

JORNADA FULL STACK IMPRESSIONADOR



Aula 4

Aula 4

Aula 4

Agora vamos para a próxima fase do nosso projeto!

Até aqui, trabalhamos na construção do **Front-End** da nossa aplicação: criamos páginas, componentes, estilos e toda a parte visual que o usuário interage.

A partir de agora, entraremos em uma nova etapa muito importante:

O que vamos aprender a seguir:

- **Construir a nossa própria API:** onde vamos definir as rotas e regras para envio e recebimento de dados.
- **Conectar com um Banco de Dados:** para armazenar informações de forma segura e estruturada.
- **Conectar o Front-End com o Back-End:** fazendo com que as páginas que criamos possam se comunicar com os dados reais da aplicação.

Organização do Projeto


Para manter tudo bem organizado e seguir boas práticas, vamos separar o nosso projeto em duas partes principais:

- **front-end/** → onde ficará tudo o que já construímos até agora: componentes, páginas, estilos, etc.
- **back-end/** → onde vamos começar a construir a API, configurar o banco de dados e implementar a lógica de comunicação com o Front-End.

Com essa separação, conseguimos desenvolver e testar as partes da aplicação de forma mais clara, escalável e profissional.

O que é uma API?

API significa *Application Programming Interface*, ou em português: **Interface de Programação de Aplicações**.

 Em resumo: uma API permite que diferentes sistemas “conversem” entre si.


No nosso caso, a API será responsável por:

- Receber pedidos do Front-End (como: "quero a lista de músicas")
- Processar esses pedidos
- Buscar dados no banco de dados
- E devolver uma resposta com essas informações para o Front-End

Criando o projeto do Back-End

Agora que decidimos separar o Front do Back, precisamos **inicializar o nosso projeto de back-end** como uma aplicação Node.js. Para isso, siga o passo abaixo:

Passo 1: Criar o package.json com npm init -y

 O que é o package.json?

Esse arquivo é a “identidade” do projeto Node.js. Nele ficam registradas:

- As **dependências** que o projeto usa (como bibliotecas e frameworks)
- O nome do projeto
- Versões
- Comandos personalizados (scripts)
- E outras configurações

 Como criar?

- Abra o terminal dentro da **pasta back-end**
- Rode o seguinte comando:

```
npm init -y
```


Esse comando cria automaticamente o arquivo package.json com valores padrões (por isso o -y).

⚙️ Por que isso é importante?

- Porque permite **instalar pacotes com o npm**
- Controla **quais bibliotecas** o projeto usa
- Permite executar comandos com npm run
- Organiza o nosso ambiente de desenvolvimento

🚀 Instalando o Express

Agora vamos instalar a biblioteca mais importante da nossa API: o **Express**.

📌 O que é o Express?

O **Express** é um **framework minimalista** para Node.js que facilita:

- Criar rotas da API
- Responder requisições HTTP (GET, POST, PUT, DELETE...)
- Organizar as funcionalidades do back-end

Ele é um dos frameworks mais usados no mundo para criar APIs em Node.js.

📦 Como instalar?

No terminal, ainda dentro da **pasta bac-kend**, digite:

```
npm install express
```


✅ **Pronto!** Agora o nosso projeto back-end está inicializado com Node.js e o Express já está instalado.

Agora que já temos o Express instalado, é hora de **criar o primeiro arquivo da nossa API**. Esse arquivo será o **ponto de entrada** do nosso back-end.

📄 Nome do arquivo: server.js

Esse será o nosso **arquivo principal**. Nele, vamos:

- Importar o Express
- Criar uma instância da aplicação
- Escolher a porta que o servidor vai usar
- Colocar o servidor para “ouvir” as requisições

1. Importando o Express

```
import express from "express";
```

Aqui estamos utilizando o import para trazer a biblioteca express para dentro do nosso projeto. O Express é responsável por facilitar a criação de rotas, respostas e requisições HTTP no servidor.

⚠️ **Importante:** Para usar import ao invés de require, o seu package.json precisa conter:

```
"type": "module"
```


2. Inicializando a aplicação

Aqui estamos criando uma instância da aplicação Express. Essa instância (app) será responsável por ouvir requisições, responder ao navegador e configurar as rotas da nossa API.

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

3. Definindo a porta

Essa constante define a porta na qual o servidor vai escutar. Por convenção, usamos a 3000, mas você pode usar outra porta livre, como 5000, 8080, etc.

```
const PORT = 3000;
```

4. Iniciando o servidor

```
app.listen(PORT, () => {  
  console.log(`Servidor está escutando na porta ${PORT}`);  
});
```

Aqui dizemos: **"Ei Express, escute essa porta!"**

Quando o servidor for iniciado com sucesso, ele mostrará no terminal a mensagem "Servidor está escutando na porta 3000".

Resultado Final

```
import express from "express";

const app = express();
const PORT = 3000;

app.listen(PORT, () => {
  console.log(`Servidor está escutando na porta ${PORT}`);
});
```

Pronto! Agora temos nosso **servidor básico Express** rodando. No próximo passo, vamos aprender a criar **rotas para responder com dados do nosso backend**.

O que são os métodos HTTP?

Quando falamos de comunicação entre **cliente** (navegador ou aplicativo) e **servidor** (nosso backend), usamos **requisições HTTP**. Cada requisição tem um **método** que indica o tipo de operação que queremos fazer.

Os principais métodos são:

- **GET** – Buscar dados
- **POST** – Enviar (criar) dados
- **PUT** – Atualizar dados existentes
- **DELETE** – Apagar dados

Esses quatro métodos formam o que chamamos de:

CRUD

O termo **CRUD** vem do inglês:

Letra	Função	Método HTTP
C = Create	Criar	POST
R = Read	Ler	GET
U = Update	Atualizar	PUT
D = Delete	Deletar	DELETE

Esse é o ciclo básico de manipulação de dados em qualquer aplicação.

O que é um Endpoint?

Um **endpoint** é um endereço específico da API, geralmente uma **URL**, que responde a uma ação.

Por exemplo:

```
GET http://localhost:3000/artistas
```

Esse é um endpoint que, quando acessado, retorna a lista de artistas.

