

1. Criando uma API RESTful com Node.js + Express + Mongoose

Neste capítulo vamos aprender como criar uma API REST em NodeJS com Express e consumir dados do banco de dados não relacional MongoDB.

1.1 Introdução

Em sistemas web modernos, uma API (Application Programming Interface) RESTful aguarda requisições HTTP vindas de um navegador da web (ou de outro tipo de cliente). Cabe lembrar que o REST (Representational State Transfer) é uma abstração da arquitetura da Web que utiliza dos verbos (GET, POST, DELETE, UPDATE, entre outros) do protocolo HTTP como base para as requisições que recebe. Assim, quando uma requisição HTTP é recebida, esta API descreve quais ações são necessárias com base no padrão de URL e, também, quais informações associadas estão contidas em dados obtidos via POST ou GET. Dependendo da ação solicitada, pode-se ler ou escrever informações em um banco de dados ou executar outras tarefas necessárias para satisfazer a requisição. Após, a API retornará uma resposta ao navegador da web.

O Node.js é um runtime JavaScript server-side, ou seja, uma solução que possibilita ao desenvolvedor executar aplicações, escritas em JavaScript, no lado do servidor e de forma simples, rápida e performática. Algumas tarefas comuns no desenvolvimento web não são suportadas diretamente pelo Node.js. Por exemplo, se precisarmos que a nossa aplicação atenda a diferentes verbos HTTP (por exemplo GET, POST, DELETE, etc), ou que gerencie requisições de diferentes URLs (as chamadas rotas da aplicação), ou utilize templates para mostrar as respostas (o objeto `response`) de maneira dinâmica, vamos encontrar dificuldades usando apenas o Node.js. Desta forma, o Express.js é um dos frameworks mais utilizados do Node.js. Resumidamente, o Express oferece uma estrutura web rápida, flexível e minimalista para aplicativos Node.js. Basicamente, o Express é uma estrutura web de roteamento e middlewares. Neste momento, considere que middlewares são funções que têm acesso ao objeto de requisição (`request`), ao objeto de resposta (`response`), e a próxima função middleware (`next`) no ciclo de requisição-resposta de uma aplicação web.

O MongoDB é um banco de dados NoSQL orientado a objetos, simples, dinâmico e escalonável. É baseado no modelo de armazenamento de documentos NoSQL, que significa “Not Only SQL” (não somente SQL). Desta forma, os objetos de dados são armazenados como documentos, separados dentro de uma coleção - em vez de armazenar os dados nas colunas e linhas de uma tabela no banco de dados relacional. O Mongoose, por sua vez, é uma biblioteca do Nodejs que proporciona uma solução baseada em esquemas para modelar os dados da sua aplicação. Ele possui sistema de conversão de

tipos, validação, criação de consultas e hooks para lógica de negócios. O Mongoose fornece um mapeamento de objetos do MongoDB similar ao ORM (Object Relational Mapping), ou ODM (Object Data Mapping) no caso do Mongoose. Isso significa que o Mongoose traduz os dados do banco de dados para objetos JavaScript para que possam ser utilizados por sua aplicação.

1.2. Projeto de Exemplo

Este exemplo mostra como criar um aplicativo desenvolvido em Node.js que se conecta a um cluster [MongoDB Atlas](#). MongoDB Atlas é um serviço de banco de dados em nuvem que hospeda seus dados em instâncias do MongoDB. Vejamos os passos que precisamos seguir neste exemplo:

1.2.1 Criando uma instância do MongoDB

O MongoDB Atlas é um serviço de DBaaS (Banco de Dados como Serviço) oferecido pela MongoDB. Basicamente, você somente precisa se preocupar em administrar os dados que estarão lá. A infraestrutura e a manutenção das máquinas, bem como segurança disso, fica por conta deles.

Primeiro, vamos fazer a criação do nosso Cluster (o MongoDB Atlas tem como implementação padrão um Replicaset com três máquinas). Assim, para iniciar no MongoDB Atlas, precisamos acessar o [site oficial](#) deste serviço. Ao clicar em “Start free”, você será encaminhado para uma tela de cadastro. O MongoDB Atlas oferece a camada M0, que é muito útil pra quem quer conhecer ou fazer algum tipo de teste com o MongoDB.

1 Select a cloud provider



2 Select a cluster configuration



Após essas configurações, você precisa dar um nome para o Cluster e em poucos minutos ele estará disponível. Agora que você já tem o cluster implementado, basta você se conectar com ele clicando no botão “Connect”.

Entretanto, você também pode optar por instalar o MongoDB em sua máquina. Para isso, siga as seguintes [instruções](#) no site oficial do MongoDB.

1.2.2 Configurando o projeto

Vamos criar um diretório chamado node-api-mongo que irá retornar e cadastrar informações sobre os cursos oferecidos no IFRS. Em seguida, vamos iniciar um projeto Node com auxílio do gerenciador de pacotes Yarn. Assim, digite os seguintes comandos no terminal:

```
cd node-api-mongo  
yarn init -y
```

Este comando cria um arquivo chamado package.json. Cabe lembrar que se você especificar a opção -y no comando, o Yarn aceita automaticamente os valores padrões.

Agora que já temos o nosso projeto criado, vamos começar a criar nosso servidor web. Assim, vamos abrir o projeto no Visual Studio Code, digitando o seguinte comando no terminal, dentro do diretório do nosso projeto:

```
code .
```

Em seguida, vamos realizar o download e instalação de nossas dependências do projeto. Neste caso, vamos utilizar o Express. Assim digite o seguinte comando no terminal, dentro do diretório do projeto:

```
yarn add express
```

OBS: Você pode usar o terminal que fica integrado ao Visual Studio Code

Em seguida, adicione o Mongoose às dependências do projeto. O Mongoose é uma biblioteca do Nodejs que fornece um modelo de mapeamento de objetos do MongoDB, atuando como um ODM (Object Data Mapping). Use o seguinte comando para instruir o Yarn a baixar e instalar o pacote mongoose.

```
yarn add mongoose
```

Nós vamos utilizar em nosso projeto o [Nodemon](#). O Nodemon é um utilitário que monitora quaisquer mudanças em seu código fonte e automaticamente reinicia seu servidor. Logo, vamos instalar o Nodemon via terminal:

```
yarn add nodemon --dev
```

Após, vamos criar o arquivo responsável por configurar e subir o servidor. Assim, crie no diretório raiz um arquivo chamado index.js.

Agora, vamos configurar melhor o script de execução do Nodemon para o nosso projeto. Assim, faça as seguintes alterações no arquivo package.json:

```
{
  "name": "node-api-mongo",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "dev": "nodemon"
  },
  "dependencies": {
    "express": "^4.17.1",
    "mongoose": "^5.10.13"
  },
  "devDependencies": {
    "nodemon": "^2.0.6"
  }
}
```

Inicialmente, alteramos o “main” informando o caminho do nosso arquivo principal do projeto. Em seguida, criamos um “script” de desenvolvimento “dev” que irá executar o comando “nodemon”. O Nodemon irá executar automaticamente o arquivo principal definido em “main”.

