

INSTITUTO FEDERAL DE MATO GROSSO DO SUL  
*CAMPUS NOVA ANDRADINA*

NOTAS DE AULA

PROGRAMAÇÃO E TECNOLOGIAS  
PARA APLICAÇÕES SERVIDOR

Prof. Me. Luiz F. Picolo

NOVA ANDRADINA - MS

Atualizado em 14 de fevereiro de 2023

# 1 Introdução

*Professores podem abrir a porta, mas você  
precisa passar por ela por você mesmo.*

**Provérbio Chinês**

## 2 Client-Side e Server-Side

*Viva como se fosse morrer amanhã.  
Aprenda como se fosse viver para sempre.*

**Mahatma Gandhi**

Client-Side (Lado do cliente) e Server-Side (lado do servidor) são dois termos importantes envolvidos no desenvolvimento de software e web. As duas categorias de desenvolvimento têm muitas diferenças, incluindo diferentes propósitos e linguagens de programação. Assim, como nosso objetivo é trabalhar com o desenvolvimento de software, é importante que possamos entender o desenvolvimento tanto do lado do cliente quanto do lado do servidor. Assim, nossa jornada se inicia entendendo um dos lados, o do servidor, a qual fará parte do nosso curso durante nossos próximos três semestres.

### 2.1 O que é desenvolvimento do lado do cliente?

O desenvolvimento do lado do cliente é uma categoria de desenvolvimento que envolve programas executados em um cliente ou dispositivo do usuário. Os(as) desenvolvedores(as) do lado do cliente se concentram em criar a parte de um sistema com a qual o usuário pode interagir. Às vezes, o desenvolvimento do lado do cliente também é chamado, erroneamente, de desenvolvimento **front-end**, pois se concentra na parte “frontal” de um aplicativo que os usuários podem ver. Os(as) desenvolvedores(as) do lado do cliente concluem uma variedade de tarefas, incluindo:

- Criação de layouts de sites
- Projetando interfaces de usuário
- Adicionando validação de formulário
- Adicionando elementos visuais como cores e fontes

Pessoas que desenvolvem para o lado do cliente se preocupam com a experiência do usuário e usam linguagens de programação específicas. Algumas linguagens comuns do lado do cliente incluem:

- O **HTML**, que significa *Hypertext Markup Language*, é uma linguagem de marcação padrão para desenvolvimento web. HTML constrói a estrutura de um site e o renderiza em um navegador.

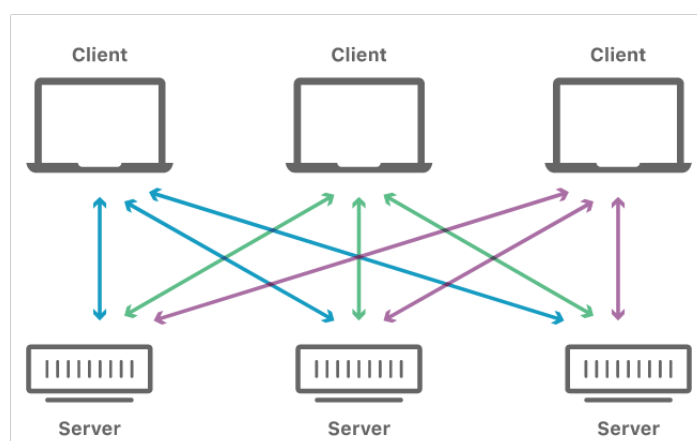
- O **CSS**, que significa **Cascading Style Sheets**, é uma linguagem de design que os(as) desenvolvedores(as) podem usar para adicionar elementos de design visual a um site codificado em HTML. Os desenvolvedores podem usar CSS para tornar seus sites visualmente atraentes nos dispositivos dos usuários.
- Já o **JavaScript** é uma linguagem de programação que os(as) desenvolvedores(as) podem usar para desenvolvimento web, aplicações web e outros propósitos. Os desenvolvedores podem usar **JavaScript** para tornar os sites dinâmicos e interativos.

Contudo, nos últimos anos, JavaScript se tornou uma linguagem que também pode ser usado do lado do servidor dando ao desenvolvedor(a) maior alcance com o conhecimento adquirido. Contudo, mesmo sendo a mesma linguagem, o objetivo acaba diferindo, por isso, vamos entender melhor como usar o JavaScript do lado do servidor.

## 2.2 O que é desenvolvimento do lado do servidor?

O desenvolvimento do lado do servidor é um categoria de desenvolvimento que envolve programas executados em um servidor. Os(as) desenvolvedores(as) do lado do servidor se concentram no desenvolvimento nos bastidores, e o desenvolvimento do lado do servidor também é conhecido, e nesse caso acertadamente, como desenvolvimento “back-end”. Esse categoria de programação é importante porque os navegadores da Web, ou clientes, interagem com os servidores da Web para recuperar informações.

**Figura 1** – Client-Side e Server-Side



Fonte: <<https://bityli.com/jwImTW>>

As tarefas comuns do lado do servidor incluem:

- Codificando sites dinâmicos

- Desenvolvendo aplicações web
- Conectando sites a bancos de dados

Desenvolvedores(as) de software, administradores(as) de banco de dados e desenvolvedores(as) da Web normalmente usam o desenvolvimento do lado do servidor. Os desenvolvedores do lado do servidor podem usar muitas linguagens de programação diferentes, tais como **PHP**, **Java**, **SQL** e também **JavaScript** a qual será a linguagem base que usaremos no decorrer do curso.

## 2.3 Lado do cliente vs. lado do servidor

Lado do cliente e lado do servidor são termos que indicam como e onde o código é executado. Alguns desenvolvedores, chamados desenvolvedores **full-stack**, sabem como usar o desenvolvimento do lado do cliente e do lado do servidor, pois ambos são importantes para o funcionamento de sites e aplicativos. No entanto, existem algumas diferenças importantes entre o desenvolvimento do lado do cliente e do lado do servidor. Algumas das principais diferenças entre os dois tipos de desenvolvimento incluem:

### 2.3.1 Onde o código é executado?

Uma grande diferença entre o desenvolvimento do lado do cliente e do lado do servidor é onde o código é executado. No desenvolvimento do lado do cliente, o código é executado no dispositivo do cliente ou do usuário. No entanto, no desenvolvimento do lado do servidor, o código é executado por meio de um servidor. É por isso que o desenvolvimento do lado do cliente também é chamado de front-end e o desenvolvimento do lado do servidor também é back-end.

### 2.3.2 Como é executado?

A maneira como os códigos são executados é outra diferença entre o desenvolvimento do lado do cliente e do lado do servidor. No código do lado do cliente, o código são simplesmente executados em um dispositivo, muitas vezes em um navegador com pouco ou nenhum acesso à memória do computador. Já do lado do servidor, no entanto, são executados em um servidor web com total acesso à memória do computador.

### 2.3.3 Quais as Finalidades?

O desenvolvimento do lado do cliente e do lado do servidor também têm propósitos diferentes. O principal objetivo do desenvolvimento do lado do cliente é criar efeitos visuais de sites, incluindo layouts, interfaces de usuário, validação de formulários e outros

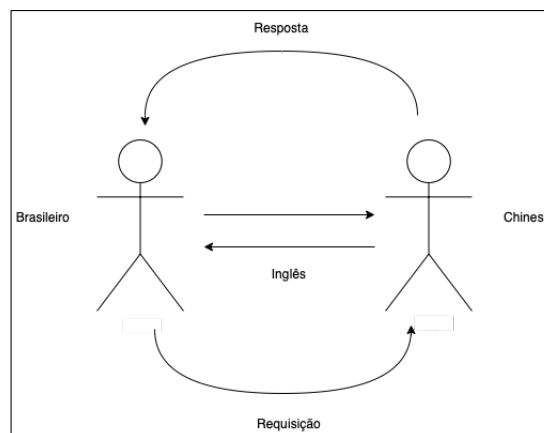
elementos visuais. O desenvolvimento do lado do servidor, no entanto, concentra-se mais no conteúdo real de uma página da Web e conclui tarefas como interagir com bancos de dados e recuperar informações de um servidor da Web.

## 2.4 Protocolo HTTP

Um protocolo é um sistema de regras que define como os dados são trafegados dentro ou entre computadores. Comunicações entre dispositivos requer que estes concordem com o formato do dado que estiver sendo trafegado. O conjunto de regras que define esse formato é chamado protocolo.

Podemos representar um protocolo como um meio de comunicação comum a dois indivíduos que não são falantes do mesmo idioma. Assim, uma das maneiras para que a comunicação aconteça e que ambos falem um idioma comum, como, por exemplo, o inglês. Clientes e servidores se comunicam trocando mensagens individuais. As mensagens enviadas pelo cliente, geralmente um navegador da Web, são chamadas de solicitações (requests), ou também requisições, e as mensagens enviadas pelo servidor como resposta são chamadas de respostas (responses).

**Figura 2** – Analogia Protocolo



Fonte: O autor

Assim, o HTTP (*Hypertext Transfer Protocol*) segue a mesma dinâmica. O HTTP é um protocolo que permite a obtenção de recursos, como documentos HTML. É a base de qualquer troca de dados na Web e um protocolo cliente-servidor, o que significa que as requisições são iniciadas pelo destinatário, geralmente um navegador da Web. Um documento completo é reconstruído a partir dos diferentes sub-documentos obtidos, como, por exemplo texto, descrição do layout, imagens, vídeos, scripts e muito mais.

### 2.4.1 Métodos de requisição HTTP

O protocolo HTTP define um conjunto de métodos de requisição responsáveis por indicar a ação a ser executada para um dado recurso. Embora esses métodos possam ser descritos como substantivos, eles também são comumente referenciados como *HTTP Verbs* (Verbos HTTP). O HTTP implementa vários verbos, porém, os mais usados são:

- **GET**: O método GET solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados.
- **POST**: O método POST é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.
- **PUT**: O método PUT substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.
- **DELETE**: O método DELETE remove um recurso específico.

## 2.5 Exercícios de fixação

1. Qual a diferença entre “Client-Side (lado do cliente) e Server-Side (Lado do servidor)”?
2. O que é um protocolo e, por que ele é usado na comunicação?
3. Explique o que é um Verbo HTTP e como eles são utilizados na comunicação.
4. O JavaScript é uma linguagem Client-Side ou Server-Side? Explique sua resposta.

## 3 Servidor Web com NodeJS

*As raízes do estudo são amargas,  
mas seus frutos são doces.*

**Aristóteles**

Ao acessar e visualizar uma página em seu navegador, como, por exemplo [<https://www.ifms.edu.br>](https://www.ifms.edu.br), você está fazendo uma solicitação (request) a outro computador na rede (ou internet), que em resposta (response), fornece a você a página Web. O computador que está se comunicando por meio da sua solicitação executa uma aplicação que é o **servidor Web**. Um servidor Web recebe solicitações HTTP de um **cliente**, como seu navegador, e fornece uma resposta HTTP, como uma página HTML.