Nosso caso no projeto Spotify

No nosso projeto de réplica do Spotify, **não vamos criar, editar ou excluir músicas ou artistas**. Nosso foco é apenas **buscar informações**. Por isso:

- ♦ O método mais importante para a gente é o GET.

Vamos usar GET para:

- Buscar uma lista de músicas
- Buscar informações de um artista
- Ver detalhes de uma música específica

Conclusão

Mesmo que o projeto use principalmente o GET, é muito importante entender que uma API completa geralmente tem todos os métodos. Isso ajuda você a estar preparado para construir e consumir APIs em outros contextos.

Aula 4 – Entendendo os Métodos HTTP e o CRUD

11

Agora que você já entendeu os conceitos de GET, vamos colocar em prática!

💡 O que esse código faz?

```
app.get("/", (request, response) => {  
  response.send("Só vamos trabalhar com os endpoints '/artists' e '/songs'");  
});
```

Esse é o **primeiro endpoint** da nossa API. Vamos entender linha por linha:

📌 Linha por linha

app.get("/")

- Estamos dizendo: "**Quando o usuário acessar o caminho / usando o método GET, execute essa função**".
- Esse / é a **rota raiz** da API. É como se fosse a **página de boas-vindas** da nossa aplicação no backend.

(request, response) => { ... }

- Essa é uma **função de callback** que será executada quando alguém fizer essa requisição.
- O request (ou req) traz os dados da requisição.
- O response (ou res) é usado para **responder** ao navegador ou ao front-end.

response.send(...)

- Aqui estamos **enviando uma resposta de texto simples**.
- Essa resposta será exibida no navegador se você acessar a URL da API:

<http://localhost:3000/>



Por que criamos esse endpoint?

- Para testar se o servidor está funcionando.
- Para mostrar ao usuário que os **endpoints importantes da API são /artists e /songs**.
- É como uma placa de sinalização avisando:
“O que você quer não está aqui, mas sim em /artists ou /songs.”

Resultado no navegador:

Quando você acessar <http://localhost:3000/>, verá o texto:

```
Só vamos trabalhar com os endpoints '/artists' e '/songs'
```

Pronto! Com isso, criamos nosso primeiro endpoint.

✖ O que é a extensão JSON Viewer?

A extensão **JSON Viewer** para o Google Chrome serve para **formatar e deixar mais legível** qualquer resposta JSON que você receba no navegador.

Por padrão, quando você acessa uma rota de uma API que retorna um JSON, o navegador mostra o conteúdo **todo em uma linha**, dificultando a leitura. A extensão resolve isso.

🔍 Por que ela é útil?

Quando você acessa uma rota como:

```
http://localhost:3000/artists
```

E a resposta da API está em JSON, o navegador **sem a extensão** pode exibir algo assim:

```
[{"id":1,"name":"Artista 1","image":"url1"}, {"id":2,"name":"Artista 2","image":"url2"}]
```

Com o **JSON Viewer instalado**, essa resposta aparece formatada assim:

Ou seja:

- ✓ Fica **organizado com indentação**,
- ✓ Fica **colorido**,
- ✓ Você pode **expandir e recolher** blocos de dados (como objetos e arrays).

```
[
  {
    "id": 1,
    "name": "Artista 1",
    "image": "url1"
  },
  {
    "id": 2,
    "name": "Artista 2",
    "image": "url2"
  }
]
```





Como instalar?

- Vá até a Chrome Web Store.
- Pesquise por **JSON Viewer**.
- Clique em **"Adicionar ao Chrome"**.
- Pronto! Agora toda vez que acessar um endpoint que retorna JSON, ele será exibido de forma clara.



Dica para o projeto

Durante o desenvolvimento da sua API, você vai acessar várias rotas como /songs ou /artists. Com o JSON Viewer:

- Você verifica **se a resposta está correta**.
- Você **entende melhor a estrutura dos dados**.
- Ajuda muito quando estiver testando manualmente, antes de integrar o backend com o frontend.



JSON Viewer & Formatter for Chrome

Usar no Chrome



curlydoggo.com



Em destaque

4,0 ★ (4 notas)



Compartilhar



Vamos entender como acrescentar ao nosso servidor Express para expor dois novos endpoints: /artists e /songs.

1. Importando os dados

Antes de qualquer coisa, precisamos trazer para dentro do nosso back-end os arrays com os dados de artistas e de músicas. Para isso foram adicionadas estas linhas no topo do server.js:

```
import { artistArray } from "../front-end/src/assets/database/artists.js";  
import { songsArray } from "../front-end/src/assets/database/songs.js";
```

- **artistArray** e **songsArray** são listas estáticas de objetos JSON que já existiam no Front-End.
- Ao importá-las, nosso back-end passa a ter acesso a esses dados para enviá-los como resposta.

2. Novo endpoint /artists

- **Rota:** GET /artists
- **O que faz:** quando alguém fizer um GET em /artists, o servidor responde enviando **todo** o array artistArray.
- **Uso prático:** no Front-End, ao fazer fetch("http://localhost:3000/artists"), você receberá em JSON a lista de artistas.

```
app.get("/artists", (request, response) => {  
  response.send(artistArray);  
});
```


3. Novo endpoint /songs

- **Rota:** GET /songs
- **O que faz:** semelhante a /artists, mas retorna o array songsArray com todas as músicas.
- **Uso prático:** no Front-End, fetch("http://localhost:3000/songs") traz todas as músicas em JSON.

```
app.get("/songs", (request, response) => {  
  response.send(songsArray);  
});
```

4. Mudança de porta (opcional)

PORT alterado para **3001**:

```
const PORT = 3001;
```

Isso é útil quando:

- Você já tem outro serviço (por exemplo, seu Front-End) rodando na porta 3000.
- Evita conflito de portas, deixando o Back-End em 3001 e o Front-End em 3000.

Fluxo completo do servidor

- ✓ O que aprendemos
- **Importar dados** de outros módulos para o Back-End.
- **Criar endpoints** adicionais com `app.get()`.
- **Enviar arrays JSON** completos como resposta.
- **Usar portas diferentes** para evitar conflito entre Front-End e Back-End.