1.2.3 Criando a primeira rota

Neste momento, vamos configurar o servidor web e criar nossa primeira rota. Assim, abra o arquivo index.js e digite o seguinte código:

```
const express = require("express");
// Cria uma aplicação Express
const app = express();
//Define uma rota
app.get("/courses", (req, res) => {
```

```
    return res.send("Cursos do IFRS");
  });
// Inicia o servidor na porta '3000'
app.listen(3000, () => {
  console.log("Exemplo de aplicativo ouvindo a porta 3000");
});
```

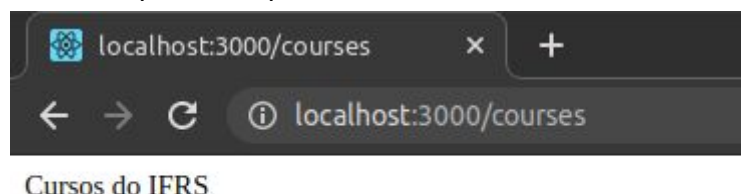
Agora, vamos iniciar o nosso servidor web digitando no terminal o seguinte com

```
yarn dev
```

Observe que ele executou o script “dev” que acabamos de criar e executou o nodemon.

```
mcrosito@mcrosito-IFRS:~/Documentos/react/node-api-mongo$ yarn dev
yarn run v1.22.5
$ nodemon
[nodemon] 2.0.6
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Exemplo de aplicativo ouvindo a porta 3000
```

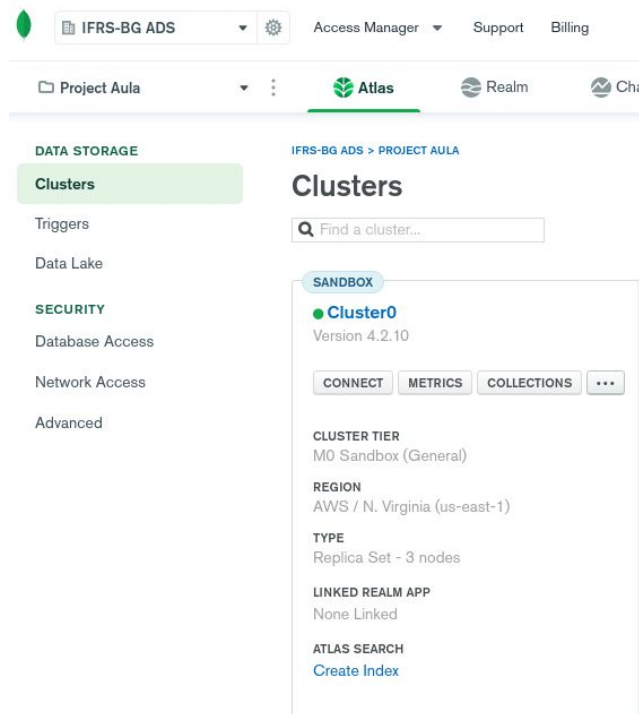
Com o servidor em execução, você pode acessar o localhost:3000/courses em seu navegador para ver o exemplo de resposta retornado.



Pronto. Agora podemos implementar novas funcionalidades neste projeto.

1.2.4 Configurando a conexão a um cluster MongoDB

Nesta etapa, nós vamos configurar nosso um aplicativo para se conectar à sua instância do MongoDB Atlas. Caso você tenha instalado o MongoDB em sua máquina, pule para o próximo passo. Bem, precisamos recuperar a string de conexão de nossa base de dados. Para recuperar sua string de conexão e o usuário que você criou no MondoBD Atlas, faça login em sua conta do Atlas, navegue até a seção Clusters e clique no botão Conectar para o cluster ao qual deseja se conectar, conforme mostrado abaixo.



Na tela que abrir, no passo “Setup connection security”, se for solicitado, informe o IP Address de sua máquina e crie um usuário e senha. Em seguida, no próximo passo, selecione a opção “Connect your Application”. Em seguida, prossiga para a etapa “Connect Your Application” e selecione o driver Node.js. Selecione a guia "Connection String Only" e clique no botão Copiar para copiar a string de conexão para a área de transferência.

1 Select your driver and version

DRIVER	VERSION
Node.js	3.6 or later

2 Add your connection string into your application code

☐ Include full driver code example

```
mongodb+srv://dbUser:<password>@cluster0.cifcl.mongodb.net/<dbname>
```

Copy

Salve sua string de conexão em um local seguro que você possa acessar na próxima etapa. Após, volte ao arquivo responsável por configurar e subir o servidor, o index.js. Agora, adicione o seguinte código neste arquivo, substituindo a variável `uri` por sua string de conexão. Certifique-se de substituir a seção "`<password>`" da string de conexão pela senha que você criou para o seu usuário. Substitua `<dbname>` pelo nome do banco de dados que as conexões usarão por padrão.

```
// Importando as dependências do projeto
const express = require("express");
const mongoose = require('mongoose');

// Cria uma aplicação Express
const app = express();

//Cria a conexão com o banco de dados
const uri =
"mongodb+srv://dbUser:12345@cluster0.cifcl.mongodb.net/ifrs_db?ret
ryWrites=true&w=majority";
mongoose.connect(uri,
  { useNewUrlParser: true, useUnifiedTopology: true });
const db = mongoose.connection;
//trata os erros da conexão
mongoose.connection.on('error',function (err) {
  console.log('Erro na conexão Mongoose padrão: ' + err);
});
//A conexão foi feita com sucesso
db.once('open', function() {
  console.log('Estamos conectados no banco de dados!')
});

//Define uma rota
app.get("/courses", (req, res) => {
  return res.send("Cursos do IFRS");
});
// Inicia o servidor na porta '3000'
app.listen(3000, () => {
  console.log("Exemplo de aplicativo ouvindo a porta 3000");
});
```

O driver MongoDB está atualizando seu analisador de string de conexão atual. Dessa forma, eles disponibilizaram uma flag `useNewUrlParser` para permitir que os usuários voltem ao analisador antigo se estes encontrarem um bug no novo analisador. A flag `useUnifiedTopology` informa que optamos por usar o novo mecanismo de gerenciamento de conexão do driver MongoDB.

