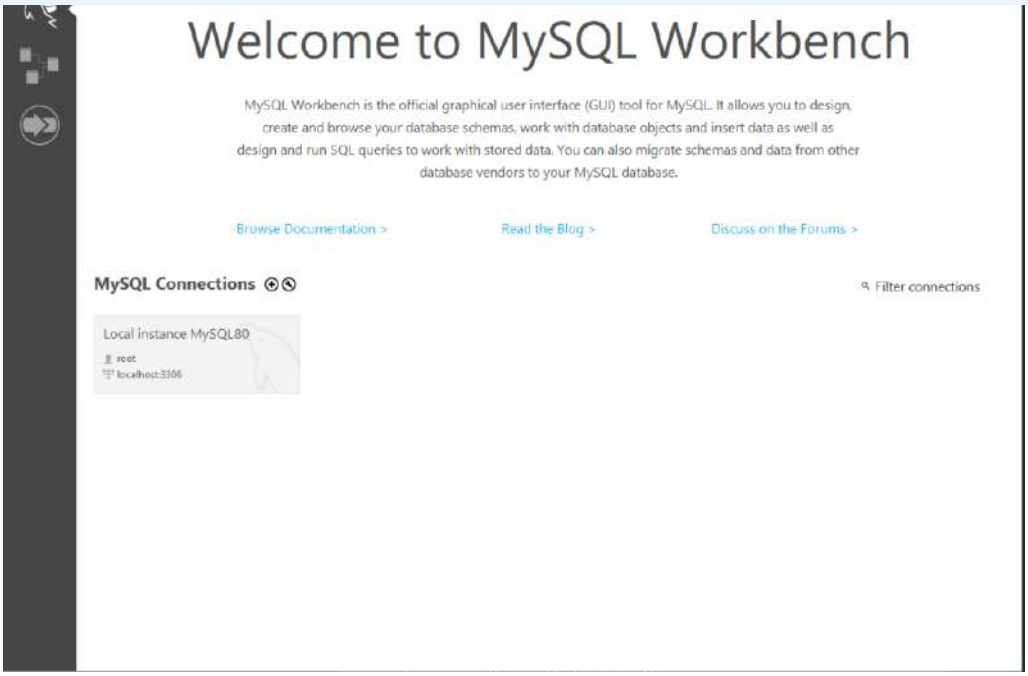
Estes artigos possuem a base do que precisamos para seguir em frente, portanto, vamos efetivamente instalar o Sequelize ORM e entender o seu funcionamento.

Neste artigo vamos utilizar o MySQL, para acessar o MySQL normalmente utilizamos clientes gráficos, portanto, eu particularmente indico o download o MySQL Workbench community, que você pode efetuar o download gratuitamente.

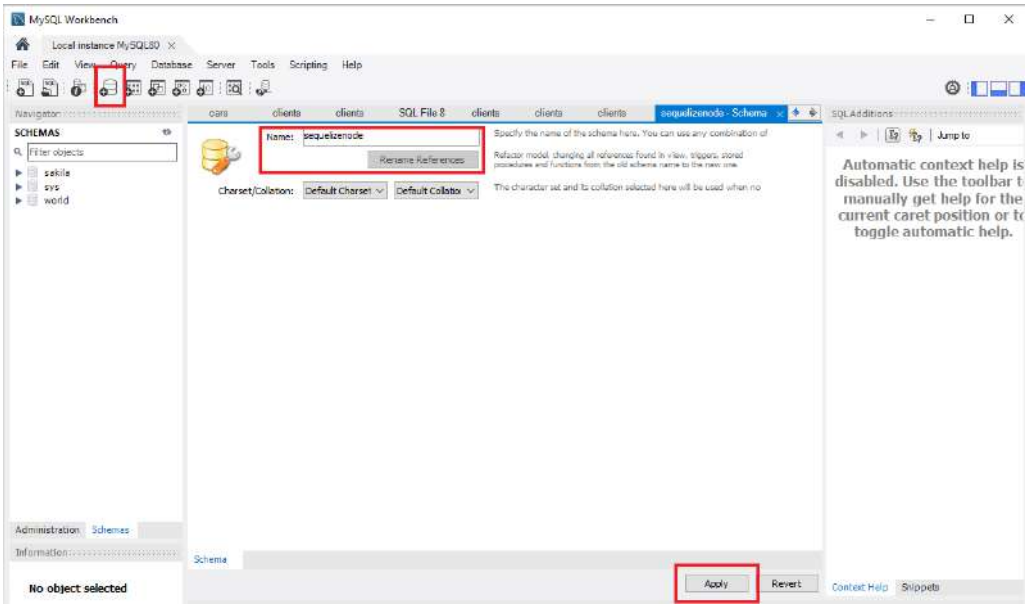
Após o download, basta abrir o executável. Logo no início irá surgir a tela de termos uso, aceite e continue a instalação, até aparecer a seguinte informação:



Neste momento temos algumas opções de instalação. A opção custom, onde você escolhe item a item das aplicações que serão instaladas, e conforme a imagem acima em destaque temos a opção de desenvolvedor, onde será instalado o MySQL server e o cliente MySQL Workbench, vamos selecionar esta opção e seguir com a instalação.



Neste momento precisamos criar o banco de dados. Para isto, clique em “LocalInstance MySQL”, logo em seguida no ícone acima “create a new schema”, dê um nome ao banco de dados e clique em apply, conforme imagem abaixo:



Feito isso o banco de dados estará criado.

Criando aplicação e instalando o Sequelize

Onde o arquivo routes.js será o arquivo responsável pelas rotas do nosso projeto:

Copiar

```
import express from "express";

const routes = express.Router();

routes.get("/", (req, res) => {
  return res.json({ name: "Ciclano Fulano" });
});

export { routes as default };
```

E o arquivo server.js responsável pelo servidor criado pelo Express:

Copiar

```
import express from "express";
import routes from "./routes.js";

const app = express();

app.use(express.json());
app.use(routes);

app.listen(3000, () => console.log("Servidor iniciado na porta 3000"));
```

Agora precisamos instalar o Sequelize e o driver do MySQL, o banco que vamos utilizar nesse artigo, para isto, basta executar o seguinte comando: `npm install sequelize mysql2` .

Obs.: instalamos a dependência mysql2, pois ela possui melhor performance e suporte a recursos mais atuais do JavaScript como promises.

Configurando variáveis de ambiente

Em seguida precisamos criar o arquivo de configuração do banco de dados, para isso vamos criar um arquivo .env, onde ficarão as variáveis de ambiente da aplicação, como as de acesso ao banco de dados que serão diferentes em produção. Para isto vamos instalar o pacote dotenv, com o comando `npm install dotenv`. Logo em seguida criaremos o arquivo .env na raiz da nossa aplicação com os dados do nosso banco de dados:

Copiar

```
DB_NAME=sequelizenode
DB_USER=root
DB_PASSWORD=00000000
DB_HOST=localhost
```

Configurar a conexão do Sequelize com o Banco de Dados

Após setar as informações de nosso banco de dados, vamos criar o arquivo db.js dentro de uma pasta `src`, e nela precisamos passar para o sequelize os dados já configurados no arquivo .env da seguinte forma:

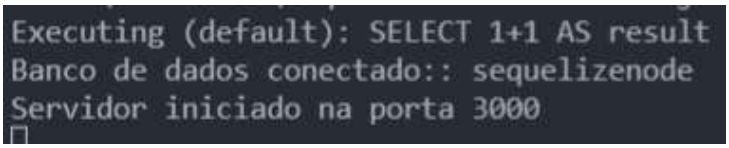
Copiar

```
import { Sequelize } from "sequelize"; // importar o sequelize
import dotenv from "dotenv/config.js"; // importar o dotenv para localizar as variáveis de a

const dbName = process.env.DB_NAME; // passar os dados do .env para as constantes
const dbUser = process.env.DB_USER;
const dbHost = process.env.DB_HOST;
const dbPassword = process.env.DB_PASSWORD;

const sequelize = new Sequelize(dbName, dbUser, dbPassword, {
  //passar os dados para o sequelize
  dialect: "mysql", //informar o tipo de banco que vamos utilizar
  host: dbHost, //o host, neste caso estamos com um banco local
});
```

```
import express from "express";
import routes from "./routes.js";
import db from "./src/db.js";

const app = express();

app.use(express.json());
app.use(routes);

db.sync(() => console.log(`Banco de dados conectado: ${process.env.DB_NAME}`));

app.listen(3000, () => console.log("Servidor iniciado na porta 3000"));
```

Importamos as configurações do db.js, em seguida executamos a função `db.sync()` .