Com isso, sua API já está pronta para fornecer listas de artistas e músicas aos componentes React do Front-End!

```
import express from "express";
import { artistArray } from "../front-end/src/assets/database/artists.js";
import { songsArray } from "../front-end/src/assets/database/songs.js";

const app = express();
const PORT = 3001;

✓ app.get("/", (request, response) => {
  response.send("Só vamos trabalhar com os endpoints '/artists' e '/songs'");
});

✓ app.get("/artists", (request, response) => {
  response.send(artistArray);
});

✓ app.get("/songs", (request, response) => {
  response.send(songsArray);
});

✓ app.listen(PORT, () => {
  console.log(`Servidor está escutando na porta ${PORT}`);
});
```


🧠 O que é o MongoDB?

O **MongoDB** é um **banco de dados não relacional** (NoSQL), o que significa que ele **não armazena os dados em tabelas**, como os bancos tradicionais (como o MySQL, PostgreSQL etc). Em vez disso, ele **guarda os dados em documentos no formato JSON**, o que é muito mais parecido com os objetos que já estamos usando no nosso projeto em JavaScript.

🚀 Como funciona o MongoDB?

- Você **envia dados** (como um artista ou uma música) e ele **salva como um documento**.
 - Você pode **buscar dados** com filtros, como: "quero todos os artistas do gênero Pop".
 - Pode também **atualizar ou excluir** documentos com comandos bem simples.
- É muito usado em aplicações modernas porque é rápido, flexível e combina perfeitamente com JavaScript, já que o formato de dado é praticamente o mesmo (JSON/Objeto).

🔑 Criando uma conta no MongoDB

Para usar o MongoDB no nosso projeto, vamos precisar de um serviço chamado **MongoDB Atlas**, que é a versão online (na nuvem) do MongoDB. E o melhor: tem um plano gratuito que já é suficiente para o nosso projeto!

✨ Passos para criar uma conta:

- Acesse: <https://www.mongodb.com/cloud/atlas>
- Clique em **"Start Free"**.
- Escolha **"Sign up with Google"** (pode usar sua conta do Gmail).
- Pronto! Agora você já pode criar seu cluster gratuito e usar o banco de dados na nuvem.



📌 Por que vamos usar o MongoDB?

- Porque ele é **simples de configurar** e **funciona muito bem com JavaScript**.
- Porque **não precisamos instalar nada localmente** — ele roda na nuvem.
- E porque vamos trabalhar com **dados parecidos com objetos JSON**, o que facilita muito para quem já está aprendendo JS!

Depois de criar sua conta no [MongoDB Atlas](#) — o serviço online e gratuito do MongoDB — você será direcionado para configurar o **seu primeiro cluster**.

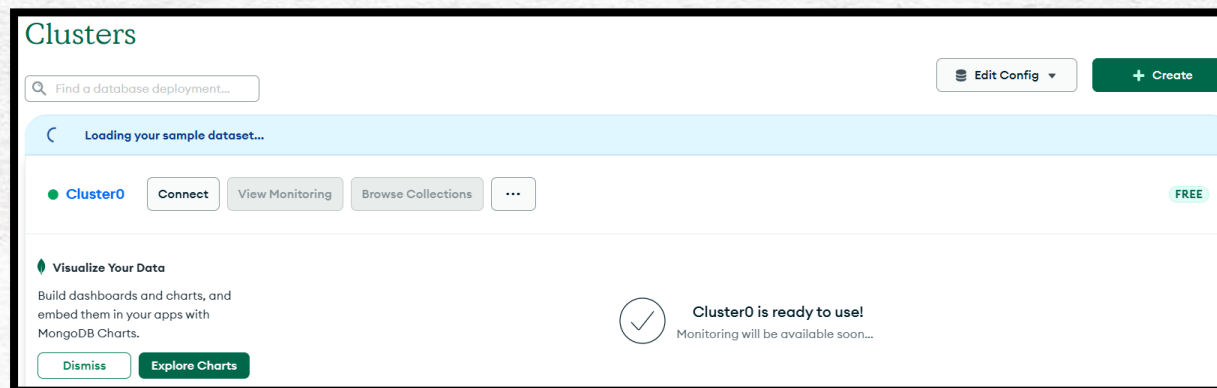
🟢 O que é um Cluster?

Um **cluster** é basicamente um grupo de servidores onde o banco de dados será hospedado. No MongoDB Atlas, **um cluster gratuito (Free Tier)** já é suficiente para pequenos projetos como o nosso.

Esse cluster gratuito:

- Permite hospedar seu banco de dados na nuvem.
- É acessível via conexão segura (com um link que você usará no backend).
- Possui um limite de uso, mas é mais que suficiente para desenvolvimento e aprendizado.

📌 **Importante:** Mesmo gratuito, você pode manter seus dados organizados e acessíveis de forma profissional.



O que são as Collections?

Dentro do MongoDB, os dados são organizados da seguinte maneira:

- O **Cluster** abriga **vários bancos de dados**.
- Cada banco de dados possui **Collections** (coleções).
- Dentro das collections, ficam os **Documentos** (em formato JSON).

Você pode pensar assim:

```
Cluster (servidor)
├── Banco de Dados
│   └── Collection (ex: songs, artists)
│       └── Documentos (ex: { name: "Imagine", artist: "John Lennon" })
```

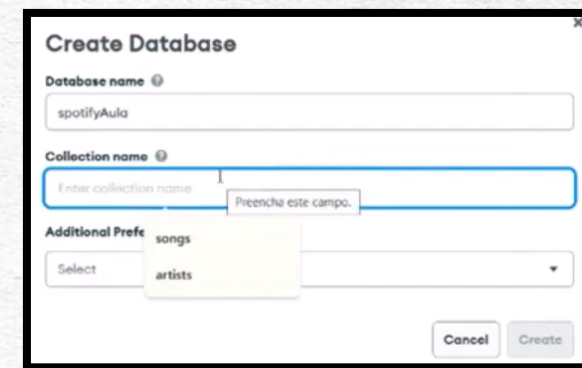
Ou seja, **collection é o equivalente a uma "tabela" em bancos de dados relacionais**, mas com mais flexibilidade, pois não exige um formato fixo de dados.

Na nossa aplicação, vamos criar uma collection chamada **artists** e outra chamada **songs**, para armazenar os dados dos artistas e das músicas que vamos consumir no frontend.