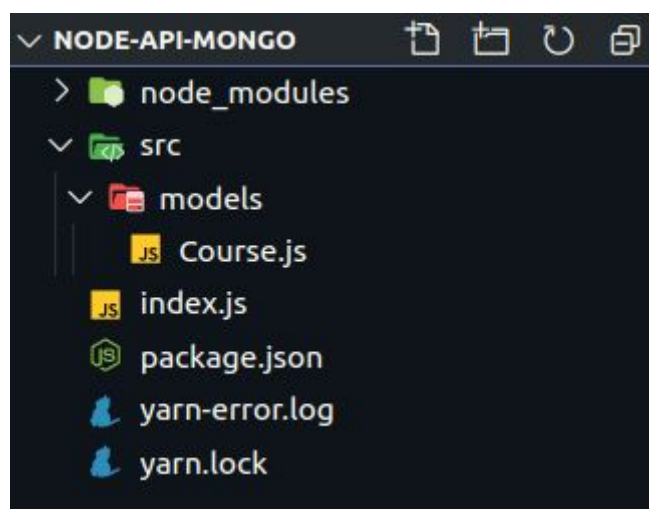
Observe o resultado no terminal informando que nossa aplicação está sendo executada, de modo que conseguimos nos conectar ao nosso banco de dados.

```
Estamos conectados no banco de dados.  
[nodemon] restarting due to changes...  
[nodemon] starting `node src/index.js`  
Exemplo de aplicativo ouvindo a porta 3000  
Estamos conectados no banco de dados!  
█
```

1.2.5 Definindo e criando modelos

Os modelos são definidos usando a interface `Schema`. O `Schema` permite definir os campos armazenados em cada documento, junto com seus requisitos de validação e valores padrão. Os `Schemas` são então compilados em modelos usando o método `mongoose.model()`. Depois de ter um modelo, você pode usá-lo para localizar, criar, atualizar e excluir objetos de um determinado tipo.

Primeiro, vamos criar um diretório dentro de `src` chamado de `models`. Dentro de `models`, crie um arquivo chamado `Course.js`.



Agora, vamos criar o `Schema` que será utilizado para gerar um modelo sobre os cursos do IFRS. Assim, digite o seguinte código em `Course.js`:


```
// Importar módulos necessários
const mongoose = require('mongoose');

// Define o schema do Curso
const CourseSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  url: {
    type: String,
    required: true,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  }
});

// Registra o model Course em nossa aplicação informando seu
schema
mongoose.model('Course', CourseSchema);
```

Observe que, primeiro, nós importamos o módulo do Mongoose. Em seguida, criamos um Schema para comportar as informações sobre os cursos de nossa aplicação. No final, observe que os modelos são criados a partir de Schemas usando o método `mongoose.model()`.

Agora, volte para o arquivo `index.js` para que possamos usar este modelo para criar um Curso em nossa base de dados:

```

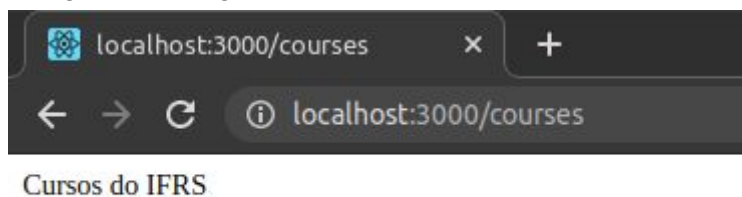
...
//Registra o Model em index.js
require('./src/models/Course');
//Referencia o model Course
const Course = mongoose.model('Course');

//Define uma rota
app.get("/courses", (req, res) => {
  Course.create({
    name: 'Análise e Desenvolvimento de Sistemas',
    description: 'Curso com duração mínima de 3 anos.',
    url: 'https://ifrs.edu.br/bento/'
  });

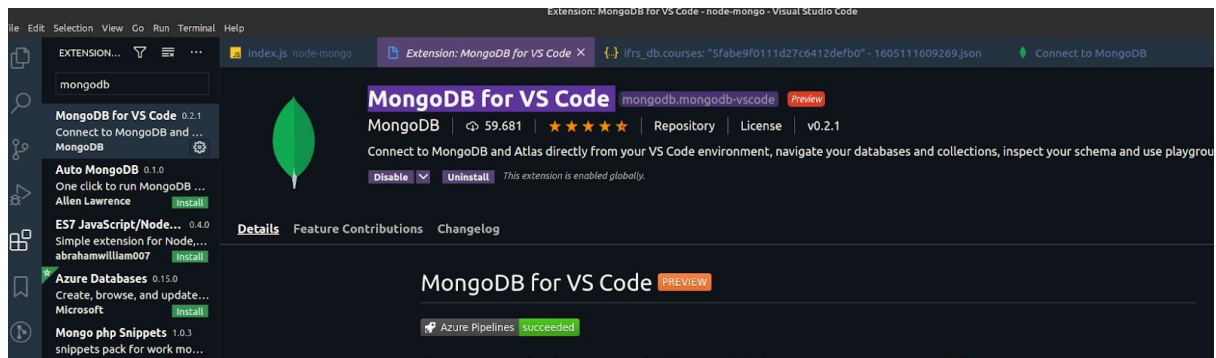
  return res.send("Cursos do IFRS");
});
...

```

Agora, acesse o navegador na seguinte url: <http://localhost:3000/courses>:



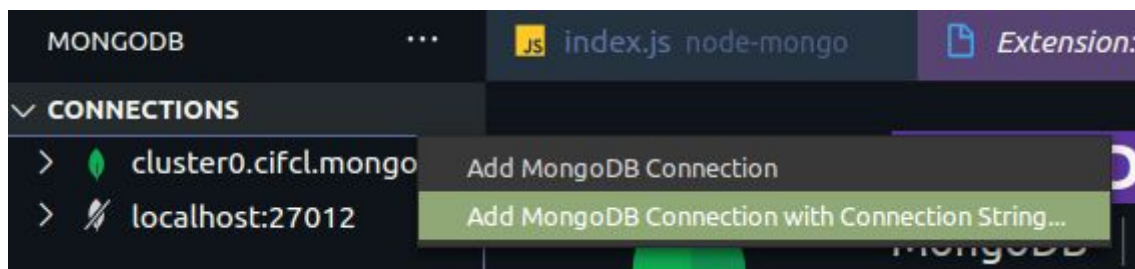
Pronto, nossa aplicação criou o primeiro curso em nossa base de dados. Para verificar se esta informação foi criada corretamente, vamos usar a extensão “MongoDB for VS Code” no Visual Studio Code. Caso você ainda não tenha instalado ela, clique no botão da direita referente às extensões do VS Code. Em seguida, pesquise por MongoDB. Clique na extensão “MongoDB for VS Code” e instale-a.



Em seguida, irá aparecer na sua barra lateral da esquerda o ícone em formato de “folha”.