Ao executar o comando `node server.js` teremos nosso servidor e a conexão com o banco efetuados com sucesso:



Configurando Model de uma tabela de acordo com o Sequelize

Perfeito, agora precisamos configurar o Model da nossa primeira tabela do banco, para isso vamos criar a pasta `models` dentro da pasta `src` , e criar o arquivo clientsModel.js logo em seguida:

```
    allowNull: false,
  },
  nome: {
    type: Sequelize.STRING,
    allowNull: false,
  },
  email: {
    type: Sequelize.STRING,
    allowNull: false,
    unique: true,
  },
});
```

Neste exemplo vamos criar uma tabela simples de clientes, com id, nome e email. Precisamos importar o sequelize, repare que importamos a classe Sequelize (maiúsculo) dentro de { }, e também a conexão do banco que configuramos em nosso arquivo db.js.

Agora vamos definir a tabela "client", vale ressaltar que neste momento colocamos a tabela no singular, o próprio Sequelize irá pluralizar ao criar a tabela no banco. Aqui, ao invés de utilizarmos uma query o ORM nos permite passar os atributos das tabelas de uma forma simples e intuitiva. Como podemos ver, ao criar o campo ID por exemplo, passamos um "type", dizendo que ele é inteiro e positivo, setamos como chave primária, que o valor é autoincremental e, para finalizar, que não pode ser vazio. Desta forma podemos usar uma série de características para os campos de nossa tabela, você pode verificar as possibilidades na documentação do Sequelize.

Configurando Controllers

Feito isso precisamos criar o nosso Controller. Para isto, vamos criar uma pasta `controllers` dentro de `src` e o arquivo clients.js dentro da pasta `controller` , neste arquivo vamos configurar as funções que serão invocadas em cada requisição referente às interações com o banco, neste primeiro momento podemos criar uma função para exibir todos os clientes em nosso banco, desta forma:

Copiar

Vamos importar o Model como ClientRepository, logo em seguida, criar a função findAll. Criar a constante clients, que será cada cliente de nosso banco, utilizando a função findAll do ClientRepository (uma das funções do Sequelize), e por fim, responder o json referente aos clientes cadastrados.

Agora precisamos passar essa função em nosso arquivo routes.js. para isso vamos importar o controller clients.js e criar a rota `/clients`, da seguinte maneira:

Copiar

```
import express from "express";
import clients from "./src/controllers/clients.js";

const routes = express.Router();

routes.get("/clients", clients.findAll);

export { routes as default };
```

Veja que passamos a função `clients.findAll`, como parâmetro para nossa rota. Agora podemos executar nossa aplicação e subir nosso servidor.

```
PS D:\TreinaWeb\sequelize> node server.js
Executing (default): CREATE TABLE IF NOT EXISTS `clients` (`id` INTEGER UNSIGNED NOT NULL
auto_increment , `nome` VARCHAR(255) NOT NULL, `email` VARCHAR(255) NOT NULL UNIQUE, `crea
tedAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB
;
Executing (default): SHOW INDEX FROM `clients`
Banco de dados conectado: sequelizenode
Servidor iniciado na porta 3000
```

Perceba que, caso a tabela clients não exista, ele irá criar uma nova, logo em seguida, exibir a mensagem de conexão com o banco e que o servidor foi iniciado na porta 3000. Ao entrar na rota `/clients`, teremos o seguinte resultado:

```
[]
```

Criando “CRUD” e testando rotas

```
function findAll(req, res) {
  ClientRepository.findAll().then((result) => res.json(result));
}

function findClient(req, res) {
  ClientRepository.findByPk(req.params.id).then((result) => res.json(result));
}

function addClient(req, res) {
  ClientRepository.create({
    nome: req.body.nome,
    email: req.body.email,
  }).then((result) => res.json(result));
}

async function updateClient(req, res) {
  await ClientRepository.update(
    {
      nome: req.body.nome,
      email: req.body.email,
    },
    {
      where: {
        id: req.params.id,
      },
    }
  );

  ClientRepository.findByPk(req.params.id).then((result) => res.json(result));
}

async function deleteClient(req, res) {
  await ClientRepository.destroy({
    where: {
      id: req.params.id,
    },
  });

  ClientRepository.findAll().then((result) => res.json(result));
}
```

para efetuar a manipulação dos dados no banco. E, logo em seguida, retornar o registro atualizado para ficar mais didático a visualização das alterações feitas.