Depois de acessar o seu **cluster** no MongoDB Atlas, você terá a opção de criar seu **banco de dados** e suas **collections** diretamente pelo navegador. A imagem mostra exatamente esse processo.

✚ Etapas da imagem:

- **Database name:**
 - Aqui você define o nome do seu **banco de dados**.
 - No exemplo da imagem, o nome escolhido foi: spotifyAula.
- **Collection name:**
 - Aqui você precisa digitar o nome da sua primeira **collection** (pense nela como uma "tabela" onde seus dados serão guardados).
 - Sugestões como songs e artists aparecem porque provavelmente já foram digitadas antes ou são exemplos comuns.
 - Para o nosso projeto, **você pode começar criando a collection artists**.
 - Depois, você repetirá esse processo para criar também a collection songs.
- **Additional Preferences:**
 - Você pode deixar esse campo no padrão ou selecionar configurações adicionais (como opções de armazenamento, por exemplo).
 - Para nosso projeto, pode deixar como está.
- **Botão "Create":**
 - Após preencher o nome do banco e da primeira collection, clique em **"Create"** para criar o banco com essa collection inicial.
 - Depois disso, **você poderá adicionar novas collections dentro do mesmo banco** (como songs).

A screenshot of the 'Create Database' form in MongoDB Atlas. The 'Database name' field is filled with 'spotifyAula'. The 'Collection name' field is highlighted with a blue border and contains the placeholder text 'Enter collection name'. Below it, a dropdown menu is open, showing suggestions 'songs' and 'artists'. The 'Additional Preferences' section is visible below the dropdown. At the bottom right, there are 'Cancel' and 'Create' buttons.

Aula 4 – Trabalhando com o MongoDB

22

Depois de criar o banco spotifyAula e as collections songs e artists, o próximo passo é **inserir dados** nessas collections.

A imagem mostra a tela de **inserção manual de um documento** na collection songs.



📄 O que você está vendo?

Essa tela permite criar manualmente um **documento** (registro) com os dados de uma música.

✅ Campos preenchidos:

```
{
  _id: "67ae7b9f413e77a52c1de3e4", // Gerado automaticamente pelo MongoDB
  name: "November Rain",          // Nome da música
  artist: "Guns N' Roses"         // Nome do artista
}
```



Explicação dos elementos:

- **_id:**
 - Campo gerado automaticamente pelo MongoDB.
 - Ele é único para cada documento (registro).
 - Você não precisa criar esse campo manualmente.
- **name:**
 - Aqui você escreve o nome da música. Exemplo: "November Rain".
- **artist:**
 - Aqui vai o nome do artista que interpreta a música. Exemplo: "Guns N' Roses".

Como preencher corretamente:


- Clique em **“Insert Document”** na sua collection songs.
- Apague os campos que aparecem se quiser começar do zero (menos o _id, ele é gerado automaticamente).
- Adicione os campos name e artist, com seus respectivos valores entre aspas.
- Clique em **“Insert”** para salvar o documento.

Agora que você criou seu cluster no MongoDB Atlas, o próximo passo é **permitir que seu computador consiga se conectar a ele**. Por padrão, o MongoDB bloqueia todas as conexões externas por segurança. Por isso, precisamos liberar os endereços IP autorizados.

 O que é um IP e por que isso é importante?

- Um **endereço IP** identifica seu computador na internet.
- Ao liberar um IP no MongoDB, você está dizendo: “Este computador pode acessar meu banco de dados”.
- Sem essa liberação, mesmo que você tenha o banco configurado, **não conseguirá acessá-lo nem conectar pela aplicação**.

✓ Duas opções de liberação:

- **Liberar apenas o seu IP atual**
Ideal para ambientes seguros e pessoais. Apenas sua máquina conseguirá acessar o banco.
- **Liberar todos os IPs (0.0.0.0/0)**
Útil durante o desenvolvimento ou quando não sabemos de onde vamos acessar.
 **Atenção: essa opção é menos segura e não deve ser usada em produção.**

➡ Para configurar:

- Acesse a aba **“Network Access”** no MongoDB Atlas.
- Clique em **“+ ADD IP ADDRESS”**.
- Escolha uma das opções acima.
- Clique em **Confirm**.

Após isso, seu cluster estará pronto para ser acessado da sua aplicação. No próximo passo, vamos criar o **usuário de acesso ao banco** e depois gerar a string de conexão.

O próximo passo: conectar sua aplicação ao cluster (Connection String)

Agora que seu cluster está configurado e o acesso por IP está liberado, é hora de **gerar a string de conexão** — o código que vai permitir que sua aplicação se conecte ao banco de dados.

Etapa 1: Escolher o tipo de conexão

No painel do MongoDB Atlas, clique em:

"Connect" > "Drivers"

Essa opção permite que você conecte seu banco a uma aplicação usando uma linguagem como **Node.js, Python, Java, etc.**

Etapa 2: Escolher o driver e a versão

Na tela seguinte:

- Em **"Choose a Driver"**, selecione:
Node.js (ou outro de sua preferência).
- Em **"Version"**, selecione a versão mais recente.

Depois disso, o MongoDB Atlas irá gerar automaticamente **uma URL de conexão**, parecida com esta:

```
mongodb+srv://<username>:<password>@<cluster>.mongodb.net/
```


Etapa 3: Instalar o driver no seu projeto

Com o seu projeto Node.js iniciado, abra o terminal e digite:

```
npm install mongodb
```

Esse comando instala o pacote oficial que permite interagir com o banco MongoDB diretamente no seu código.

Etapa 4: Preencher a string de conexão

Na string gerada pelo Atlas, você deve:

- Substituir <username> pelo usuário que você criou para acessar o banco.
- Substituir <password> pela senha desse usuário.

Exemplo final:

```
mongodb+srv://admin:aula123@cluster0.mongodb.net/
```

Com isso, sua aplicação já pode iniciar uma conexão com o banco MongoDB!

Nesta aula vamos entender e criar o nosso **arquivo connect.js**, que será responsável por:

- **Estabelecer a conexão** com o banco MongoDB Atlas;
- **Exportar** a instância do banco para ser usada em toda a aplicação;
- **Utilizar JavaScript assíncrono** (async/await) para garantir que a conexão esteja pronta antes de realizarmos qualquer operação de leitura ou escrita.