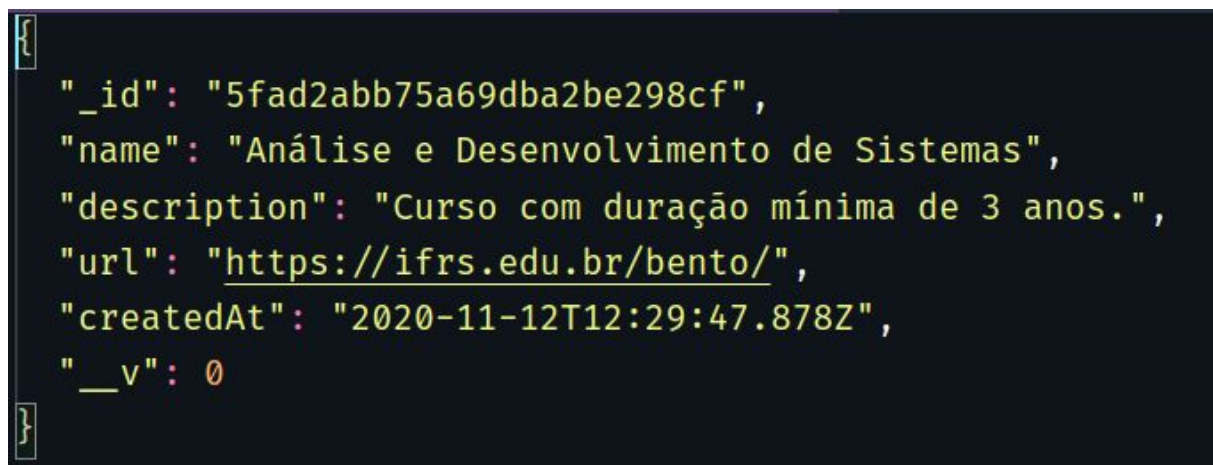


Clique neste ícone. Em “Connections”, clique na opção “Add MongoDB Connection with Connection String...” e informe a string de conexão que usamos em nosso aplicativo.



Pronto, você está visualizando nossa base de dados no MongoDB Atlas.

Assim, podemos acessar o documento com as informações do curso que acabamos de criar. Clique no documento específico e veja o seu conteúdo.



1.2.6 Organizando nossas rotas

Bem, neste passo vamos organizar as rotas de nossa aplicação. Usaremos a classe `express.Router` para criar um manipulador de rota modular. Cabe ressaltar que uma instância de `Router` é um middleware e sistema de roteamento completo. Assim, vamos criar um roteador como sendo um módulo, definindo algumas rotas para ele. Desta forma, crie em `src` um arquivo chamado `routes.js` com o seguinte código:

```
// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();

//Define uma rota
routes.get("/", (req, res) => {
  return res.send("Cursos do IFRS");
});

module.exports = routes;
```

Agora, vamos usar este módulo no nosso `index.js`. Porém, cabe ressaltar que em uma aplicação costumamos criar vários Models. Consequentemente, teríamos que criar um `require('modelX')` para cada modelo. Entretanto, podemos fazer uso da biblioteca [require-dir](#), que permite para usar o `require()` recursivamente em um diretório. Veja um exemplo de uso:

```
const requireDir = require("require-dir");
requireDir("./models");
```

Assim, abra um terminal e instale esta biblioteca como seguinte comando:

Agora, vamos iniciar o nosso servidor web digitando no terminal o seguinte com

```
yarn add require-dir
```

Agora, altere o arquivo `index.js` da seguinte forma:

```
// Importando as dependências do projeto
const express = require("express");
const mongoose = require('mongoose');
```

```

const requireDir = require("require-dir");

// Cria uma aplicação Express
const app = express();

//Cria a conexão com o banco de dados
const uri =
"mongodb+srv://dbUser:12345@cluster0.cifcl.mongodb.net/ifrs_db?ret
ryWrites=true&w=majority";
mongoose.connect(uri,
  { useNewUrlParser: true, useUnifiedTopology: true });

//Registra o Model em index.js
requireDir("./src/models");

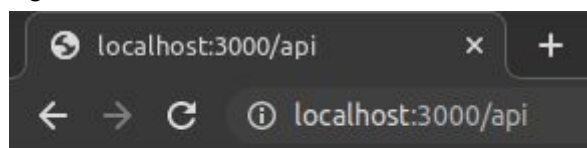
// Redireciona o caminho http://localhost:3000/api para o routes
app.use('/api', require('./src/routes'));

// Inicia o servidor na porta '3000'
app.listen(3000, () => {
  console.log("Exemplo de aplicativo ouvindo a porta 3000");
});

```

Cabe salientar que o `app.use()` recebe todo o tipo de requisição HTTP feita para a url “<http://localhost:3000/api>” e repassar para o módulo `routes` (que irá fazer o tratamento destas requisições).

Veja o resultado no navegador:



1.2.7 Criando nosso Controller

Uma vez que criamos nosso Model, podemos agora criar o Controller (seguindo o padrão MVC, que é o acrônimo de Model-View-Controller). Um Controller irá coordenar as ações

referentes à esse Model. Assim, crie em src um diretório chamado controllers e, depois, crie um arquivo chamado CourseController.js com o seguinte código:

```
// Importando as dependências
const mongoose = require('mongoose');
//Referencia o model Course
const Course = mongoose.model('Course');
// Vamos exportar um objeto com algumas funções
module.exports = {
  // Vai retornar todos os cursos de nosso banco de dados
  async index(req, res) {
    // retorna os cursos de nosso banco de dados
    const courses = await Course.find();
    // vamos retornar em formato JSON
    return res.json(courses);
  }
}
```

Observe que este módulo está exportando um objeto que contém algumas funções. O método `index()` vai retornar todos os cursos de nosso banco de dados. Ainda, veja que o `await` garante que a próxima linha somente seja executada quando eu retornar os registros do banco de dados. O retorno dos dados será em formato JSON.

Agora, precisamos associar as rotas da nossa aplicação ao controller. Para isso, altere o arquivo `routes.js` da seguinte forma:

```
// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();

// Referencia o Controller CourseController
const CourseController = require('./controllers/CourseController');
// associa as rotas ao seu método do Controller
routes.get('/courses', CourseController.index);
```

```
module.exports = routes;
```

Acesse a url: <http://localhost:3000/api/courses> e veja o resultado na tela uma listagem JSON dos cursos que temos cadastrados em nossa base de dados:

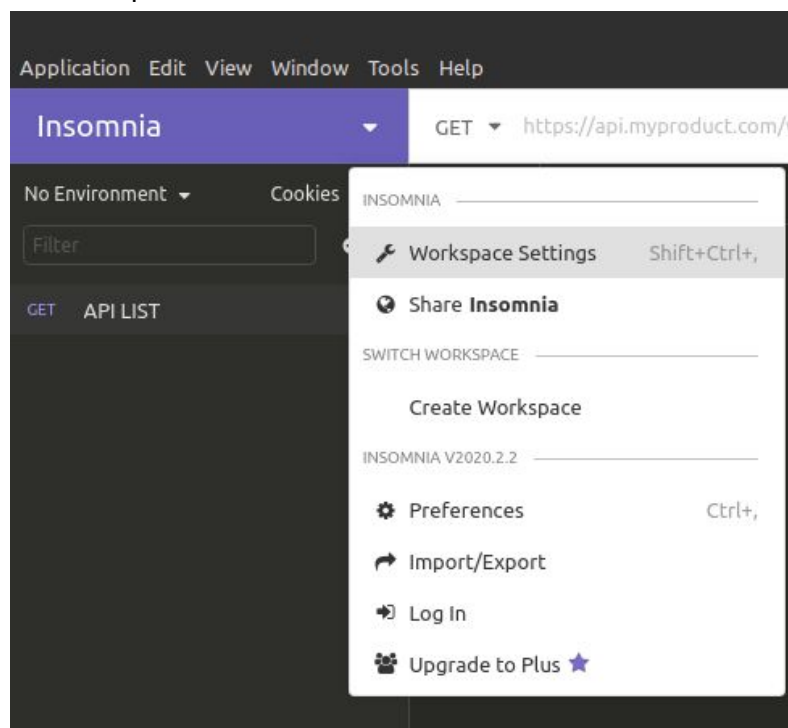


```
[{"_id": "5fad2abb75a69dba2be298cf", "name": "Análise e Desenvolvimento de Sistemas", "description": "Curso com duração mínima de 3 anos.", "url": "https://ifrs.edu.br/bento/", "createdAt": "2020-11-12T12:29:47.878Z", "__v": 0}]
```