Feito isso precisamos criar uma rota para cada operação configurada, vamos atualizar o arquivo routes.js:

Copiar

```
import express from "express";
import clients from "./src/controllers/clients.js";

const routes = express.Router();

routes.get("/clients", clients.findAll);
routes.post("/clients", clients.addClient);
routes.get("/clients/:id", clients.findClient);
routes.put("/clients/:id", clients.updateClient);
routes.delete("/clients/:id", clients.deleteClient);

export { routes as default };
```

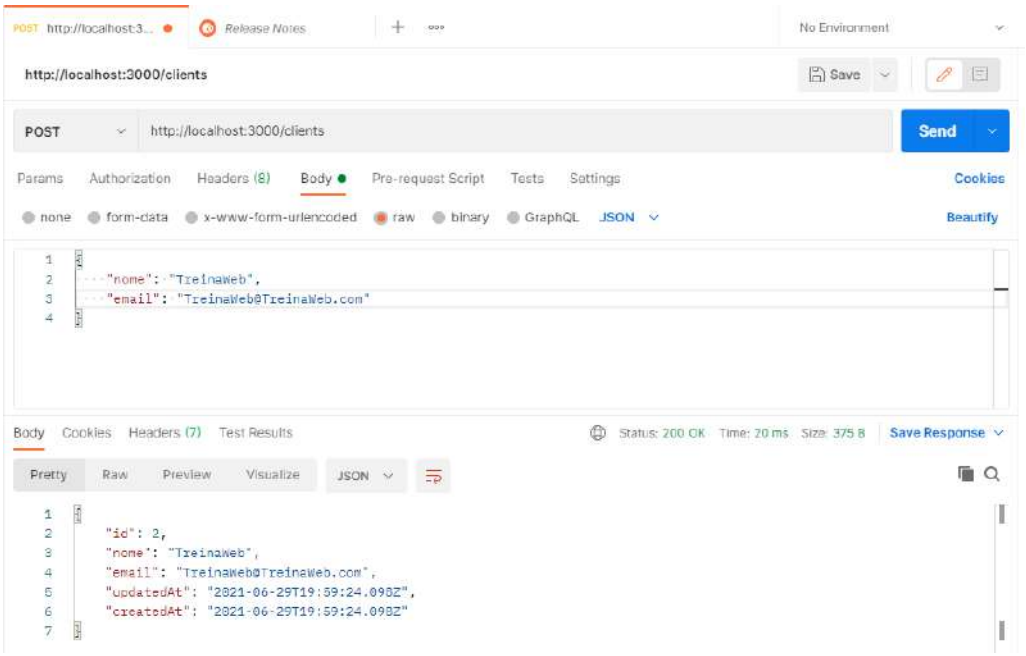
Testando rotas e operações no Banco de Dados

Agora podemos testar nossa aplicação, para isto vamos usar o Postman para efetuar as requisições, primeiro cadastrando um cliente, realizando uma requisição HTTP com o verbo POST para a rota `/clients` :

Obs.: O Postman é uma ferramenta que podemos utilizar para testar requisições HTTP. Você pode acessar o site da ferramenta para saber mais sobre ela e efetuar o download (também existe a versão no navegador caso seja sua preferência).

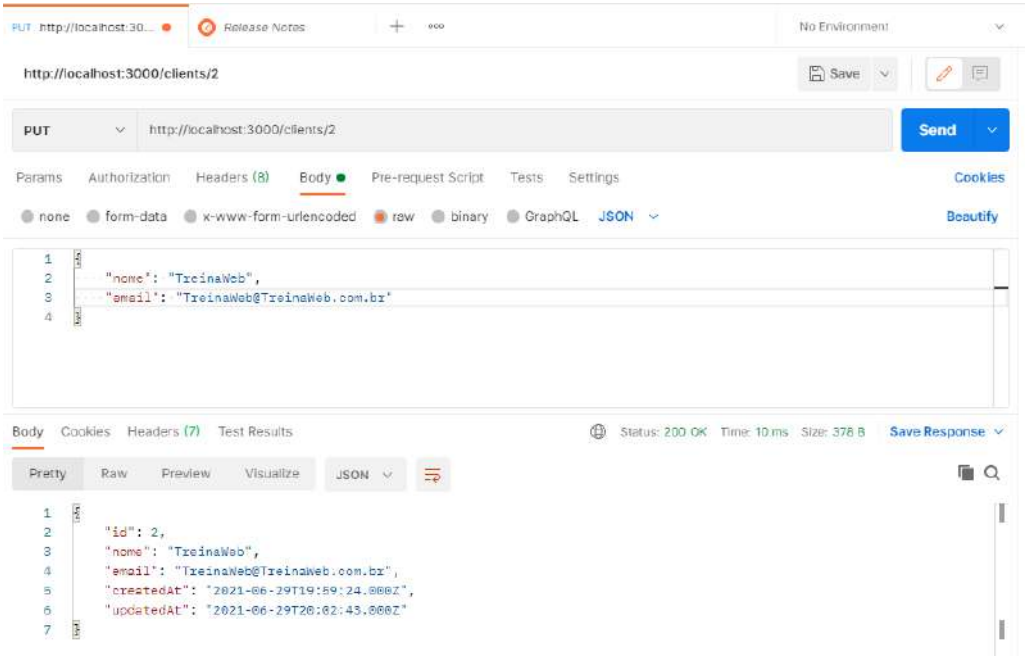
Perceba que passamos o nome e email via json, ele efetuou o cadastro em nosso banco de dados e já retornou também os campos id, updateAt e createdAt preenchidos. Esses dois últimos o Sequelize cria como padrão, é interessante pois sabemos quando aquele registro foi criado e atualizado.