1. Importando o MongoClient

- O MongoClient é o driver oficial do Node.js para se comunicar com o MongoDB.
- Ele gerencia a conexão, pooling e toda a lógica de rede.

```
import { MongoClient } from "mongodb";
```

2. Definindo a URI de Conexão

- Essa string **já vem do Atlas** (na etapa anterior você copiou do “Connect > Drivers”).
- Substitua <usuário> e <senha> pelos dados do seu usuário de banco.
- Os parâmetros retryWrites e w=majority são boas práticas de confiabilidade.

```
const URI = "mongodb+srv://<usuário>:<senha>@<cluster>.mongodb.net/?retryWrites=true&w=majority";
```


3. Criando o Cliente

- Aqui instanciamos o cliente passando a URI.
- Neste momento, **a conexão ainda não foi aberta**; criamos apenas o objeto.

```
const client = new MongoClient(URI);
```

4. Conectando-se ao Banco e Exportando o db

- Chamamos `client.db("spotifyAula")` para selecionar o **nome do banco** que criamos no Atlas.
- Ao exportar `db`, tornamos disponível em outros módulos a instância já apontada para o nosso banco.

```
export const db = client.db("spotifyAula");
```


Neste momento do nosso projeto, vamos abordar conceitos importantes de **JavaScript assíncrono**, como o uso de `async` e `await`, e como utilizá-los para trabalhar com dados do **MongoDB**.

1. O que é JavaScript Assíncrono?

JavaScript é tradicionalmente **síncrono**, ou seja, ele executa as instruções na ordem em que são escritas, uma após a outra. Mas muitas vezes precisamos realizar tarefas que **não podem ser feitas imediatamente**, como carregar dados de uma API, ler arquivos ou fazer consultas ao banco de dados.

Nesses casos, utilizamos **JavaScript assíncrono** para que o código não "trave" enquanto espera essas tarefas.

2. O que é `async` e `await`?

2.1. `async`

A palavra-chave **`async`** é usada para **definir funções assíncronas**. Quando você marca uma função com `async`, ela permite o uso do `await` dentro dela.

- Uma função `async` sempre retorna uma **`Promise`**. Ou seja, ela garante que o valor retornado será **assíncrono**.

2.2. `await`

A palavra-chave **`await`** é usada dentro de uma função **assíncrona** para **esperar que uma `Promise` seja resolvida** (ou rejeitada) antes de continuar a execução do código.

- O **`await`** só pode ser usado dentro de funções `async`.
- Ele faz o código esperar a `Promise` ser resolvida, mas sem bloquear o restante do código.

3. Como isso funciona no MongoDB?

Agora, vamos analisar o código:

```
const songCollection = await db.collection("songs").find({}).toArray();
```

3.1. db.collection("songs")

Esse trecho acessa a collection **"songs"** no MongoDB.

- Uma **collection** no MongoDB é uma "tabela" onde armazenamos documentos (dados), similar ao conceito de tabela em bancos de dados relacionais.

3.2. .find({})

A função **find({})** é utilizada para realizar uma **consulta** (query) no banco de dados.

- O **{}** dentro do find significa que estamos **buscando todos os documentos** da collection "songs".
- Esse comando retorna um **cursor** que é como um ponteiro para os dados, mas os dados ainda não foram carregados.

3.3. .toArray()

Como o comando .find() retorna um **cursor**, que é um objeto que aponta para os documentos encontrados, usamos **.toArray()** para **transformar** esse cursor em um **array** de documentos.

- Ou seja, ele vai pegar todos os documentos que atendem à nossa consulta e transformá-los em uma **lista** (array).

3.4. await

Com a palavra-chave **await**, estamos dizendo para o JavaScript **esperar até que a consulta no banco de dados termine** e retorne o resultado como um array. Sem o await, o código continuaria a execução enquanto a consulta estava sendo processada, o que poderia gerar problemas de sincronização.

Então, com o await, o código fica mais claro e fácil de entender, além de garantir que o valor de songCollection será **o resultado da consulta**, e não um "promise pendente".

4. O que acontece após a consulta?

Depois que a consulta é realizada e os dados são carregados em songCollection, utilizamos o **console.log** para **imprimir** esses dados no console.

```
console.log(songCollection);
```

O resultado de songCollection será um array contendo todos os documentos (músicas) encontrados na coleção "songs", que pode ser algo como:

```
[
  {
    _id: "1",
    name: "Song 1",
    artist: "Artist A",
    duration: "3:30"
  },
  {
    _id: "2",
    name: "Song 2",
    artist: "Artist B",
    duration: "4:00"
  },
  ...
]
```


5. Fullfilled - O que é?

Em relação a Promises, o termo **"fullfilled"** refere-se ao estado em que uma **Promise foi resolvida** com sucesso. Em outras palavras, o código assíncrono foi executado com sucesso e o valor de retorno foi obtido.

- Quando a Promise é **fullfilled**, o `await` obtém o valor dessa Promise e o executa.
- Se a Promise for **rejeitada** (por algum erro, como problemas de conexão com o banco de dados), o código **não será executado corretamente** e pode lançar um erro.

6. Resumo do Código:

- **async**: Marca uma função como assíncrona, permitindo o uso de `await` dentro dela.
- **await**: Faz com que o JavaScript espere até que a Promise seja resolvida (nesse caso, até que a consulta ao MongoDB seja concluída).
- **db.collection("songs").find({})**: Consulta o banco de dados para encontrar todos os documentos na collection "songs".
- **.toArray()**: Converte o cursor retornado pela consulta em um array de objetos.
- **console.log(songCollection)**: Exibe os resultados da consulta no console.

✓ Conclusão:

O uso de `async` e `await` torna o código assíncrono muito mais fácil de ler e entender, pois ele parece com código síncrono, mas mantendo a performance não bloqueante. Neste caso, garantimos que os dados sejam carregados corretamente do MongoDB antes de usá-los no código.

O nosso último passo, configuramos o servidor e criamos a conexão com o banco de dados usando o MongoDB. Agora, vamos tornar nosso código ainda mais dinâmico, fazendo com que as rotas busquem os dados diretamente do banco, ao invés de apenas devolver um array estático.