1.2.8 Usando o Insomnia

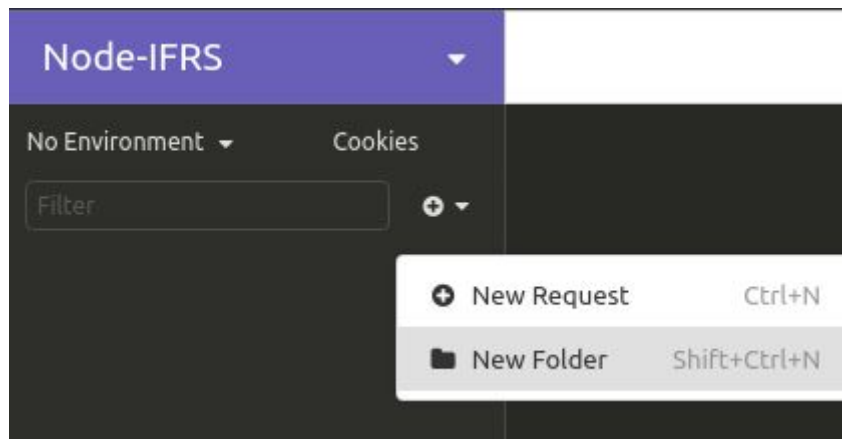
Agora, vamos testar este código com o Insomnia. Cabe lembrar que o [Insomnia](#) é um aplicativo open-source que permite organizar, executar e depurar as requisições HTTP.

Inicialmente, vamos criar um workspace no Insomnia para cada um dos projetos que formos testar. Os workspaces são usados para isolar projetos dentro do Insomnia. Você pode criar, excluir e alternar entre os workspaces no menu superior esquerdo do aplicativo. Assim, clique em “Create Workspace”



Na janela que abrir, vamos informar o nome deste workspace, neste caso, chamei de Node-IFRS.

Em nossos próximos exemplos, vamos criar tratar dos métodos HTTP (GET, POST, PUT e DELETE) para uma rota chamada “courses”. Desta forma, considerando que uma API pode ter diferentes rotas, podemos criar uma pasta no workspace para cada rota. Assim, vamos pressionar o botão com um símbolo de mais (+) e então clicar em “New Folder”.



Na janela que abrir, vamos informar o nome da pasta, neste caso “courses”. Agora, dentro desta pasta, vamos adicionar um “New Request” para testar o método GET. Nesta nova janela, informe o nome “API GET COURSES” e selecione o verbo GET.

New Request

Name

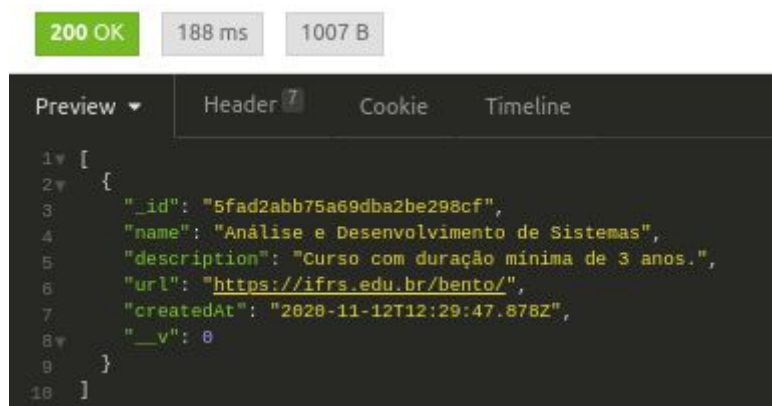
API GET COURSES

GET

* Tip: paste Curl command into URL afterwards to import it

Create

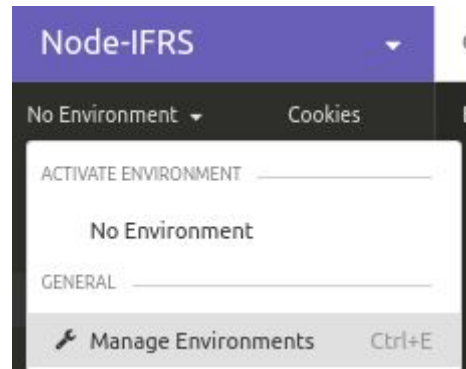
Como resultado, se consultarmos a rota `http://localhost:3000/api/courses`, teremos todas as contas em nosso array `courses`:



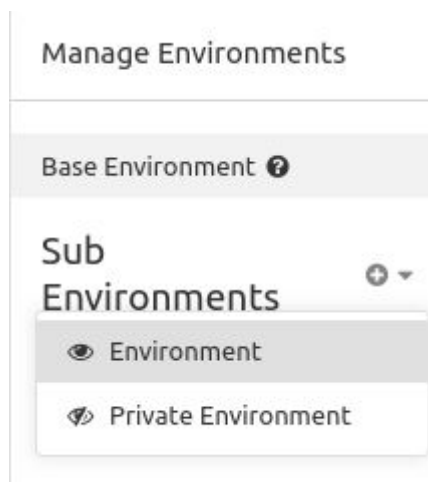
Usamos o servidor `http://localhost:3000/api` como base para nossos testes. Contudo, imagine que precisássemos mudar o endereço desse servidor para realizar novos testes. Ou se precisássemos testar a nossa API em produção. Neste caso, precisaríamos alterar o endereço deste servidor em todas as requisições que já criamos.

Para facilitar esse processo, o Insomnia possui variáveis de ambiente, que podem ser compartilhadas entre as diferentes requisições definidas em diferentes ambientes de testes.