O **Node.js** permite que os desenvolvedores utilizem o JavaScript para escrever o código do lado do servidor (Server-Side), embora ele seja tradicionalmente usado no navegador para escrever o código do lado do cliente (Client-Side). Assim, vamos aprender como desenvolver servidores Web usando o módulo `http` que está incluído no Node.js que poderá retornar dados em JSON, arquivos CSV e páginas Web em HTML.

### 3.1 Criando um servidor HTTP básico

Para iniciarmos a configuração vamos criar um servidor que retorna um texto sem formatação ao usuário. Primeiro, precisamos configurar um ambiente de programação acessível para fazer nossos exercícios. Para tanto, vamos criar e acessar uma pasta chamada **primeiro-servidor**:

```
1 mkdir primeiro-servidor
2 cd primeiro-servidor
```

Dentro do diretório vamos criar um arquivo com o nome de **index.js** e nele iniciaremos as primeiras linhas para a implementação do nosso servidor HTTP. Começaremos carregando o módulo **http**, padrão em todas as instalações do Node.js. Adicione a linha seguinte ao **index.js**:

```
1 const http = require("http");
```



Nosso próximo passo definiremos duas constantes, o **host** e a **porta** em que nosso servidor se associará. O **host** nada mais é do que o “endereço” que faremos o acesso ao servidor e a **porta** pode ser qualquer número entre 1024 até 40.000. Contudo, por padrão, vamos usar a porta 8080.

```
1  const http = require("http");
2
3  const host = 'localhost';
4  const porta = 8080;
```

Como mencionado anteriormente, os servidores Web aceitam solicitações de navegadores e de outros clientes. Podemos interagir com um servidor Web ao digitar um nome de domínio. O valor **localhost** é um endereço privado especial que os computadores utilizam para se referir a eles mesmos. Normalmente, ele é equivalente ao endereço **IP interno 127.0.0.1** e está disponível apenas para o computador local, ou seja, não está disponível para nenhuma rede local da qual participamos ou para a Internet. Por fim, definimos a porta, que é o número que representa o “local” no computador onde o servidor está sendo executado.

Ao vincularmos nosso **servidor** a este **host** e **porta**, conseguiremos acessar nosso servidor ao visitarmos `http://localhost:8080` em um navegador local.

Vamos adicionar uma função especial que em Node.js, chamamos **resposta**. Esta função foi criada para processar uma solicitação HTTP de entrada e retornar uma resposta HTTP. A função deve ter dois argumentos: um **objeto de solicitação** e um **objeto de resposta**. O objeto de solicitação capta todos os dados da solicitação HTTP que chegam. O objeto de resposta é usado para devolver respostas HTTP para o servidor.

```
1  const http = require("http");
2  const host = 'localhost';
3  const porta = 8080;
4
5  const resposta = function (req, res) {
6    res.writeHead(200);
7    res.end("Meu primeiro servidor!!!");
8  };
```

Por fim, podemos criar nosso servidor e usar nossa função de resposta:

```
1  const http = require("http");
2  const host = 'localhost';
3  const porta = 8080;
4
5  const resposta = function (req, res) {
6    res.writeHead(200);
7    res.end("Meu primeiro servidor!!!");
8  };
9
10 const server = http.createServer(resposta);
11 server.listen(porta, host, function() {
12   console.log(`Servidor sendo executado em http://${host}:${porta}`);
13 });
```

Caso deseje disponibilizar o acesso para todos na rede, basta adicionar o endereço **0.0.0.0** no lugar no **localhost**.

```
1  const http = require("http");
2  const host = '0.0.0.0';
3  const porta = 8080;
4
5  const resposta = function (req, res) {
6    res.writeHead(200);
7    res.end("Meu primeiro servidor!!!");
8  };
9
10 const server = http.createServer(resposta);
11 server.listen(porta, host, function() {
12   console.log(`Servidor sendo executado em http://${host}:${porta}`);
13 });
```

## 3.2 Retornando tipos diferentes de conteúdo

Para podermos retorna uma categoria de conteúdo diferente do que é apresentado, devemos adicionar o tipo de conteúdo que queremos. Para tanto, vamos adicionar um

cabeçalho informado ao navegador a forma com que ele deve tratar o conteúdo que chega-lhe por meio da resposta.

```
1 res.setHeader("Content-Type", "text/html");
2 res.end(`<html><body><h1>This is HTML</h1></body></html>`);
```

O código completo, adicionado indentação e tags para inserir css, ficará da seguinte forma.

```
1 const http = require("http");
2 const host = '0.0.0.0';
3 const porta = 8080;
4
5 const resposta = function (req, res) {
6   res.setHeader("Content-Type", "text/html");
7   res.writeHead(200);
8   res.end(`
9     <html>
10      <head>
11        <style>
12          body {
13            background: #000
14          }
15        </style>
16      </head>
17      <body>
18        <h1>This is HTML</h1>
19      </body>
20    </html>
21  `);
22 };
23
24 const server = http.createServer(resposta);
25 server.listen(porta, host, function() {
26   console.log(`Servidor sendo executado em http://${host}:${porta}`);
27 });
```

## 3.3 Gerenciador de pacotes para Node.js

Existem diversas formas para instalar pacotes em diversas linguagem e ambientes de desenvolvimento. A mais utilizada é por meio da utilização de aplicativos específicos, conhecidos como **gerenciadores de pacotes**. **Pacotes** são arquivos que contém bibliotecas e seus arquivos de configuração, como também, todas as dependências (outros pacotes) requeridos para a instalação.

Várias linguagem possuem seus respectivos gerenciadores, e, no caso do Nodejs os mais usados são o **Yarn** e o **NPM**. O Yarn é um gerenciador de pacotes para Node.js que se concentra em velocidade, segurança e consistência. Ele foi criado originalmente para resolver alguns problemas com o popular gerenciador de pacotes NPM. Embora os dois gerenciadores de pacotes tenham convergido em termos de desempenho e recursos, o Yarn continua popular, especialmente no mundo do desenvolvimento React.

### 3.3.1 Usando o Yarn

O Yarn tem muitos subcomandos, mas precisamos apenas de alguns para começar. Vejamos os primeiros subcomandos desejamos usar.

```
1 yarn --help // Mostra a ajuda
2 yarn install --help // Mostra a ajuda para os comandos de instalação
3 yarn init // Inicia um projeto
4 yarn add [pacote] // Instala um pacote
5 yarn add [pacote] --dev // instala um pacote em desenvolvimento
6 yarn install // Instala as dependências
```

### 3.3.2 Iniciando um novo projeto

Para iniciar um projeto criamos um diretório e, interno a ele, executamos o comando os seguintes comandos:

```
1 mkdir nome-projeto
2 cd nome-projeto
3 yarn init // ou como recomendo o npm init
```

Após o comando o Yarn executará algumas perguntas, Responda com os dados desejados e por fim será gerado um arquivo **package.json**. O package.json é um arquivo

de configuração utilizado para estipular e configurar **dependências** do seu projeto (quais outros pacotes ele vai precisar para ser executado) e **scripts** automatizados.

Agora, para adicionar uma dependência, vamos ao site do Yarn <<https://yarnpkg.com/>> ou Npm <<https://www.npmjs.com/>> e buscar o pacote que desejamos. No caso, vamos usar um para validar CPFs, o qual pode ser encontrado em <<https://yarnpkg.com/package/cpf>>.

Seguindo a documentação, para instalarmos o pacote devemos executar o seguinte comando:

```
1 yarn add cpf
```

Assim, "magicamente", o pacote é instalado em nosso computador dentro de um diretório no projeto chamado **Node Modules** e o arquivo **package.json** é atualizado com a nova dependência, inclusive, em sua última versão.

```
1 ...  
2 "dependencies": {  
3   "cpf": "^2.0.1"  
4 }  
5 ...
```

O **Node Modules** é um diretório que pode ser deletado e recriado quando necessário. Para tanto, devemos apenas executar o comando abaixo que todas as dependências serão reinstaladas.

```
1 yarn install // ou apenas yarn
```

Para utilizar o pacote instalado, será necessário verificar a documentação. Contudo, o instalado anteriormente, pode ser usado da seguinte forma. Em um arquivo **index.js** adicione os seguintes comandos:

```
1 const CPF = require('cpf');  
2 console.log(CPF.format('11144477735'))  
3 console.log(CPF.isValid('111.444.777-35'));
```

Após isso podemos executar o comando **node index.js** ou adicionar um executado no arquivo **package.json** da seguinte forma:

```
1  "scripts": {  
2    "start": "node index.js",  
3    "test": "echo \"Error: no test specified\" && exit 1"  
4  }
```

Por fim, executar o comando **yarn start** que obteremos a seguinte saída:

```
1  $ yarn start  
2  yarn run v1.22.19  
3  $ node index.js  
4  
5  111.444.777-35  
6  true  
7  
8  Done in 1.02s.
```

Agora, para executarmos o próximo exercício, vamos usar um pacote chamado **nodemon**. Para tanto, instale o pacote **yarn add nodemon -dev** e observe como foi modificado o **package.json**. Por fim, adicione o seguinte comando ao **script** do seu arquivo para que, a cada mudança feita, o servidor seja reiniciado.

```
1  "scripts": {  
2    "start": "nodemon index.js"  
3  }
```

### 3.4 Exercício de fixação

1. Crie uma página pessoal contendo informações sobre você, tais como, data de nascimento, nome completo, endereço, email, etc. Formate ela usando CSS e disponibilize na rede para que outros possam acessá-la.
2. Usando comentários, explique, com detalhes, o que faz cada linha do código criado no exercício 1.

3. Liste, ao menos, 5 pontos negativos relacionados ao desenvolvimento do código da questão 1. Pense em pontos que dificultaram o desenvolvimento e que podem ser melhorados.
4. Baseando-se nas explicações dadas em aula, explique o que é um **Host** e o que é uma **Porta**. Caso deseje, use analogias para reforçar sua resposta.

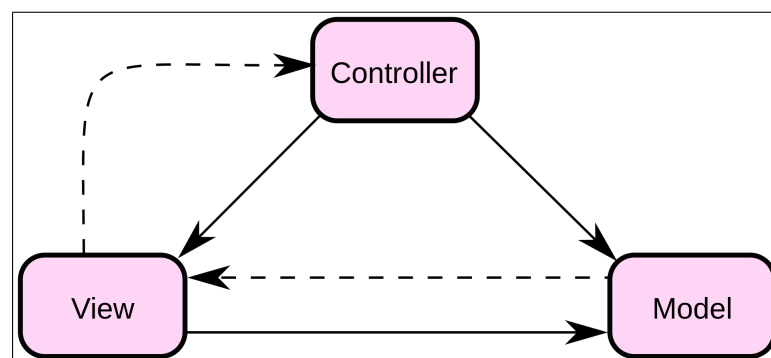
### 3.5 Padrão em design de software MVC

MVC (Model-View-Controller) é um padrão em design de software comumente usado para implementar **interfaces** de usuário, **dados** e **lógica de controle**. Ele enfatiza uma separação entre a lógica de negócios do software e a exibição. Esta “separação de interesses” proporciona uma melhor divisão do trabalho e uma melhor manutenção.