Ao enviar um novo registro, perceba que nosso id também foi incrementado:



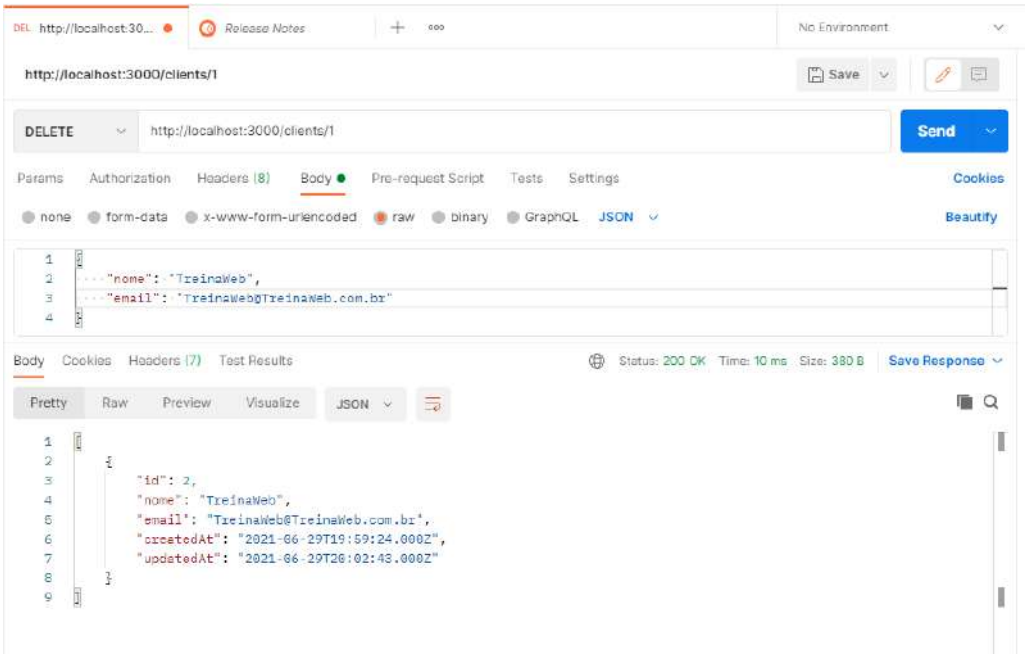
Ótimo, agora podemos testar as outras rotas, como buscar por id:

Atualizar o email do registro de id 2 por exemplo:



Buscar todos os registros:

Excluir um registro:



Conclusão

Neste artigo aprendemos a utilizar um ORM, no caso o Sequelize, para trabalharmos com o Node.js e o banco de dados MySQL, executando as operações básicas relacionadas a manipulação de registros do banco de dados. Com isso obtemos algumas vantagens, como: produtividade, código mais elegante, facilidade de manutenção, legibilidade de código, entre outras.

Veja quais são os tópicos abordados durante o curso de Express - Desenvolvendo aplicações web:

- Conhecendo a estrutura;
- Entendendo os objetos Request e Response;
- Servindo arquivos estáticos;
- Roteamento;
- Conexão com banco de dados;
- Template Engine;
- Express Generator.

#JavaScript #Back-end #ORM #Node.js #Sequelize

Autor(a) do artigo

Wesley Gado

Formado em Análise e Desenvolvimento de Sistemas pelo Instituto Federal de São Paulo, atuou em projetos como desenvolvedor Front-End. Nas horas vagas grava Podcast e arrisca uns três acordes no violão.

Todos os artigos

Artigos relacionados

Ver todos →