Para acessá-las, basta clicar em “No Environment” > “Manage Environments” abaixo do menu principal do Insomnia:



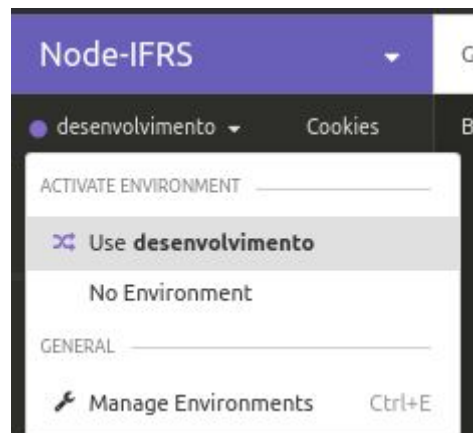
Uma nova janela será aberta. Nela, podemos adicionar Sub Environments, por exemplo, um para conter informações para “desenvolvimento” e outro para “produção”. Clique para adicionar um Environment:



Assim, crie um Sub Environment chamado “desenvolvimento”. Nele podemos ver uma área para adicionar um código JSON. Aqui podemos definir nossas variáveis de ambiente. Por exemplo, vamos definir que a nossa URL base será o localhost na porta 3000. Usaremos o seguinte código para isso:



Agora, vamos configurar o Insomnia para visualizar este Sub Environment de desenvolvimento:



O próximo passo é substituir a string `http://localhost:3000/api` pela variável `base_url` em nossas requisições. Para isso, basta digitar o nome da variável e o próprio Insomnia irá sugerir-la nas requisições. Por exemplo, veja como ficou a URI para a rota especificada para “API GET COURSES”

GET `base_url` /courses

1.2.9 Criando novos registros no MongoDB

Agora, vamos criar a funcionalidade para adicionar novos registros ao nosso banco de dados. Assim, altere o arquivo `CourseController.js` da seguinte forma:

```
// Importando as dependências
const mongoose = require('mongoose');
//Referencia o model Course
const Course = mongoose.model('Course');
// Vamos exportar um objeto com algumas funções
module.exports = {
  // Vai retornar todos os cursos de nosso banco de dados
  async index(req, res){
    // retorna os cursos de nosso banco de dados
    const courses = await Course.find();
    // vamos retornar em formato JSON
    return res.json(courses);
  },
}
```

```
// Criar um novo curso
async store(req, res){
  const course = await Course.create(req.body);
  // Vamos retornar o curso que criamos
  return res.json(course);
},
}
```

Observe que estamos passando os dados que vem no corpo da requisição para a criação de um novo curso em:

```
const course = await Course.create(req.body);
```

Em seguida, precisamos criar esta nova rota em routes.js:

```
// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();
// Referencia o Controller CourseController
const CourseController = require('./controllers/CourseController');
// associa as rotas ao seu método do Controller
routes.get('/courses', CourseController.index);
routes.post('/courses', CourseController.store);

module.exports = routes;
```

Bem, uma vez que vamos receber dados no corpo da requisição, precisamos avisar o express que ele deve permitir que sejam passados dados em formato JSON. Assim, altere o arquivo index.js:

```
// Importando as dependências do projeto
const express = require("express");
const mongoose = require('mongoose');
const requireDir = require("require-dir");
// Cria uma aplicação Express
```

```
const app = express();

//Permitir enviar dados para a App no formato JSON
app.use(express.json());

...
```

Para testar esta rota, vamos no Insomnia e criamos uma nova requisição chamada “API POST COURSES” como sendo do tipo POST e trabalhando com dados JSON.

New Request ✕

Name

API POST COURSES

POST ▾

JSON ▾

* Tip: paste Curl command into URL afterwards to import it

Create

Agora, vamos testar esta rota passando dados de um novo curso:

POST ▾ base_url /courses Send

JSON ▾

Auth ▾

Query

Header 1

Docs

```
1 {
2   "name": "Agronomia",
3   "description": "Curso superior de Agronomia",
4   "url": "http://ifrs.edu.br/bento"
5 }
```

Ao realizarmos esta requisição, teremos o seguinte resultado:

200 OK

216 ms

177 B

Preview ▾

Header 7

Cookie

Timeline

```
1 {
2   "_id": "5fada804b8fe8443f7859a04",
3   "name": "Agronomia",
4   "description": "Curso superior de Agronomia",
5   "url": "http://ifrs.edu.br/bento",
6   "createdAt": "2020-11-12T21:24:20.397Z",
7   "__v": 0
8 }
```

1.2.10 Mostrando detalhes de um registro no MongoDB

Agora, vamos criar a funcionalidade para mostrar os detalhes de um curso específico do nosso banco de dados. Assim, altere o arquivo `CourseController.js` da seguinte forma:

```
// Importando as dependências
const mongoose = require('mongoose');

//Referencia o model Course
const Course = mongoose.model('Course');

// Vamos exportar um objeto com algumas funções
module.exports = {

  ...

  // Mostrar o detalhe de um curso
  async show(req, res){

    const course = await Course.findById(req.params.id);

    // Vamos retornar o course que encontramos
    return res.json(course);

  },
}
```

Observe que passamos um `id` que será informado junto com a requisição. Este `id` será informado na rota, de modo que precisamos alterar o `routes.js`:

```
// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();

// Referencia o Controller CourseController
const CourseController = require('./controllers/CourseController');

// associa as rotas ao seu método do Controller
routes.get('/courses', CourseController.index);
routes.get('/courses/:id', CourseController.show);
```

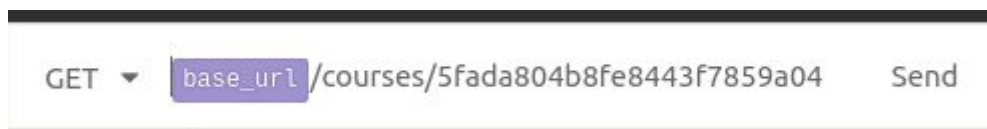
```
routes.post('/courses', CourseController.store);  
module.exports = routes;
```

Para testar esta rota, vamos no Insomnia e criamos uma nova requisição chamada “API SHOW COURSE” como sendo do tipo GET.



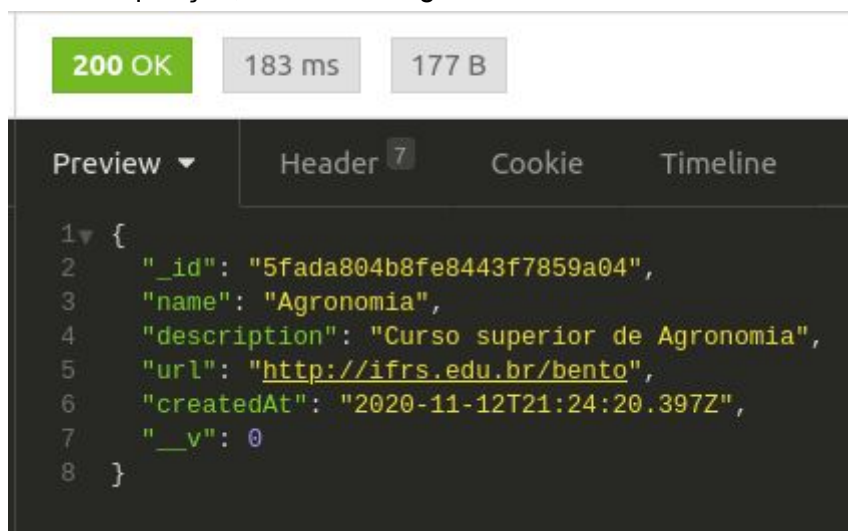
The image shows the 'New Request' form in the Insomnia client. The 'Name' field is filled with 'API SHOW COURSE'. The 'Method' dropdown is set to 'GET'. A tip at the bottom says '* Tip: paste Curl command into URL afterwards to import it'. A 'Create' button is in the bottom right corner.