As três partes do padrão de projeto de software MVC podem ser descritas da seguinte forma:

- **Modelo:** Gerencia dados e lógica de negócios.
- **View:** lida com layout e exibição.
- **Controlador:** roteia comandos para as peças do modelo e da visualização.

Figura 3 – MVC



Fonte: <<https://pt.wikipedia.org/wiki/MVC>>

#### 3.5.1 O que faz cada camada?

As camadas do **Padrão de Design MVC** possuem cada uma sua responsabilidade. Tais responsabilidades são listadas abaixo:

- **Controladores - Controllers:** Um Controller representa as classes que conectam o modelo e a visão e é usado para comunicação entre as classes no **modelo** e na **visão**.

- **Visualizações - Views:** Uma View é uma coleção de classes que representam os elementos na interface do usuário (todas as coisas que o usuário pode ver e responder na tela, como botões, caixas de exibição e assim por diante)
- **Modelos - Models:** Um modelo representa a estrutura lógica subjacente de dados em um aplicativo de software e a classe de alto nível associada a ele. Este modelo de objeto não contém nenhuma informação sobre a interface do usuário.

Assim, para podermos usar a estrutura MVC em nossos projetos, devemos usar uma ferramenta que possibilite as conexões listas acima. Para tanto, vamos usar um específico, o **ExpressJS**.

## 3.6 Express Web Framework

Alguns problemas surgem quando vamos desenvolver. No exercício anterior, para criarmos uma página simples, conseguimos resolver facilmente. Contudo, caso queiramos uma organização ou apenas criar uma página a tarefa é bem mais complicada. Assim, para resolver esses problemas, que não são apenas nossos, mas comuns durante o desenvolvimento profissional, foram criadas ferramentas cujo objetivo é facilitar a codificação e reduzir a repetição de código. Uma dessas é o **Framework ExpressJS**.

Assim um **Framework** tem como principal objetivo resolver problemas recorrentes com uma abordagem genérica, permitindo ao desenvolvedor focar seus esforços na resolução do problema em si, e não ficar reescrevendo software [Para iniciantes].

Express é um popular framework web estruturado, escrito em JavaScript que roda sobre o ambiente NodeJS em tempo de execução. Este módulo explica alguns dos principais benefícios deste framework, como configurar o seu ambiente de desenvolvimento e como executar tarefas comuns de desenvolvimento e implantação da web.

### 3.6.1 Instalando o ExpressJS

Para realizar a instalação do ExpressJS vamos seguir o Guia presente em <https://expressjs.com/pt-br/starter/installing.html>. Assumindo que já tenha instalado o Node.js, crie um diretório para conter o seu aplicativo, e torne-o seu diretório ativo iniciando um projeto usando o **Yarn** como feito em 3.3.1.

Para instalar o express usando o seguinte comando:

1

```
yarn add express
```



### 3.6.2 Exemplo Hello World

Para nosso primeiro exemplo vamos continuar usando o Guia citado anteriormente <<https://expressjs.com/pt-br/starter/hello-world.html>>. No diretório criado crie um arquivo chamado **index.js** e inclua o seguinte código:

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => {
6    res.send('<h1>Hello World!</h1>')
7  })
8
9  app.listen(port, () => {
10    console.log(`Servidor escutando na porta ${port}`)
11  })
```

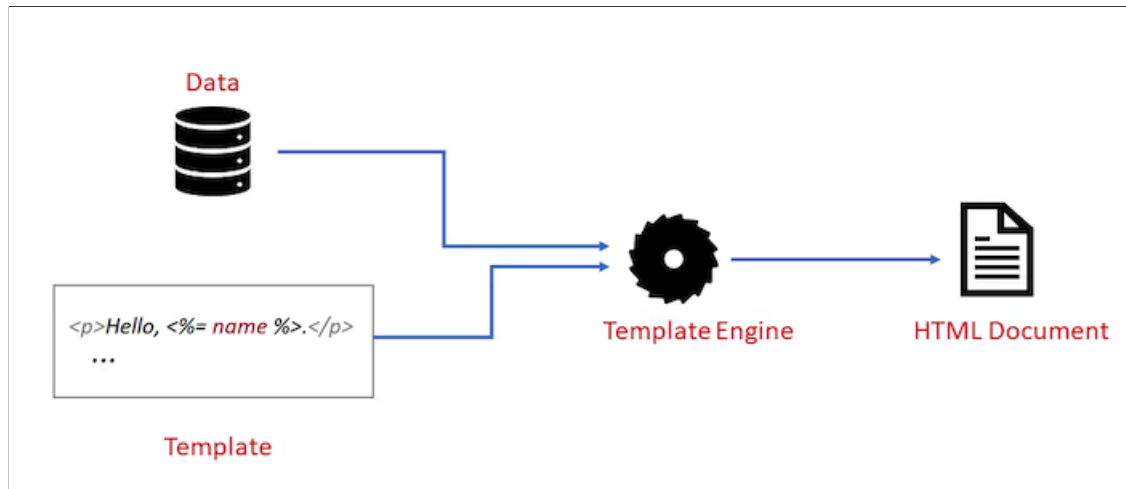
Também modifique o **package.json** adicionado o **Script** para iniciar o projeto usando o **nodemon**. Não se esqueça de instalar o **nodemon** em desenvolvimento.

## 3.7 Usando Embedded Javascript Templating (ejs)

Se você estiver escrevendo um aplicativo *Server Side* em Node.js e quiser enviar HTML de volta para os clientes com conteúdos dinâmicos, você deve encontrar uma maneira de misturar ou interpolar os dados. O EJS (Embedded JavaScript Templating) é um dos mecanismos de modelo mais populares para JavaScript. Como o nome sugere, ele nos permite incorporar código JavaScript em uma linguagem de modelo que é então usada para gerar HTML.

Um mecanismo de modelo é um software projetado para combinar modelos de dados para produzir, no nosso caso, código HTML real.

Figura 4 – Mecanismos de modelo



Os mecanismos de modelo lidam com a tarefa de interpolar dados em código HTML enquanto fornecem alguns recursos (como parciais em EJS) que seriam difíceis de replicar concatenando strings.

### 3.7.1 Instalando e configurando o EJS

Para instalar e configurar o EJS em nosso projeto, devemos, primeiramente, instalar a biblioteca que fornecerá o suporte em nossa aplicação e criar um diretório, na raiz do projeto, com o nome de **views**.

```
1 yarn add ejs
2 npm install ejs
```

Agora, com a dependência já instalada, vamos configurar a nossa aplicação para usar o EJS. Isso tudo será feito dentro do nosso arquivo **index.js**.

```
1 const express = require('express')
2 const path = require('path');
3 const app = express();
4 const port = 3000
5
6 app.use(express.static('public'));
7 app.set('view engine', 'ejs');
8
9 app.get('/', (req, res) => {
```

```
10     res.render('index');
11   })
12
13   app.listen(port, () => {
14     console.log(`Servidor executando na porta ${port}`)
15   })
```

E, no diretório **views**, criaremos um arquivo que será o utilizado para a exibição com o nome de **index.ejs**.

### 3.7.2 Passando dados para serem renderizados

Lembre-se de que nosso objetivo é combinar dados com modelos. Podemos fazer isso passando um segundo argumento para `res.render`. Este segundo argumento deve ser um objeto, que estará acessível no arquivo de modelo EJS.

Para tanto, adicione o seguinte código no arquivo **index.js**

```
1   const express = require('express')
2   const path = require('path');
3   const app = express();
4   const port = 3000
5
6   app.use(express.static('public'));
7   app.set('view engine', 'ejs');
8
9   const usuario = {
10     nome: 'Luiz',
11     sobrenome: 'Picolo',
12   }
13
14   app.get('/', (req, res) => {
15     res.render('index', {
16       usuario
17     })
18   })
19
20   app.listen(port, () => {
```

```
21 console.log(`Servidor executando na porta ${port}`)  
22 })
```

E no arquivo **index.ejs** inclua o objeto da seguinte forma:

```
1 <h1>Olá, eu sou o <%= usuario.nome %></h1>
```

### 3.7.3 Exercício de fixação

1) Crie um rota de artigos. Depois, crie uma lista de postagens (como se fosse um blog) e liste-as ao acessar a rota artigos. Lembre-se de criar o arquivo EJS como feito anteriormente.

## 3.8 Requisições de objetos: req.params e req.query

Até o momento, todos os dados que trafegaram em nossos estudos foram apenas do Servidor para o Cliente. Agora, a ideia é transmitir dados do Cliente para o Servidor. Para isso, vamos usar as três expressões descritas acima, req.body, req.query e req.params, que fazem parte do **Objeto de solicitação no Express.js** e são usados, pelo cliente, para enviar dados para o servidor. A princípio, vamos trabalhar com dados vindos das URLs.

### 3.8.1 req.params

O **req.params** tem com objetivo capturar propriedades anexadas ao URL, ou seja, parâmetros de rota nomeados. Você prefixa o nome do parâmetro com dois pontos (:) ao escrever suas rotas. Por exemplo, vamos capturar o parâmetro da URL abaixo:

**http://localhost:3000/artigos/123**

Assim, nossa rota artigos deve ser modificada da seguinte forma:

```
1 app.get('/artigos/:numero', (req, res) => {  
2   console.log(req.params.numero)  
3 })
```

Caso houvesse mais parâmetros, poderíamos fazer da seguinte forma:

**http://localhost:3000/artigos/123/titulo**

```
1 app.get('/artigos/:numero/:titulo', (req, res) => {  
2   console.log(req.params.numero)  
3   console.log(req.params.titulo)  
4 })
```

### 3.8.2 req.query

Já o `req.query` é usado principalmente para pesquisa, classificação, filtragem, paginação, etc. Digamos, que desejássemos obter a mesma URL anterior mas usando o `query`, ou, que tivéssemos 10, 15 parâmetros. A escrita da rota ficaria comprometida pela dificuldade de entendimento. Assim, podemos rescrever tanto a **URL** como a **rota** ela usando o **`req.query`**.

**`http://localhost:3000/artigos?numero=123&titulo=titulo-do-artigo`**

**Observe com atenção**

Perceba que, após a rota é usando um ponto de interrogação, e, para separar os demais parâmetros, usando o `&` também conhecido com E comercial.