1. Introdução: O que vamos aprender

Até agora, nosso servidor estava funcionando com arrays estáticos. Porém, em um projeto real, queremos que ele busque os dados diretamente no banco de dados, para que as informações sejam sempre atualizadas conforme o conteúdo real da nossa base de dados.

Agora, vamos trabalhar nas rotas de **/artists** e **/songs**, fazendo com que elas acessem o banco de dados MongoDB e devolvam os dados dinamicamente.

2. Importando a Conexão com o Banco de Dados

Primeiro, vamos importar a nossa conexão com o banco de dados. No arquivo `connect.js`, já criamos a conexão com o MongoDB, então agora só precisamos trazê-la para o nosso `server.js`.

Com isso, podemos acessar o banco de dados MongoDB a partir do nosso código, sem precisar escrever as configurações novamente.

```
import { db } from "../connect.js";
```


3. Tornando as Rotas Assíncronas

Como as consultas ao banco de dados são operações assíncronas, precisamos usar a palavra-chave `async` nas nossas funções de rota. Isso permite que o código "espere" a resposta do banco de dados antes de continuar.

Exemplo de como tornamos as rotas assíncronas:

```
app.get("/artists", async (request, response) => {  
  const artists = await db.collection("artists").find({}).toArray();  
  response.send(artists);  
});
```

- **async:** garante que a função seja assíncrona, permitindo o uso do `await`.
- **await:** faz o código esperar até que a consulta ao banco de dados seja concluída.
- **db.collection("artists").find({}):** aqui, estamos pedindo todos os dados da coleção `artists`.
- **.toArray():** converte o resultado da consulta (que seria um cursor) para um array, facilitando o retorno.

4. Adicionando a Rota de songs

A lógica para as músicas (rota `/songs`) é a mesma que usamos para os artistas. Apenas mudamos o nome da coleção e a resposta. Aqui, buscamos todos os documentos da coleção `songs` e os retornamos em formato de array.

```
app.get("/songs", async (request, response) => {  
  const songs = await db.collection("songs").find({}).toArray();  
  response.send(songs);  
});
```


7. Resultado Esperado

Com essa nova configuração, agora as rotas /artists e /songs vão buscar os dados diretamente da base MongoDB, o que significa que:

- **Se você adicionar, alterar ou remover artistas ou músicas no banco de dados**, esses dados serão refletidos automaticamente nas respostas da API.
- Isso torna o nosso servidor mais dinâmico e eficiente.

8. Resumo do Processo

Passamos pelas seguintes etapas:

- **Conectamos ao banco de dados MongoDB** através do arquivo connect.js.
- **Tornamos as rotas assíncronas** usando async e await para garantir que os dados sejam carregados do banco antes de enviar a resposta.
- **Buscamos os dados diretamente das coleções** de artists e songs no MongoDB.
- **Respondemos com os dados reais** que estão armazenados no banco, permitindo que a aplicação sempre tenha as informações mais atualizadas.

Vamos aprender a criar um script que insere **vários documentos de uma vez** no banco de dados MongoDB, usando os dados que criamos previamente no nosso front-end. Essa operação é essencial quando queremos popular o banco com dados iniciais — o que chamamos de **semente** (ou *seed*).

O nome do arquivo que vamos criar é: insertMany.js.

Objetivo

- Conectar ao banco de dados MongoDB.
- Importar os dados de artistas e músicas que já criamos.
- Preparar os dados para que estejam prontos para serem inseridos no banco.
- Inserir todos os artistas e músicas de uma só vez usando insertMany.

1. Importando os dados

No início do arquivo, importamos os arrays de objetos que contêm os dados de artistas e músicas. Esses arquivos estão localizados no nosso front-end:

```
import { artistArray } from "../../front-end/src/assets/database/artists.js";  
import { songsArray } from "../../front-end/src/assets/database/songs.js";
```

Esses dados são os mesmos que usamos para exibir os cards no front-end, por isso podemos reaproveitá-los aqui para popular o banco.

Também importamos o objeto db que criamos no connect.js, para podermos usar a conexão com o banco:

```
import { db } from "../connect.js";
```

2. Preparando os dados

No banco de dados MongoDB, não devemos enviar o campo id manualmente se já temos o campo _id sendo gerado automaticamente.

Por isso, antes de fazer a inserção, criamos novos arrays sem o campo id:

```
const newArtistArray = artistArray.map((currentArtistObj) => {  
  const newArtistObj = { ...currentArtistObj };  
  delete newArtistObj.id;  
  
  return newArtistObj;  
});
```

E fazemos o mesmo para as músicas:

```
const newSongsArray = songsArray.map((currentSongObj) => {  
  const newSongObj = { ...currentSongObj };  
  delete newSongObj.id;  
  
  return newSongObj;  
});
```


✨ **Dica:** O map cria um novo array com os objetos modificados. Aqui, usamos o spread operator (...) para copiar o objeto original e em seguida usamos delete para remover a propriedade id.

4. Inserindo no banco com insertMany

Depois de preparar os dados, usamos o método insertMany() para inserir todos os artistas e músicas de uma só vez no banco de dados:

```
const responseSongs = await db.collection("songs").insertMany(newSongsArray);
const responseArtists = await db
  .collection("artists")
  .insertMany(newArtistArray);
```

Essas duas linhas fazem a inserção em massa nas coleções songs e artists.

5. Visualizando o resultado no console

Para garantir que tudo correu bem, imprimimos no console o resultado da operação:

```
console.log(responseSongs);
console.log(responseArtists);
```

O resultado vai mostrar quantos documentos foram inseridos com sucesso, além dos _id gerados automaticamente para cada um.

6. Quando rodar esse arquivo?

Esse tipo de script deve ser executado **uma única vez** para popular o banco de dados com os dados iniciais. É ideal para o ambiente de desenvolvimento, testes ou para quando for iniciar o projeto do zero com dados prontos.

💡 **Importante:** Caso execute esse script várias vezes, ele vai duplicar os dados no banco. Uma alternativa é apagar os dados antes de inserir, ou usar validações de duplicidade.

7. Conclusão

Neste arquivo insertMany.js, aprendemos a:

- Importar os dados do front-end.
- Conectar ao banco de dados.
- Preparar os dados removendo o campo id.
- Inserir todos os artistas e músicas de uma vez com insertMany.