Agora, vamos testar esta rota passando o id de um curso específico na URL. Neste caso, vou usar o id do curso de Agronomia que acabamos de criar, ou seja, 5fada804b8fe8443f7859a04.



The image shows the request bar in Insomnia. The method is 'GET'. The URL is 'base_url /courses/5fada804b8fe8443f7859a04'. A 'Send' button is on the right.

Ao realizarmos esta requisição, teremos o seguinte resultado:



The image shows the response preview in Insomnia. The status is '200 OK', the response time is '183 ms', and the size is '177 B'. The response body is a JSON object:

```
1 {  
2   "_id": "5fada804b8fe8443f7859a04",  
3   "name": "Agronomia",  
4   "description": "Curso superior de Agronomia",  
5   "url": "http://ifrs.edu.br/bento",  
6   "createdAt": "2020-11-12T21:24:20.397Z",  
7   "__v": 0  
8 }
```

1.2.11 Atualizando registro no MongoDB

Agora, vamos criar a funcionalidade para atualizar dados de um curso específico do nosso banco de dados. Assim, altere o arquivo CourseController.js da seguinte forma:

```
// Importando as dependências
```

```

const mongoose = require('mongoose');
//Referencia o model Course
const Course = mongoose.model('Course');
// Vamos exportar um objeto com algumas funções
module.exports = {
  ...
  // Atualizar um curso
  async update(req, res){
    // procura um curso pelo ID e atualiza ele
    const course = await Course.findByIdAndUpdate(req.params.id,
req.body, { new: true });
    // Vamos retornar o curso que encontramos
    return res.json(course);
  },
}

```

Observe que no método `findByIdAndUpdate()` nós passamos o identificador do registro e os dados a serem atualizados. Ainda, veja que `{new:true}` diz para o Mongoose retornar o registro atualizado para nossa constante `course`. Sem isso, ele retorna o curso em sua situação antes da atualização.

Observe que passamos um `id` que será informado junto com a requisição. Este `id` será informado na rota, de modo que precisamos alterar o `routes.js`:

```

// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();
// Referencia o Controller CourseController
const CourseController = require('./controllers/CourseController');
// associa as rotas ao seu método do Controller
routes.get('/courses', CourseController.index);
routes.get('/courses/:id', CourseController.show);
routes.post('/courses', CourseController.store);

```

```
routes.put('/courses/:id', CourseController.update);  
module.exports = routes;
```

Para testar esta rota, vamos no Insomnia e criamos uma nova requisição chamada “API UPDATE COURSE” como sendo do tipo PUT e informa que os dados serão passados em formato JSON.

New Request ✕

Name

API UPDATE COURSE

PUT ▾

JSON ▾

* Tip: paste Curl command into URL afterwards to import it

Create

Agora, vamos testar esta rota passando o id de um curso específico na URL. Neste caso, vou usar o id do curso de Agronomia que acabamos de criar, ou seja, 5fada804b8fe8443f7859a04. Ainda, vamos alterar alguma informação, neste caso, o nome do curso.

PUT ▾ base_url /courses/5fada804b8fe8443f7859a04 Send

JSON ▾

Auth ▾

Query

Header 1

Docs

1 {

2 "name": "Bacharel em Agronomia"

3 }

Ao realizarmos esta requisição, teremos o seguinte resultado:

200 OK

356 ms

189 B

Preview ▾

Header 7

Cookie

Timeline

1 {

2 "_id": "5fada804b8fe8443f7859a04",

3 "name": "Bacharel em Agronomia",

4 "description": "Curso superior de Agronomia",

5 "url": "http://ifrs.edu.br/bento",

6 "createdAt": "2020-11-12T21:24:20.397Z",

7 "__v": 0

8 }

1.2.12 Excluindo um registro no MongoDB

Agora, vamos criar a funcionalidade para excluir um curso específico do nosso banco de dados. Assim, altere o arquivo CourseController.js da seguinte forma:

```
// Importando as dependências
const mongoose = require('mongoose');
//Referencia o model Course
const Course = mongoose.model('Course');
// Vamos exportar um objeto com algumas funções
module.exports = {
  ...
  // Excluir um curso
  async delete(req, res){

    await Course.findByIdAndDelete(req.params.id);

    // Vamos retornar uma mensagem de sucesso sem conteúdo
    return res.send({ msg: "Registro apagado com sucesso!"});
  }
}
```

Observe que passamos um `id` que será informado junto com a requisição. Este `id` será informado na rota, de modo que precisamos alterar o `routes.js`:

```
// Importando as dependências do projeto
const express = require('express');
const routes = express.Router();
// Referencia o Controller CourseController
const CourseController = require('./controllers/CourseController');
// associa as rotas ao seu método do Controller
routes.get('/courses', CourseController.index);
routes.get('/courses/:id', CourseController.show);
routes.post('/courses', CourseController.store);
routes.put('/courses/:id', CourseController.update);
```

```
routes.delete('/courses/:id', CourseController.delete);  
module.exports = routes;
```

Para testar esta rota, vamos no Insomnia e criamos uma nova requisição chamada “API DELETE COURSE” como sendo do tipo DELETE.



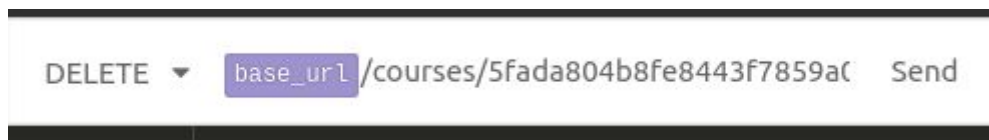
New Request ✕

Name

API UPDATE COURSE PUT JSON

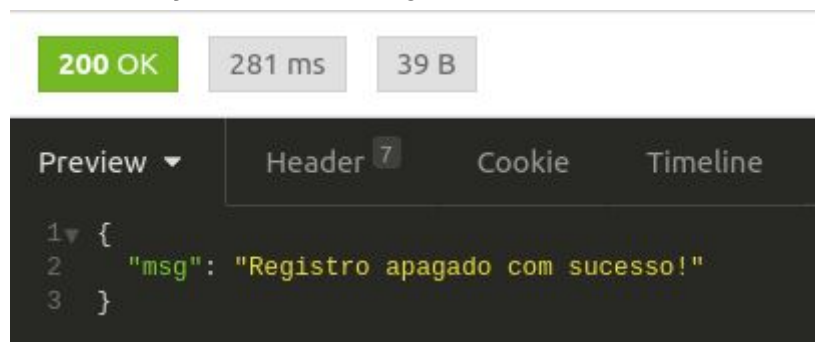
* Tip: paste Curl command into URL afterwards to import it Create

Agora, vamos testar esta rota passando o id de um curso específico na URL. Neste caso, vou usar o id do curso de Agronomia que acabamos de criar, ou seja, 5fada804b8fe8443f7859a04.