```
1 app.get('/artigos', (req, res) => {  
2   console.log(req.query.numero)  
3   console.log(req.query.titulo)  
4 })
```

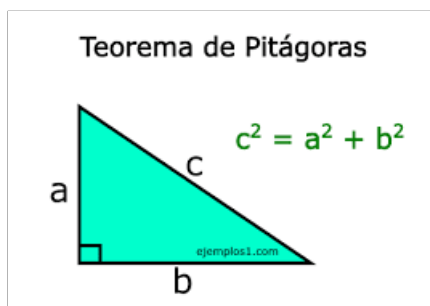
### 3.8.3 Exercício de Fixação

1) A partir das URLs abaixo, crie suas respectivas rotas e, usando os conceitos aprendidos na aula anterior, apresente os parâmetros em um arquivos EJS.

- `http://localhost:3000/girafa?altura=15&nome=juju`
- `http://localhost:3000/girafa/15/juju`
- `http://localhost:3000/noticias?id=123`
- `http://localhost:3000/noticias/123`

2) Vamos usar as rotas para calcular o Teorema de Pitágoras. Use a seguinte URL para criar a rota e os parâmetros:

`http://localhost:3000/pitagoras?a=4&b=6&c=3`



## 4 Banco de dados

*Persistência é a irmã gêmea da excelência.  
Uma é a mãe da qualidade, a outra é a mãe do tempo.*

**Marabel Morgan**

O armazenamento de dados é uma faceta crucial em qualquer aplicação de computador. Ele permite a captura e preservação de informações para análise futura ou para alimentar a geração de resultados esperados. Há uma ampla variedade de opções de armazenamento de dados, incluindo arquivos (texto e/ou binários), bancos de dados relacionais, sistemas NoSQL e outros tipos de soluções.

Contudo, apesar da existência de diversas formas de armazenar dados, o objetivo deste texto é demonstrar o uso de um banco de dados relacional juntamente com o NodeJS. Para tal, será utilizada a linguagem padrão de consulta, a SQL (Structured Query Language).

A linguagem de consulta SQL é um padrão amplamente utilizado em muitos sistemas de gerenciamento de banco de dados relacional (também conhecidos como SGDBs). No entanto, a implementação do SQL pode variar em relação ao padrão oficial. Essas variações normalmente não são um problema quando você trabalha apenas com uma plataforma específica de banco de dados. No entanto, quando você precisa mudar entre diferentes provedores de banco de dados ou mesmo entre versões diferentes do mesmo provedor, essas diferenças podem se tornar importantes. A mudança para sistemas não relacionais aumenta ainda mais a complexidade do problema. Portanto, é importante considerar cuidadosamente o projeto antes de decidir qual banco de dados utilizar.

Já o banco escolhido, inicialmente, será o Postgresql. Esse SGDB, segundo seus desenvolvedores, é poderoso sistema de banco de dados objeto-relacional de código aberto com mais de 30 anos de desenvolvimento ativo que lhe rendeu uma forte reputação de confiabilidade, robustez de recursos e desempenho<sup>1</sup>.

As notas de aula, não se preocupam com detalhes de instalação e/ou configuração, pois se acredita que, nesse momento do curso, o estudante já está capacitado para tal tarefa. Contudo, caso o leitor queira montar seu ambiente de trabalho, segue as tecnologias base que serão utilizadas nesse primeiro capítulo.

- NodeJS - versão 16+
- Express - versão 4+

---

<sup>1</sup> Para mais detalhes, acesse: <<https://www.postgresql.org>>

- Postgres - 14+ (será usado a versão online disponível em: <<https://www.elephantsql.com>>)

## 4.1 Conceitos básicos sobre Banco de dados Relacionais

Segundo Elmasri et al. 2005, um banco de dados é uma coleção de dados relacionados. Os dados são fatos que podem ser gravados e que possuem um significado implícito. Por exemplo, considere nomes, números telefônicos e endereços de pessoas que você conhece. Esses dados podem ter sido escritos em uma agenda de telefones ou armazenados em um computador. Essas informações são uma coleção de dados com um significado implícito, consequentemente, um banco de dados.

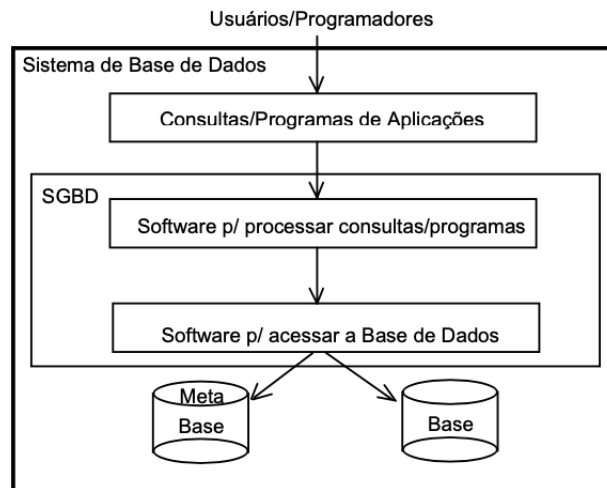
Ainda, segundo os autores:

Um banco de dados pode ser gerado e mantido manualmente ou pode ser automatizado (computadorizado). Por exemplo, um catálogo de cartões bibliotecários é um banco de dados que oferece a possibilidade de ser criado e mantido manualmente. Um banco de dados computadorizado pode ser criado e mantido tanto por um grupo de aplicativos escritos especialmente para essa tarefa como por um sistema gerenciador de banco de dados [Elmasri et al. 2005, p. 04].

Já, um Sistema Gerenciador de Banco de dados, (SGBD) é uma coleção de programas que permite aos usuários criar e manter um banco de dados. O SGBD é, portanto, um sistema de software de propósito geral que facilita os processos de definição, construção, manipulação e compartilhamento de bancos de dados entre vários usuários e aplicações [Elmasri et al. 2005].

Assim, a base de dados e o software de gerenciamento da base de dados compõem o chamado Sistema de Base de Dados. A Figura 5 apresenta um esquema genérico de um Sistema de Banco de Dados em sua interação com seus usuários [Takai, Italiano e Ferreira 2005].

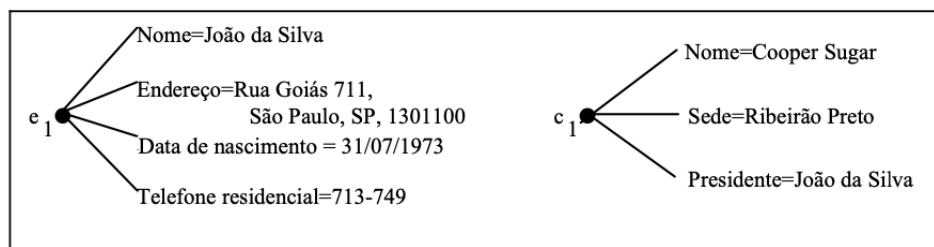




**Figura 5** – Sistema de Banco de Dados Fonte: [Takai, Italiano e Ferreira 2005]

#### 4.1.1 Modelo Entidade-Relacionamento: Entidades e atributos

O objeto básico que o Modelo Entidade-Relacionamento (MER) representa é a **entidade**. Uma entidade é algo do mundo real que possui uma existência independente. Uma entidade pode ser um objeto com uma existência física - uma pessoa, carro ou empregado - ou pode ser um objeto com existência conceitual - uma companhia, um trabalho ou um curso universitário. Já, cada entidade tem propriedades particulares, chamadas atributos, que a descrevem [Takai, Italiano e Ferreira 2005].



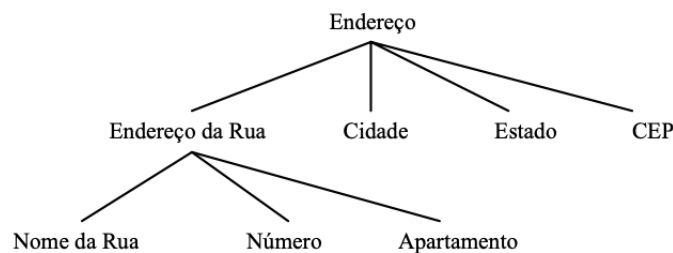
**Figura 6** – Exemplo de entidades e seus respectivos atributos Fonte: [Takai, Italiano e Ferreira 2005]

Alguns atributos podem ser divididos em subpartes com significados independentes. Por exemplo, **Endereço**, como visto na Figura 7, pode ser dividido em **Endereço da Rua**, **Cidade**, **Estado** e **CEP**. Um atributo que é composto de outros atributos mais básicos é chamado **composto**. Já, atributos que não são divisíveis são chamados **simples** ou **atômicos**.

Tipos de atributos:

- **Compostos**: Podem ser divididos em partes menores;
- **Simples**: Eles não são divisíveis;

- **Monovalorados:** Possuem apenas um valor;
- **Multivalorado:** Um ou mais valores para o mesmo;
- **Armazenado:** Em geral todos os atributos são armazenados;
- **Derivado:** Alguns atributos podem ter uma relação entre si;
- **Nulo:** Quando valores podem não ser aplicados.



**Figura 7** – Exemplo de atributos compostos Fonte: [Takai, Italiano e Ferreira 2005]

Um banco de dados inclui uma coleção de conjuntos de entidades, cada qual contendo um número de entidades do mesmo tipo. A figura abaixo mostra parte de um banco de dados que consiste em dois conjuntos de entidades: clientes e contas [Sanches 2005]

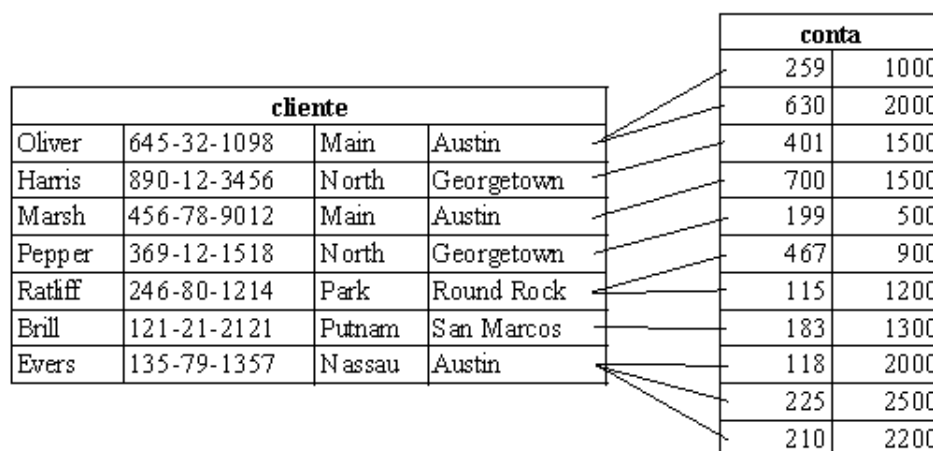
cliente				conta	
Oliver	645-32-1098	Main	Austin	259	1000
Harris	890-12-3456	North	Georgetown	630	2000
Marsh	456-78-9012	Main	Austin	401	1500
Pepper	369-12-1518	North	Georgetown	700	1500
Ratliff	246-80-1214	Park	Round Rock	199	500
Brill	121-21-2121	Putnam	San Marcos	467	900
Evers	135-79-1357	Nassau	Austin	115	1200
				183	1300
				118	2000
				225	2500
				210	2200

**Figura 8** – Conjuntos de entidades cliente e conta. Fonte: [Sanches 2005]

#### 4.1.2 Relacionamentos e conjunto de relacionamentos

Um relacionamento, segundo Sanches 2005, é uma associação entre diversas entidades. Por exemplo, podemos definir um relacionamento que associa o **cliente Harris** à **conta 401**. Isto especifica que Harris é um cliente com conta bancária número 401.

Assim, o relacionamento ContaCliente é um exemplo de um conjunto de relacionamentos **binários** - isto é, ele envolve dois conjuntos de entidades.



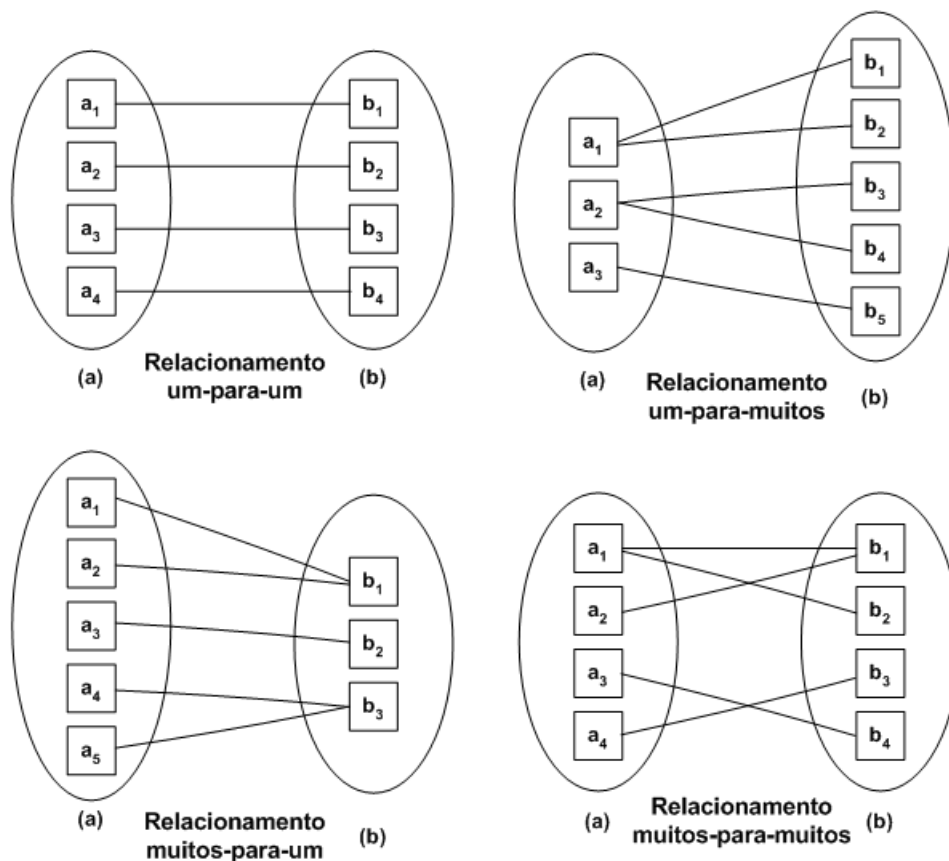
cliente				conta	
Oliver	645-32-1098	Main	Austin	259	1000
Harris	890-12-3456	North	Georgetown	630	2000
Marsh	456-78-9012	Main	Austin	401	1500
Pepper	369-12-1518	North	Georgetown	700	1500
Ratliff	246-80-1214	Park	Round Rock	199	500
Brill	121-21-2121	Putnam	San Marcos	467	900
Evers	135-79-1357	Nassau	Austin	115	1200
				183	1300
				118	2000
				225	2500
				210	2200

**Figura 9** – Relacionamento envolvendo entidades: **Cliente** e **Conta**. Fonte: [Sanches 2005]

Os relacionamentos podem ser:

- **Um-para-um:** uma entidade A está associada no máximo a uma entidade B e uma entidade B está associada no máximo a entidade de A;
- **Um-para-muitos:** uma entidade A está associada a qualquer número de entidades de B. Uma entidade de B, entretanto, pode estar associada no máximo a uma entidade de A;
- **Muitos-para-um:** uma entidade A está associada no máximo a uma entidades de B. Uma entidade de B, entretanto, pode estar associada a qualquer número de entidades de A;
- **Muitos-para-muitos:** uma entidade A está associada a qualquer número de entidades de B e uma entidade de B está associada a qualquer número de entidades de A.

Abaixo, na Figura 10, pode-se visualizar as formas de relacionamento, como as entidades podem se relacionar para mais dar sentido a informação desejada.



**Figura 10** – Tipos de relacionamento. Fonte: [Sanches 2005]

#### 4.1.3 Linguagens de SGBD's relacionais

A linguagem de SGBD's é essencialmente SQL, que no que lhe concerne é subdividido em grupos de comandos, conforme a função de cada comando.

e, resumindo,

- **DDL** = Data Definition Language - Linguagem de definição de dados, possibilita a definição ou descrição dos objetos do BD.
- **DML** = Data Manipulation Language - Linguagem de manipulação de dados, que suporta manipulação (acesso e alteração).
- **DCL** = Data Control Language - Linguagem de controle de dados, possibilita a determinação das permissões que cada usuário terá sobre os objetos do BD (permissão de criação, consulta, alteração, etc.).

Para este estudo, consideraremos apenas as DML que serão utilizadas no próximo capítulo. A Linguagem de Manipulação de Dados (DML - **Data Manipulation Language**) é um conjunto de comandos que determinam quais valores estão presentes nas tabelas em qualquer momento e como serão manipulados. No SQL, por exemplo, são instruções DML:

- SELECT - consulta (seleção)
- UPDATE - atualização
- DELETE - exclusão
- INSERT - inclusão

De forma simples, para ser realizada uma seleção de alguns dados no banco, pode-se fazer da seguinte forma:

```
1  select * from ENTIDADE
2  select * from ENTIDADE where ATRIBUTO = VALOR DESEJADO
3  select NOME_DA_COLUNA FROM ENTIDADE_1 inner join ENTIDADE_2
4  ON ENTIDADE_1.coluna_nome = ENTIDADE_2.coluna_nome;
```

Existem varias formas de se fazer uma seleção, atualização, entre outros. Contudo, em nossas práticas diárias, faremos algumas delas considerando o conhecimento adquirido pelo estudante no decorrer de várias disciplinas relacionadas a Banco de dados.

#### 4.1.4 Exercícios de fixação

1. Defina os seguintes termos: dados, informação, banco de dados, SGBD e sistema de banco de dados.
2. Quais podem ser as diferentes categorias de usuários finais de banco de dados? Discuta as atividades principais de cada um.
3. Cite e explique algumas vantagens do uso de Banco de dados.
4. Quais são as categorias de linguagens de SGBD? Para que serve cada uma?
5. Uma escola possui várias turmas. Uma turma contém vários professores na qual um professor pode lecionar aulas em mais de uma turma. Uma turma possui sempre aulas na mesma sala, mas uma sala pode estar associada a várias turmas.
  - a) Liste as possíveis entidades que podem ser encontradas
  - b) Liste os possíveis atributos das entidades encontradas
  - c) Relacione, de forma simples, as entidades.

## 5 Criação e acesso a banco de dados relacional usando NodeJS

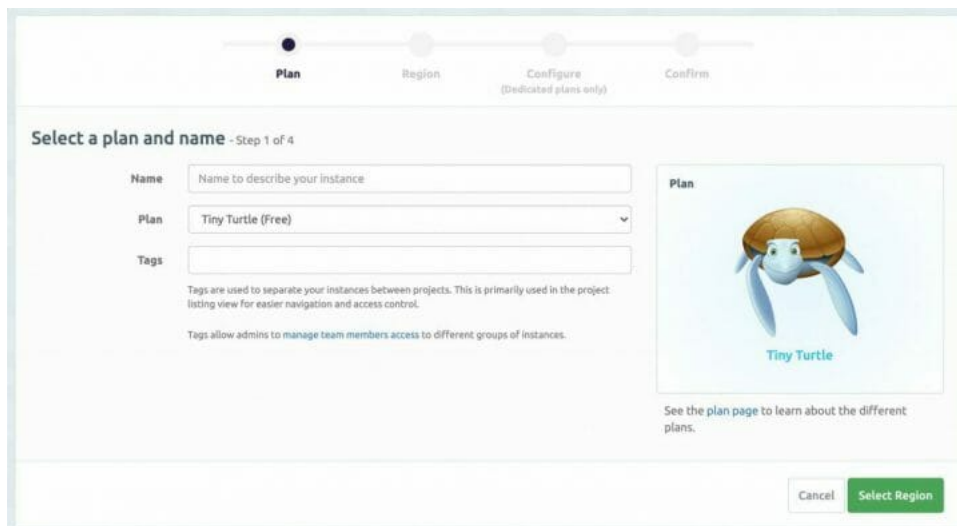
*Hoje você acha cansativo, porém mais tarde receberá a recompensa por todo esse tempo que passou estudando.*

**Anônimo**

Em nossa aula vamos aprender como podemos usar Node.js com PostgreSQL. Apesar de MongoDB e outras bases não relacionais serem uma das escolhas mais comuns com Node, muitos desenvolvedores, conhecem e usam PostgreSQL e não querer abrir mão de seu conhecimento a respeito. Também é importante deixar claro que partimos do pressuposto que você já saiba usar, minimamente, um banco de dados relacional como o Mysql por exemplo, e que também já saiba o básico de Node.js. Assim, vamos focar nossa aula em apenas ligar os pontos, ou seja, o Node.js com nosso banco dados PostgreSQL.

### 5.1 Usando o ElephantSQL

Para evitar o passo de instalação do banco em nossa maquina local, vamos usar um serviço online que poderá se comunicar com nossas aplicações desenvolvidas no Replit. Para tanto, vamos usar o plano *free* da <elephantsql.com> chamado de **Tiny Turtle**.

The image shows a web form for creating a new ElephantSQL instance. At the top, there is a progress bar with four steps: 'Plan' (selected), 'Region', 'Configure (Dedicated plans only)', and 'Confirm'. The main heading is 'Select a plan and name - Step 1 of 4'. Below this, there are three input fields: 'Name' with a placeholder 'Name to describe your instance', 'Plan' with a dropdown menu showing 'Tiny Turtle (Free)', and 'Tags' with a placeholder. Below the 'Tags' field, there is a small text block explaining that tags are used for separating instances between projects. To the right of the input fields, there is a 'Plan' section featuring a cartoon turtle illustration and the text 'Tiny Turtle'. Below the turtle, there is a link to 'See the plan page to learn about the different plans.' At the bottom right of the form, there are two buttons: 'Cancel' and 'Select Region'.

**Figura 11** – Criação da primeira instância

Após adicionar um nome, selecione a região (não é necessário alterar), revise os dados e clique em *Criar Instância*. Quando terminar, clique no nome da instância criada

você terá acesso às credenciais de acesso e todas as seções que utilizaremos para manipular nosso banco.

Para criar nossa primeira tabela, clique em *Browser* e será aberto o *SQL Browser*. Nele vamos digitar nossos comandos SQL.

### 5.1.1 Comando CREATE TABLE no PostgreSQL

O comando **SQL CREATE TABLE** é utilizado para definir uma nova tabela, inicialmente vazia (sem nenhum registro), no esquema de banco de dados atual. A tabela criada pertence ao usuário que executa o comando. Vejamos a sintaxe básica para a criação de uma tabela no postgres:

```
1 CREATE TABLE [IF NOT EXISTS] nome_tabela (  
2     nome_coluna tipo_dados [COLLATE colecao] constraint,  
3     nome_coluna tipo_dados constraint,  
4     nome_coluna tipo_dados constraint,  
5     ...,  
6     [FOREIGN KEY chave_estrangeira REFERENCES coluna]  
7     [ON DELETE acao ] [ ON UPDATE acao ]  
8 )
```

Vamos criar uma tabela **autores** que irá conter os campos **id nome**, **sobrenome** e **datanascimento**

```
1 CREATE TABLE autores (  
2     id SERIAL CONSTRAINT pk_id_autor PRIMARY KEY,  
3     nome varchar(30) NOT NULL,  
4     sobrenome varchar(40) NOT NULL,  
5     datanascimento date  
6 );
```

Agora, vamos criar a tabela de livros, incluindo os relacionamentos com as demais tabelas por meio do uso de chaves estrangeiras.

```
1 CREATE TABLE livros (  
2     id SERIAL CONSTRAINT pk_id_livro PRIMARY KEY,  
3     nome varchar(50) NOT NULL,
```

```
4      autor integer NOT NULL,  
5      editora integer NOT NULL,  
6      datapublicacao date,  
7      preco money,  
8      FOREIGN KEY (autor) REFERENCES autores (id) ON DELETE CASCADE  
9  );
```

Outra forma de estabelecer esse relacionamento é simplesmente indicar a referência de uma coluna em sua própria declaração, o que a torna uma chave primária. Neste exemplo poderíamos escrever simplesmente:

```
1  CREATE TABLE livros (  
2      id SERIAL CONSTRAINT pk_id_livro PRIMARY KEY,  
3      nome varchar(50) NOT NULL,  
4      autor integer REFERENCES autores(id) NOT NULL,  
5      editora integer NOT NULL,  
6      datapublicacao date,  
7      preco money  
8  );
```

### 5.1.2 Exercícios de fixação

**Crie as seguintes tabelas com seus respectivos relacionamentos:**

**Empregado** (id:Serial, matricula:integer, cpf:varchar, nome:varchar, endereço:varchar, cep:integer)

**Projeto** (id:Serial, nom:varchar, verba:money)

**Alocação** (id:Serial, projeto:foreingkey, empregado:foreingkey)

1. Escreva o código antes de adicionar ao banco
2. Valide o código com o professor
3. Adicione o código ao banco

## 5.2 Usando Node.js com o PostgreSQL

Para iniciarmos nossa jornada com Node.js e PostgreSQL, vamos criar nosso primeiro projeto usando o **express.js**. Para maiores detalhes, acesse o link para relem-



brar sobre o funcionamento do *Express Generator* <<https://expressjs.com/pt-br/starter/generator.html>>. Se, estiver usando o <<https://replit.com>> use o comando abaixo

```
1 npx express --view=ejs .
```

Contudo, para facilitar nosso trabalho, vamos realizar um *fork* do seguinte repositório <<https://github.com/luizpicolo/skeleton-nodejs-express-ejs.git>> e importá-lo no **replit**.

### 5.2.1 Conexão com o banco de dados

Assim, após todo o processo ter ocorrido sem erros, vamos configurar a conexão com o banco. Crie um arquivo **db.js** na raiz do seu repositório. Não podemos criando conexões infinitas no banco pois isso, além de ser lento, é inviável do ponto de vista de infraestrutura. Usaremos aqui um conceito chamado *connection pool*, onde um objeto irá gerenciar as nossas conexões, para abrir, fechar e reutilizar conforme possível. Vamos guardar este único pool em uma variável global, que testamos logo no início da execução para garantir que se já tivermos um *pool*, que vamos utilizar o mesmo.

```
1 let connect = function() {  
2   if (global.connection){  
3     return global.connection.connect();  
4   }  
5  
6   const { Pool } = require('pg');  
7   const pool = new Pool({  
8     connectionString: 'URL PARA O BANCO DE DADOS'  
9   });  
10  
11   global.connection = pool;  
12   return pool.connect();  
13 }  
14  
15 module.exports = { connect };
```

Para que o código acima funcione corretamente, devemos instalar a dependência **pg**, executando o comando abaixo.

```
1 npm i pg --save
```

### 5.2.2 Criando nosso primeiro modelo para acesso ao dados

Para que possamos testar a conexão com o banco em nossos modelos, vamos criar um modelo de exemplo chamado **Autor** e invocar o código de conexão da seguinte forma.

```
1 const db = require("../db");
2
3 class Autor {
4
5 }
6
7 module.exports = Autor;
```

Usando apenas a chamada acima, já possuímos um modelo que pode obter os dados necessários por meio de uma conexão com o banco de dados.

### 5.2.3 As quatro operações básicas em um banco de dados

Nas manipulações de registros realizadas diretamente em banco de dados ou em plataformas desenvolvidas no padrão *RESTful*, o conceito **CRUD** estabelece o modelo correto no manuseio desses dados.

CRUD representa as quatro principais operações realizadas em banco de dados, seja no modelo relacional (SQL) ou não-relacional (NoSQL), facilitando no processamento dos dados e na consistência e integridade das informações.

A sigla CRUD significa as iniciais das operações create (criação), read (leitura), update (atualização) e delete (exclusão). Essas quatro siglas tratam a respeito das operações executadas em bancos de dados relacional (SQL) e não-relacional (NoSQL). Essas operações pertencem ao agrupamento chamado de *Data Manipulation Language (DML)*, utilizado na linguagem Structured Query Language (SQL)<sup>1</sup>.

#### 5.2.3.1 Create

A operação de criação de um registro em uma tabela é realizada pelo comando INSERT. Exemplo:

<sup>1</sup> Para mais detalhes acesse: <<https://blog.betrybe.com/tecnologia/crud-operacoes-basicas/>>

```
1 class Autor {
2   static async insert(data){
3     const connect = await db.connect();
4     const sql = 'insert into autores(nome, sobrenome, datanascimento)
5       ↪ values ($1, $2, $3)';
6     const values = [data.nome, data.sobrenome, data.datanascimento];
7     return await connect.query(sql, values);
8   }
}
```

#### 5.2.3.2 Read

A operação de consulta de um ou mais registros em uma tabela é realizada pelo comando SELECT. Exemplo:

```
1 static async select(){
2   const connect = await db.connect();
3   return await connect.query('select * from clientes');
4 }
```

#### 5.2.3.3 Update

Comando utilizado para a atualização de um ou mais registros de uma tabela. Exemplo:

```
1 static async update(id, data){
2   const connect = await db.connect();
3   const sql = 'UPDATE clientes SET nome=$1, idade=$2, uf=$3 WHERE id=$4';
4   const values = [data.nome, data.idade, data.uf, id];
5   return await connect.query(sql, values);
6 }
```

#### 5.2.3.4 Delete

Comando utilizado para a exclusão de registro (s) de uma tabela. Exemplo:

```
1  static async delete(id){
2      const connect = await db.connect();
3      const sql = 'DELETE FROM clientes where id=$1;';
4      return await connect.query(sql, [id]);
5  }
```

#### 5.2.4 Invocando os métodos nas rotas

Para que possamos invocar os métodos de manipulação de dados do modelo, precisamos criar uma rota. Para tanto, crie uma nova rota utilizando o **express** e adicione a seguinte rota para que seja feita a seleção dos dados.

```
1  var express = require('express');
2  var router = express.Router();
3  // Invocando o modelo Autor
4  const Autor = require("../models/autor");
5
6  /* Listando os usuários e apresentando um Json */
7  router.get('/', async function(req, res, next) {
8      const data = await Autor.select();
9      res.json(data.rows);
10 });
11
12 module.exports = router;
```

## 6 Mapeamento objeto-relacional (ORM)

*É melhor você tentar algo, vê-lo não funcionar e aprender com isso, do que não fazer nada.*

**Mark Zuckerberg**

Criar um banco de dados, ou mais formalmente o Sistema Gerenciador de Banco de Dados (SGBD), é uma tarefa complexa presente em todos os projetos. Escolher qual será utilizar em meio a tantos como: **mysql**, **sqlserver**, **oracle**, **sqlite**, **postgre**, **mongodb** e **etc**, ou, qual se adapta melhor ao projeto é uma decisão que deve ser tomada com cautela.

Outro fato é a forma com que os dados são tratados em cada um. Por exemplo, para um campo **id** que deve ser auto-incrementado, no Postgres usando o seguinte código SQL:

```
1 CREATE TABLE Pessoas (  
2     id SERIAL NOT NULL,  
3     nome varchar  
4 );
```

Já no Mysql, o mesmo código seria escrito da seguinte forma:

```
1 CREATE TABLE Pessoas (  
2     id int NOT NULL AUTO_INCREMENT,  
3     nome varchar  
4 );
```

Logo, caso houvesse uma mudança, nosso código teria que ser alterado para satisfazer a nova base de dados com todas as suas diferenças. Assim, pensando nas possíveis mudanças que o projeto pode ter durante sua vida útil, foram desenvolvidas algumas técnicas para facilitar a vida dos desenvolvedores. Uma que iremos comentar é o Object Relational Mapping (ORM) ou Mapeamento Objeto Relacional. Para diminuir a complexidade, já que o ORM torna o banco de dados mais próximo da arquitetura de classe, removendo os comando SQL de vista, para que possamos focar em “Qual é o fluxo que minha aplicação deve seguir” e deixando de lado “Qual é a query que eu deveria usar aqui?”. Então, nesse aspecto, iremos abordar um pouco sobre o que são ORM, suas práticas e exemplos usando JavaScript.

## 6.1 ORM: o que é?

Como já citei anteriormente um ORM é a sigla em inglês para Mapeamento Objeto Relacional e consiste em manter o uso de orientação a objetos e um pouco do conceito de non-query, pois serão raros os momentos nos quais teremos que escrever uma linha de código SQL [Muramatsu 2020]. Um ORM é uma solução completa para resolver a incompatibilidade de **impedância** entre objetos de programa e tabelas de banco de dados, promovendo bancos de dados relacionais com recursos de orientação a objetos [Song e Gao 2012].

Outro fato muito importante e curioso sobre os ORM é que eles operam como um agente de banco de dados, sendo possível através de pouquíssimas mudanças, utilizar o mesmo código para mais de um banco de dados. Não importa se ele está em Mysql, SqlServer ou até mesmo Oracle. Ele consegue agir da mesma forma em alguns bancos de dados, você só precisa mudar o driver de conexão e está pronto para uso. Neste conceito é importante lembrar que cada uma de nossas tabelas são vistas como uma instância de uma classe, tendo suas características declaradas diretamente na sua classe “esquema”. Quando trabalhamos com esse tipo de esquema sempre teremos um arquivo de configuração, responsável por fornecer os dados para que o componente de ORM possa se comunicar com o banco e aplicação. Uma outra questão que pode causar dúvidas é como gerar o banco de dados através dessas classes que comentamos anteriormente. Bom veremos isso na prática mais a abaixo [Muramatsu 2020].

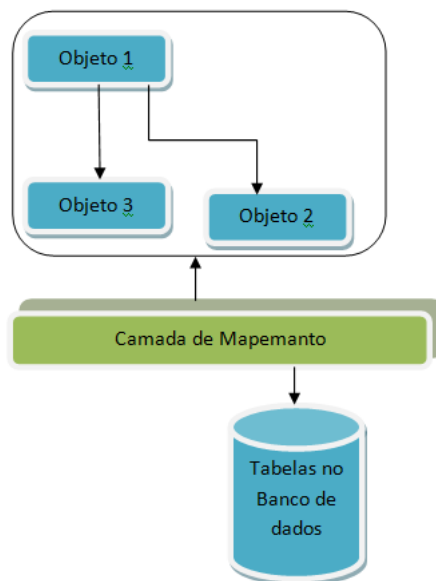
Além da manipulação, outra preocupação, quando se retrata o cenário de desenvolvimento de software com banco de dados, é a segurança. Ai utilizar um ORM (consolidado) adquirimos a possibilidade de direcionar os esforços do projeto para a inovação mantendo a confiabilidade, segurança, e tradição dos bancos de dados relacionais.

## 6.2 ORM: como funciona?

Para facilitar a aplicação desta técnica surgiram os frameworks ORM, como por exemplo, o Hibernate para Java, o ActiveRecord para Ruby, e o que usaremos para nossos projeto o Sequelize para NodeJS (JavaScript). Desta forma, esses frameworks se localizam em uma camada intermediária entre a lógica da sua aplicação e o seu SGDB [Magazine 2020].

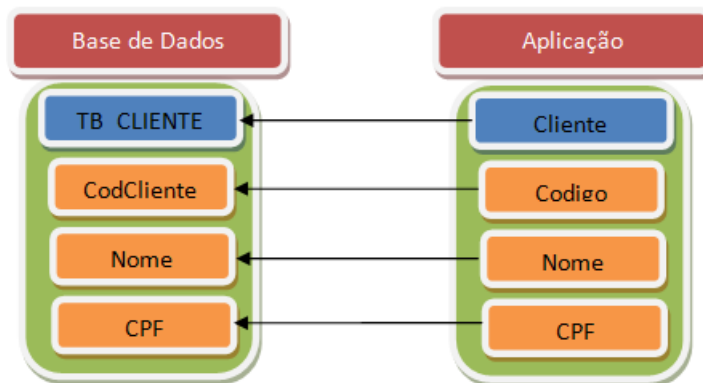
O framework passa a receber as solicitações de interação com o SGDB através de objetos de sua aplicação e gera automaticamente todo o SQL necessário para a operação solicitada, nos poupando do trabalho de escrita e manutenção deste SQL e abstraindo o uso do SGDB, fazendo com que nos preocupemos apenas com nosso modelo de objetos. Além disso, o framework já trata as variações de tipos de dados existentes entre nossa

aplicação e nosso SGDB [Magazine 2020].



**Figura 12** – Ilustração do mapeamento Fonte: [Magazine 2020]

O ORM funciona através do mapeamento das características da base de dados para os objetos de nossa aplicação. O primeiro conceito chave é traçar um paralelo entre Classe x Tabela e Propriedade x Coluna. O ORM nos permite informar em que tabela cada classe será persistida e em que coluna do SGDB cada propriedade ficará armazenada.



**Figura 13** – Ilustração de uma classe para uma tabela Fonte: [Magazine 2020]

### 6.3 Configurando o frameworks ORM Sequelize

O Sequelize possui um utilitário de linha de comando chamado **Sequelize CLI** que auxilia em diversas atividades incluindo funcionalidades para nos ajudar com **migrations**, as quais veremos mais a frente.

De forma geral, vamos iniciar um novo projeto Node usando as instruções mais básicas. Primeiro, crie um diretório e, a partir dele, inicie um novo projeto.

```
1 mkdir novo-projeto
2 cd novo-projeto
3 npm init -y
```

Logo após, ainda no diretório, adicione as bibliotecas que usaremos

```
1 npm install sequelize pg --save
2 npm install --save-dev sequelize-cli
```

Agora, vamos inicializar nosso projeto com o **Sequelize-CLI**, rodando o comando abaixo.

```
1 npx sequelize-cli init
```

O resultado será o retorno e a criação de alguns diretórios listados abaixo.

```
1 Created "config/config.json"
2 Successfully created models folder at ...
3 Successfully created migrations folder at ...
4 Successfully created seeders folder at ...
```

O diretório **Models** nós já conhecemos. Contudo, os novos criados tem como função conectar ao banco e criar tabelas e adicionar dados. Vamos ver cada um:

- **config/config.json** é o local onde colocaremos os dados para conexão, como usuário, senha, host, etc. Nesse arquivo, um dos parâmetros é o **dialect**. Nele devemos colocar o SGBD desejado. Por padrão, ele é gerado como **mysql**, por isso, vamos mudar para **postgres**/
- **migrations** é o local no qual criaremos nosso código equivalente ao SQL para a criação das tabelas, porém, usando apenas JavaScript.
- **seeders** será onde criaremos o código que semeará os dados no banco.



## 6.4 Criando nosso primeiro modelo e migração

Vamos criar um cenário bem simples no qual criaremos uma nova tabela no banco de dados. Para criar uma **migration**, podemos usar o seguinte comando no console, que baixa e executa o **Sequelize CLI** com o comando de criação de migration.

```
1 npx sequelize-cli model:create --name usuario --attributes nome:string
```

O comando acima irá criar uma pasta **migrations** no seu projeto (se ela não existir) e dentro dela nosso primeiro arquivo de **migration**, idêntico ao abaixo.

```
1 'use strict';
2 module.exports = {
3   async up(queryInterface, Sequelize) {
4     await queryInterface.createTable('usuarios', {
5       id: {
6         allowNull: false,
7         autoIncrement: true,
8         primaryKey: true,
9         type: Sequelize.INTEGER
10      },
11      nome: {
12        type: Sequelize.STRING
13      },
14      createdAt: {
15        allowNull: false,
16        type: Sequelize.DATE
17      },
18      updatedAt: {
19        allowNull: false,
20        type: Sequelize.DATE
21      }
22    });
23  },
24  async down(queryInterface, Sequelize) {
25    await queryInterface.dropTable('usuarios');
26  }
27 };

```

Toda migração possui um **up** e um **down**, referente ao script de migration (criação) e o rollback (desfaz o que foi criado) da mesma. Isso permite que em caso de necessidade seja desfeita a migration, permitindo a gestão das alterações do banco de dados com muito mais detalhe.

Também será criado um modelo com o código abaixo

```
1  'use strict';
2  const { Model } = require('sequelize');
3  module.exports = (sequelize, DataTypes) => {
4    class usuario extends Model {
5      /**
6       * Helper method for defining associations.
7       * This method is not a part of Sequelize lifecycle.
8       * The `models/index` file will call this method automatically.
9       */
10     static associate(models) {
11       // define association here
12     }
13   }
14   usuario.init({
15     nome: DataTypes.STRING
16   }, {
17     sequelize,
18     modelName: 'usuario',
19   });
20   return usuario;
21 };
```

Assim, para enviar os dados ao banco, devemos executar dois comandos, o primeiro para criar o banco e outro para criar as tabelas do banco selecionado.

```
1  npx sequelize-cli db:create // Não deve ser usado para o ElephantSQL
2  npx sequelize-cli db:migrate
```

## 6.5 Adicionado o Frameworks Express ao Projeto

Para que possamos utilizar os recursos do **sequelize** em um ambiente web, vamos usar no frameworks **ExpressJS**. Para tanto, crie um arquivo, na raiz do projeto com o nome de **index.js** e adicione o seguinte código:

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function(req, res) {
5    res.send('Olá Mundo!');
6  });
7
8  app.listen(3000, function() {
9    console.log('App de Exemplo escutando na porta 3000!');
10 });
```

Vamos adicionar o ExpressJS ao nosso projeto executando o código abaixo no terminal.

```
1  npm i express --save
```

Outra biblioteca opcional é o **nodemon** que auxilia no desenvolvimento executando o *restart* da aplicação;

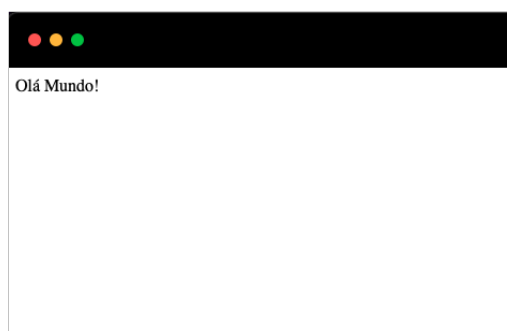
```
1  npm i nodemon --save-dev
```

Agora, edite o arquivo **package.json** e altere a seguinte linha:

```
1  // Terço código antigo
2  "scripts": {
3    "test": "echo \"Error: no test specified\" && exit 1"
4  },
```

```
1 // Trecho código novo
2 "scripts": {
3   "start": "nodemon index.js" // caso esteja usando o nodemon
4 },
5
6 "scripts": {
7   "start": "node index.js" // caso não esteja usando o nodemon
8 },
```

Por fim, executando o ‘npm start’ e abrindo o endereço ‘localhost:3000’ devemos visualizar a seguinte mensagem:



**Figura 14** – Fonte: O autor

Agora, se tudo deu certo, estamos prontos para criar nossas interações com o banco de dados, também conhecido como CRUD.

## 6.6 CRUD com Sequelize

Os conceitos sobre CRUD (Create, Read, Update e Delete), já foram abordados no Capítulo 5.2.3. Portanto, não iremos retornar o conceitos, mas, apenas aplicá-lo.