Esse processo é essencial para preparar nosso banco com os dados que vamos utilizar na API.

Código Completo:

```
// Importa os dados dos artistas e das músicas a partir do front-end
import { artistArray } from "../../front-end/src/assets/database/artists.js";
import { songsArray } from "../../front-end/src/assets/database/songs.js";

// Importa a conexão com o banco de dados
import { db } from "./connect.js";

// Prepara os dados de artistas, removendo a propriedade 'id' que não deve ser inserida no MongoDB
const newArtistArray = artistArray.map((currentArtistObj) => {
  const newArtistObj = { ...currentArtistObj };
  delete newArtistObj.id;
  return newArtistObj;
});
```

```
// Prepara os dados de músicas, removendo a propriedade 'id'
const newSongsArray = songsArray.map((currentSongObj) => {
  const newSongObj = { ...currentSongObj };
  delete newSongObj.id;
  return newSongObj;
});

// Insere todos os documentos na coleção 'songs'
const responseSongs = await db.collection("songs").insertMany(newSongsArray);

// Insere todos os documentos na coleção 'artists'
const responseArtists = await db.collection("artists").insertMany(newArtistArray);

// Exibe o resultado das inserções no console
console.log("Músicas inseridas:", responseSongs.insertedCount);
console.log("Artistas inseridos:", responseArtists.insertedCount);
```


Agora que já temos nossos dados armazenados no MongoDB e expostos por meio de **endpoints no back-end (server.js)**, o próximo passo é fazer a **conexão do front com o back**. Ou seja, vamos fazer o **front-end buscar os dados dos artistas e músicas diretamente da nossa API**.

📁 Estrutura da pasta

Dentro da pasta front-end, vamos criar uma nova subpasta chamada api, e dentro dela um novo arquivo:

```
front-end/  
├── api/  
│   └── api.js
```

Instalar o Axios no front-end

Se ainda não instalou, abra o terminal dentro da pasta front-end e rode o comando:

```
npm install axios
```


Criar o arquivo api.js

Dentro da pasta api, crie um arquivo chamado api.js. Esse arquivo será responsável por fazer as requisições para a API do back-end.

Explicação do código

Abaixo está o conteúdo do api.js, seguido da explicação detalhada:

Conceitos importantes

- **Axios:** biblioteca que facilita fazer chamadas HTTP (como GET, POST etc.) em JavaScript.
- **Requisição GET:** usamos para “pegar” os dados da API. Ex: lista de artistas e músicas.
- **await:** como a chamada da API pode demorar, usamos await para aguardar a resposta antes de continuar.
- **Exportação:** os dados são exportados para que possamos usar o artistArray e o songsArray em outros arquivos do front-end.

```
// Fetch ou Axios
import axios from "axios";

// URL base da API
const URL = "http://localhost:3001";

// Requisições para buscar artistas e músicas
const responseArtists = await axios.get(`${URL}/artists`);
const responseSongs = await axios.get(`${URL}/songs`);

// Exportando os dados para utilizar no front-end
export const artistArray = responseArtists.data;
export const songsArray = responseSongs.data;

// console.log(responseArtists.data); // Para testar os dados no console
```


✂ Fetch vs Axios – Qual a diferença?

Quando queremos **buscar dados de uma API** no front-end, temos duas opções populares: **fetch** e **axios**.

1. Fetch

- O fetch é **nativo do JavaScript** moderno.
- Já está incluso nos navegadores, **não precisa instalar nada**.
- Sua sintaxe é um pouco mais **verboooosa** (longa) e não lida automaticamente com alguns erros.

Exemplo com fetch:

```
fetch("http://localhost:3001/artists")
  .then((res) => res.json())
  .then((data) => console.log(data))
  .catch((err) => console.error(err));
```

2. Axios

- O axios é uma **biblioteca externa** que precisa ser instalada com `npm install axios`.
- Mais **simples de usar**, **lida melhor com erros** e tem **mais recursos** prontos.
- É muito usada em projetos reais e em conjunto com o React.

Exemplo com axios:

```
import axios from "axios";

const response = await axios.get("http://localhost:3001/artists");
console.log(response.data);
```


Comparação:

Recurso	Fetch	Axios
Precisa instalar?	✗ Não	✓ Sim
Sintaxe simples?	✗ Verbosa	✓ Simples
Lida com erros automaticamente?	✗ Não	✓ Sim
Suporte a JSON automático?	✗ Precisa <code>.json()</code>	✓ Sim
Recomendado para iniciantes?	😬 Médio	✓ Sim

💡 Conclusão

Se você está começando, pode usar fetch para entender como funciona a base das requisições. Mas para projetos reais (como nosso Spotify clone), **Axios é mais prático, direto e profissional.**

O que é um Middleware?

Um **middleware** no Express é como um "meio do caminho" entre a **requisição** que chega do cliente (como o navegador ou o React) e a **resposta** enviada pelo servidor.

Ele **intercepta a requisição** para realizar algum tipo de ação antes de continuar, como:

- Autorizar um usuário
- Ler dados do corpo da requisição
- Liberar acesso entre diferentes origens
- Tratar erros

No nosso projeto, vamos usar um middleware chamado cors.

O que é CORS?

CORS significa: **Cross-Origin Resource Sharing**

Ele serve para permitir que o front-end se conecte ao back-end, mesmo que estejam em **portas diferentes** ou até **domínios diferentes**.

Por padrão, o navegador **bloqueia requisições** de origens diferentes por segurança.

O CORS **desbloqueia** isso de forma controlada.

Como instalar o CORS

No terminal do seu projeto, rode:

```
npm install cors
```

Como usar o CORS como middleware

Depois de instalar, adicione o seguinte no topo do seu arquivo server.js:

```
import cors from "cors";  
  
app.use(cors());
```

Isso permite que o navegador (nosso front-end React) consiga **fazer requisições HTTP para o servidor** sem erro.