DELETE base_url /courses/5fada804b8fe8443f7859a04 Send

Ao realizarmos esta requisição, teremos o seguinte resultado:



200 OK 281 ms 39 B

Preview Header 7 Cookie Timeline

```
1 {  
2   "msg": "Registro apagado com sucesso!"  
3 }
```

Execute no Insomnia a rota “API GET COURSES” e verifique que este curso já não aparece mais no banco de dados.

1.2.13 Paginação de listas

Caso tenhamos uma requisição que retorna uma lista de muitos elementos, é recomendado fazer uma paginação dessa listagem para não comprometer a performance. Primeiro instalamos o mongoose-paginate como dependência. Assim, abra um terminal e digite o seguinte comando:

```
yarn add mongoose-paginate
```

Agora, vamos em `src/models/Course.js` e adicionamos o plugin `mongoosePaginate` em nosso Schema:

```
// Importar módulos necessários
const mongoose = require('mongoose');
const mongoosePaginate = require('mongoose-paginate');
// Define o schema do Curso
const CourseSchema = new mongoose.Schema({
  ...
});
// Adiciona o plugin mongoosePaginate em nosso schema
CourseSchema.plugin(mongoosePaginate);

// Registra o model Course em nossa aplicação informando seu
schema
mongoose.model('Course', CourseSchema);
```

Agora, vamos alterar o método `index()` do arquivo `CourseController.js` para que a nossa aplicação possa realizar a paginação:

```
...
module.exports = {
  // Vai retornar todos os cursos de nosso banco de dados
  async index(req, res) {
    //pega os parâmetros get da requisição
    const { page = 1 } = req.query;

    // retorna os cursos de nosso banco de dados
    const courses = await Course.paginate({}, { page, limit: 10 });
    // vamos retornar em formato JSON
    return res.json(courses);
  },
}
```

...

Observe que em `const { page = 1 } = req.query;` nós usamos desestruturação para pegar parâmetros GET vindos da requisição e definimos um valor padrão. Ainda, observe que o método `paginate()` recebe como primeiro parâmetro um critério de seleção (tal como o WHERE no SQL), o segundo parâmetro recebe um objeto que informa a página atual e o tamanho desta página. Para o parâmetro `page` estamos usando o Object Short Syntax, ou seja, é o mesmo que dizer `{page: page, limit: 10}`

Para testar esta rota, vamos no Insomnia e, primeiro, executamos a rota “API POST COURSES” pelo menos umas 15 vezes (para criarmos 15 cursos). Após, vamos na rota “API GET COURSES” e informamos a página que queremos pesquisar (sabendo que são mostrados 10 registros a cada página). Assim, informe o seguinte:



The image shows a screenshot of the Insomnia API client interface. It displays a GET request to the endpoint `base_url/courses?page=2`. The interface includes a dropdown menu set to 'GET', a text input field containing the URL, and a 'Send' button.

E observe o resultado:

200 OK 198 ms 1848 B

Preview Header 7 Cookie Timeline

```
38     "description": "Curso superior de Agronomia9",
39     "url": "http://ifrs.edu.br/bento",
40     "createdAt": "2020-11-12T22:11:09.868Z",
41     "__v": 0
42   },
43   {
44     "_id": "5fadb3025f7c4e5431af4178",
45     "name": "Agronomia",
46     "description": "Curso superior de Agronomia10",
47     "url": "http://ifrs.edu.br/bento",
48     "createdAt": "2020-11-12T22:11:14.210Z",
49     "__v": 0
50   },
51   {
52     "_id": "5fadb3045f7c4e5431af4179",
53     "name": "Agronomia",
54     "description": "Curso superior de Agronomia11",
55     "url": "http://ifrs.edu.br/bento",
56     "createdAt": "2020-11-12T22:11:16.477Z",
57     "__v": 0
58   },
59   {
60     "_id": "5fadb3065f7c4e5431af417a",
61     "name": "Agronomia",
62     "description": "Curso superior de Agronomia12",
63     "url": "http://ifrs.edu.br/bento",
64     "createdAt": "2020-11-12T22:11:18.633Z",
65     "__v": 0
66   },
67   {
68     "_id": "5fadb3085f7c4e5431af417b",
69     "name": "Agronomia",
70     "description": "Curso superior de Agronomia13",
71     "url": "http://ifrs.edu.br/bento",
72     "createdAt": "2020-11-12T22:11:20.791Z",
73     "__v": 0
74   },
75   {
76     "_id": "5fadb30a5f7c4e5431af417c",
77     "name": "Agronomia",
78     "description": "Curso superior de Agronomia14",
79     "url": "http://ifrs.edu.br/bento",
80     "createdAt": "2020-11-12T22:11:22.989Z",
81     "__v": 0
82   }
83 ],
84 "total": 21,
85 "limit": 10,
86 "page": "2",
87 "pages": 3
88 }
```

Observe que estamos na página 2 de um total de 3 páginas (e contém 21 registros, mas estamos mostrando apenas 10).

1.2.15 Usando CORS

Utilizamos o CORS (Cross-Origin Resource Sharing) para permitir que outros domínios acessem nossa API. Entenda que os navegadores fazem uso de uma funcionalidade de segurança chamada Same-Origin Policy: um recurso que permite que um site só pode ser chamado por outro site se estes dois estiverem sob o mesmo domínio (mesmo endereço, por ex.: meudominio.com.br). Isso limita a chamada de APIs RESTful por sites hospedados em servidores diferentes (front-end e back-end em camadas distintas). Isso porque o

navegador considera recursos do mesmo domínio somente aqueles que usam o mesmo protocolo (http ou https), a mesma porta e o mesmo endereço (mesmo subdomínios, subdominio.meudominio.com.br, por exemplo, não são considerados seguros e não funcionam). Assim, o CORS é uma especificação do W3C que permite que um site acesse recursos de outro site (ou API) mesmo estando em domínios diferentes.

Para tal, primeiramente instalaremos o CORS como dependência:

```
yarn add cors
```

Em seguida, altere o index.js:

```
// Importando as dependências do projeto
const express = require("express");
const mongoose = require('mongoose');
const requireDir = require("require-dir");
const cors = require("cors");

// Cria uma aplicação Express
const app = express();

//Permitir enviar dados para a App no formato JSON
app.use(express.json());

//Permite o uso do CORS (acesso a domínios externos da nossa API)
app.use(cors());

...
```

Pronto! Finalizamos este exercício.

Referências

- Site Oficial do MongoDB. Disponível em: <https://docs.mongodb.com/>
- Site do Mongo DB Atlas. Disponível em: <https://www.mongodb.com/cloud/atlas>
- Site Oficial do Mongoose. Disponível em: <https://mongoosejs.com/>