Código Completo:

Resumo:

- **Middleware** são funções que interceptam a requisição e fazem algo com ela.
- O cors é um middleware que permite que o front-end acesse nosso back-end com segurança.
- Instalamos com npm install cors e aplicamos com app.use(cors()).

```
1  import express from "express";
2  import cors from "cors";
3  import { db } from "../connect.js";
4
5  const app = express();
6  const PORT = 3001;
7
8  app.use(cors()); // Middleware que libera o acesso ao nosso servidor
9
10 v app.get("/", (request, response) => {
11     response.send("Só vamos trabalhar com os endpoints '/artists' e '/songs'");
12 });
13
14 v app.get("/artists", async (request, response) => {
15     response.send(await db.collection("artists").find({}).toArray());
16 });
17
18 v app.get("/songs", async (request, response) => {
19     response.send(await db.collection("songs").find({}).toArray());
20 });
21
22 v app.listen(PORT, () => {
23     console.log(`Servidor está escutando na porta ${PORT}`);
24 });
25
```


Cenário atual

No FrontEnd você está fazendo isso:

```
const { name, banner } = artistArray.filter(  
  (currentArtistObj) => currentArtistObj._id === id  
)[0];
```

Aqui, você já está comparando com `_id` — o que é **correto**, pois esse campo vem do MongoDB.

O que ajustar no **BackEnd**

Quando os dados chegam como id em algum ponto do seu código, **e são usados para buscar algo no banco**, você deve garantir que está comparando com `_id`.

Além disso, no MongoDB o `_id` geralmente é do tipo `ObjectId`, então não precisa fazer `Number(id)` como seria em bancos relacionais.

Evitar Number(id)

O Mongo usa `ObjectId` como tipo padrão de `_id`, então **não transforme o id em número**, a menos que você tenha mudado a estrutura para usar IDs numéricos (o que não é comum em MongoDB).

Vamos concluir a funcionalidade do nosso componente Player, tornando-o **interativo** e com a **barra de progresso animada**, além do controle de **play** e **pause** da música.

✦ Novos conceitos que vamos aplicar:

- useRef para acessar o <audio> diretamente
- useState para controlar se o áudio está tocando
- useEffect para atualizar o tempo atual da música a cada segundo
- Funções auxiliares para converter tempo entre formato "mm:ss" e segundos

✂1. Adicionando estado e referências

Primeiro, criamos referências e estados para controlar a reprodução e progresso:

```
const audioPlayer = useRef(); // acesso direto ao <audio>
const progressBar = useRef(); // barra de progresso visual

const [isPlaying, setIsPlaying] = useState(false); // controla play/pause
const [currentTime, setCurrentTime] = useState(formatTime(0)); // tempo atual
```


🕒 2. Funções de conversão de tempo

Criamos duas funções auxiliares:

```
// Converte segundos para "mm:ss"
const formatTime = (timeInSeconds) => {
  const minutes = Math.floor(timeInSeconds / 60).toString().padStart(2, "0");
  const seconds = Math.floor(timeInSeconds % 60).toString().padStart(2, "0");
  return `${minutes}:${seconds}`;
};

// Converte "mm:ss" para segundos
const timeInSeconds = (timeString) => {
  const [minutes, seconds] = timeString.split(":").map(Number);
  return minutes * 60 + seconds;
};
```

Assim podemos usar o tempo da música vindo do banco (em string) e trabalhar com ele como número.

3. Criando a função playPause

Essa função alterna entre **tocar** e **pausar** o áudio:

```
const playPause = () => {  
  isPlaying ? audioPlayer.current.pause() : audioPlayer.current.play();  
  setIsPlaying(!isPlaying); // alterna o botão  
};
```

4. Atualizando o tempo da música

Com o `useEffect`, criamos um intervalo que roda a cada segundo enquanto a música estiver tocando:

```
1  useEffect(() => {  
2    const intervalId = setInterval(() => {  
3      if (isPlaying) {  
4        const current = audioPlayer.current.currentTime;  
5        setCurrentTime(formatTime(current));  
6  
7        progressBar.current.style.setProperty(  
8          "--_progress",  
9          (current / durationInSeconds) * 100 + "%"   
10       );  
11     }  
12   }, 1000);  
13  
14   return () => clearInterval(intervalId); // limpa o intervalo quando o componente for atualizado  
15 }, [isPlaying]);  
16
```

A barra de progresso é atualizada usando uma **CSS custom property (--_progress)**.

🎵 5. Adicionando o elemento <audio>

Por fim, o elemento de áudio fica assim:

```
<audio ref={audioPlayer} src={audio}></audio>
```

Esse ref permite que os controles (play, pause, currentTime, etc.) funcionem corretamente.

✅ Resultado final

- O botão **play/pause** alterna corretamente.
- A **barra de progresso** se move conforme o tempo.
- Os **tempos inicial e final** aparecem corretamente.
- Ao clicar em avançar/voltar, navegamos para outras músicas.

Atualizando o componente Song com o audio

No componente Song, vamos passar a propriedade audio para o componente Player.

```
<Player  
  duration={duration}  
  randomIdFromArtist={randomIdFromArtist}  
  randomId2FromArtist={randomId2FromArtist}  
  audio={audio}  
/>
```

Com isso, o componente Player vai receber o áudio da música para poder tocar. Essa informação virá da API, então o campo audio precisa estar presente nos dados da música no backend.

Parabéns pela Conclusão do Projeto!

Você chegou ao fim de mais um projeto e merece reconhecimento por todo o esforço e dedicação. Vamos relembrar tudo o que você conquistou até aqui:

O que você aprendeu

✓ **Manipular arrays e objetos**

Você aprendeu a filtrar, mapear e extrair dados de estruturas complexas com segurança e clareza.

✓ **Usar useState, useEffect e componentes funcionais**

Conseguiu montar uma interface dinâmica e interativa, controlando os estados da aplicação com React.

✓ **Passar props entre componentes**

Aprendeu a estruturar a comunicação entre componentes pais e filhos, como no caso do componente `Player`.

✓ **Conectar o frontend com dados reais**

Você buscou e manipulou dados reais, integrando o frontend com o backend usando `id` e `_id` de forma consistente.

✓ **Lógica de comparação e renderização condicional**

Implementou verificações e garantiu que o conteúdo aparecesse corretamente baseado nos dados recebidos.

✓ **Organização e boas práticas de código**

Refatorou verificações, limpou conversões desnecessárias e deixou o projeto mais limpo e escalável.