### 6.6.1 Inserindo dados - Create

Agora vamos iniciar nosso CRUD com o C de Create. O Sequelize fornece alguns métodos para a inserção de dados, sendo a mais direto deles o método **create**. Esse método cria imediatamente o registro na tabela com o **objeto** passado por parâmetro. Veja o código:

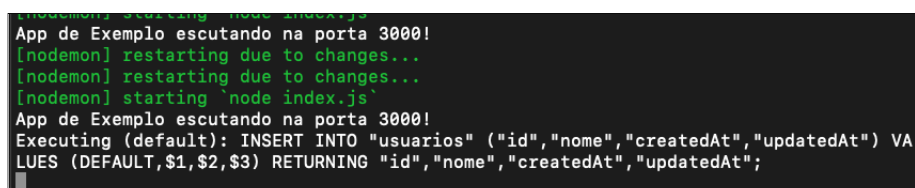
```
1  const usuario = await Usuario.create({
2    nome: 'Picoló'
3  })
```

Contudo, como estamos usando o ExpressJS, vamos adicionar o código criado em um rota com o verbo HTTP **Get** inicialmente.

```
1  var express = require('express');
2  var { usuario } = require('./models');
3
4  var app = express();
5
6  app.use(express.json());
7  app.use(express.urlencoded({ extended: true }));
8
9  app.post('/', async function(req, res) {
10    var resultado = await usuario.create()
11    res.json(resultado);
12  });
13
14  app.listen(3000, function() {
15    console.log('App de Exemplo escutando na porta 3000!');
16  });
```

Note as linhas 2. 6 e 7. Elas não existias no código anterior e foram adicionado, respectivamente, para importar o modelo **usuario** para nossas rotas e realizar parsing do conteúdo das requisições que ela receber.

Ao rodar a aplicação agora você vai notar que no console vão aparecer algumas informações relevantes desta vez, como, por exemplo, o SQL do INSERT (porque ela não existia) que foi gerado automaticamente para você.



```
[nodemon] starting 'node index.js'
App de Exemplo escutando na porta 3000!
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting 'node index.js'
App de Exemplo escutando na porta 3000!
Executing (default): INSERT INTO "usuarios" ("id","nome","createdAt","updatedAt") VA
LUES (DEFAULT,$1,$2,$3) RETURNING "id","nome","createdAt","updatedAt";
```

Figura 15 – Fonte: O autor

### 6.6.2 Retornando dados do banco - Read

Agora vamos fazer o R do CRUD, de Read/Retrieve. Ele é ainda mais simples do que o Create, basta usarmos a função **findAll** disponibilizada pelo nosso objeto **Usuario**, que teremos um array de produtos à nossa disposição.

```
1 app.get('/', async function(req, res) {
2   var resultado = await usuario.findAll();
3   res.json(resultado);
4 });
```

Caso seja necessário fazer uma busca no banco para retornar apenas um **usuário**, podemos utilizar o método **findByPk** ou **findOne**

```
1 app.get('/:id', async function(req, res) {
2   var resultado = await usuario.findByPk(req.params.id);
3   res.json(resultado);
4 });
5
6 app.get('/:id', async function(req, res) {
7   var resultado = await usuario.findOne({
8     where: {
9       id: req.params.id
10    }
11  });
12  res.json(resultado);
13 });
```

Aqui vale uma explicação sobre as três propriedades de requisição que são preenchidas de diferentes formas dependendo da origem:

**req.query** vem de parâmetros de consulta na URL, como link, em que **req.query.name** obtém o parâmetro passado pela seguinte url 'url/teste?nome=teste'

**req.params** vem de segmentos de caminho do URL que correspondem a um parâmetro na definição de rota, como `/song/:id`. Então, com uma rota usando essa designação e uma URL como `/song/48586`, então **req.params.id** retornará "48586".

Já a propriedade **req.body** vêm de uma postagem de formulário em que os dados do formulário (que são enviados no conteúdo do corpo) foram analisados nas propriedades da tag body.

## 6.7 Atualizando dados - Update

O próximo passo é atualizarmos um item da nossa tabela, o U do CRUD: Update! Para atualizarmos um item, primeiro precisamos retorná-lo do banco de dados usando alguma função de find do Sequelize. No passo anterior, usamos a `findByPk` para retornar o produto com ID 1. Vamos escrever o nosso código de update imediatamente abaixo.

```
1  app.put('/:id', async function(req, res) {
2    // Buscando dados
3    var resultado = await usuario.findByPk(req.params.id);
4
5    // Atualizada dados
6    resultado.nome = req.body.nome
7    var novo_resultado = await resultado.save();
8    res.json(novo_resultado);
9  });
```

## 6.8 Deletando dados - Delete

E por fim, vamos ao D do CRUD, de DELETE! Assim como para salvar e retornar dados existem diversas formas de fazer, para o Delete não é diferente. Você pode usar `Produto.destroy` e passar um `where` por parâmetro, ou então retornar um produto e usar a função `destroy` do próprio objeto retornado, você decide.

```
1  app.delete('/:id', async function(req, res) {
2    var resultado = await usuario.destroy({
3      where: { id: req.params.id }
4    });
5
6    res.json(resultado);
7  });
```

## 6.9 Padrão de rotas

Se vocês perceberam, as rotas seguem um padrão. Este pode ser usado em todos os projetos da forma descrita abaixo, sendo elas, as rotas, o básico de um projeto. Outras podem ser criadas especificadamente para atender determinada necessidade.

```
1  app.get('/', async function(req, res) {
2    // Listar todos os dados
3  });
4
5  app.get('/:id', async function(req, res) {
6    // Listar dado específico
7  });
8
9  app.put('/:id', async function(req, res) {
10   // Atualizar dado específico
11 });
12
13 app.post('/', async function(req, res) {
14   // Adicionar um novo dado
15 });
16
17 app.delete('/:id', async function(req, res) {
18   // Deletar dado específico
19 });
```

## 6.10 Relacionamentos com Sequelize

Ao trabalharmos com banco de dados relacionais, pela própria definição do nome, percebe-se que os relacionamentos se fazem presente. Dessa forma, o Sequelize, que é o ORM que estamos utilizando, tem mecanismos para abstrair os relacionamentos e “importá-los” para a programação orientada a objeto.

### 6.10.1 Tipos de relacionamento

Os tipos de relacionamento em um banco relacional, provavelmente, você já conhece. Sendo eles:

- 1 para 1



- 1 para N
- N para N

O Sequelize é compatível com as associações citadas, e os métodos de criação de relacionamentos, respectivamente, são:

- `hasOne` (tem um)
- `belongsTo` (pertence a)
- `hasMany` (tem muitos)
- `belongsToMany` (pertence a muitos)

## 6.11 Criando o primeiro relacionamento com a entidade **Usuario**

Utilizando a entidade **Usuario** criada anteriormente, vamos criar uma nova entidade chamada **Empresa** com apenas o atributo **nome**.

1

```
npx sequelize-cli model:create --name empresa --attributes nome:string
```

Agora, devemos relacionar **Empresa** com **Usuario**, no qual, as duas entidades devem possuir a seguinte relação:



**Figura 16** – Fonte: O Autor

Para tanto, devemos adicionar um novo atributo em **Usuário** que represente a **chave estrangeira** que referencia a tabela **empresas** que será gerada no banco de dados pelo Sequelize. Porém, nosso banco com a tabela **usuarios** já foi criada e, não devemos alterar as migração já persistida no banco. Para isso então, devemos criar uma **migração** que adicione os atributos desejados, da seguinte forma:

1

```
npx sequelize-cli migration:create --name  
↪ adicionar-empresa-id-para-usuario
```

E, diferente de quanto criamos o modelo, as migrações não adicionam o código para que seja adicionado o atributo. Logo, devemos realizar as mudanças de forma manual. Abra o arquivo criado e adicione o seguinte código:

```
1      'use strict';
2
3      module.exports = {
4          async up (queryInterface, Sequelize) {
5              await queryInterface.addColumn('usuarios', 'empresaId', {
6                  allowNull: false,
7                  type: Sequelize.INTEGER,
8                  references: { model: 'empresas', key: 'id' }
9              })
10         },
11
12         async down (queryInterface, Sequelize) {
13             await queryInterface.removeColumn('pacientes', 'empresaId')
14         }
15     };
```

Neste momento, você também deve criar todas as rotas necessárias para manipular os dados de uma **empresa**. Caso não lembre, recomendo que retorne a seção 6.6. Assim, usando o Sequelize, a nível de Banco de Dados, criamos o relacionamento. Agora, devemos criar o mesmo usando as referências do Sequelize.

## 6.12 Relacionamento 1 para N

Para relacionarmos **empresa** com **usuario**, devemos usar os métodos de relacionamento presentes no Sequelize listando anteriormente. No caso de um relacionamento **1 para N** dizemos que:

- Um usuário pertence a uma empresa
- Já uma empresa, possui muitos usuários

Adicionado ao código para cada classe do respectivo modelo, ficaria da seguinte forma:

```
1 // Entidade Empresa
2 static associate(models) {
3   this.hasMany(models.usuario, { as: 'usuarios' })
4 }
5
6 // Entidade Usuario
7 static associate(models) {
8   this.belongsTo(models.empresa, { as: 'empresa' })
9 }
```

### 6.12.1 Retornado dados por meio do relacionamento criado

Agora, vamos retornar os dados de cada entidade. Um **usuário** deve estar contido dentro de uma empresa. Assim, vamos criar uma rota para retornar a empresa que o usuário está relacionado.

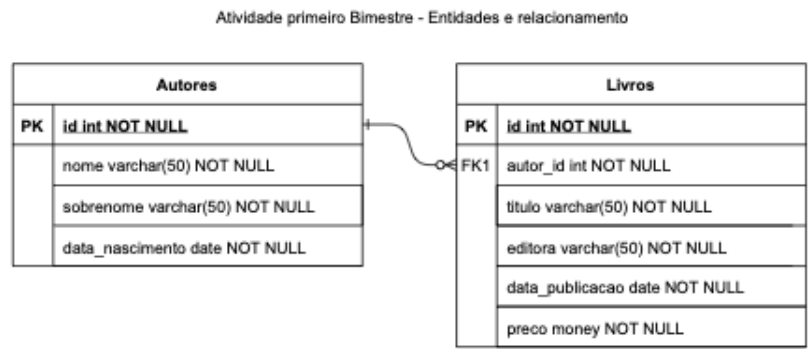
```
1 app.get('/usuarios/:id/empresa', async function(req, res) {
2   var resultado = await usuario.findByPk(req.params.id, {include:
  ↳ 'empresa'});
3   res.json(resultado.empresa);
4 });
```

E da mesma forma, vamos criar uma rota que retorne todos os usuários de uma empresa.

```
1 app.get('/empresas/:id/usuarios', async function(req, res) {
2   var resultado = await empresa.findByPk(req.params.id, {include:
  ↳ ['usuarios']});
3   res.json(resultado.usuarios);
4 });
```

## 6.13 Exercício de fixação

Crie dois modelos (usando o sequelize-cli) seguindo o Diagrama Entidade Relacional (DER) utilizado no trabalho do primeiro bimestre. Crie todas as rotas necessárias.



**Figura 17** – Relacionamentos Autores e Livros Fonte: O Autor

## 7 Código Limpo

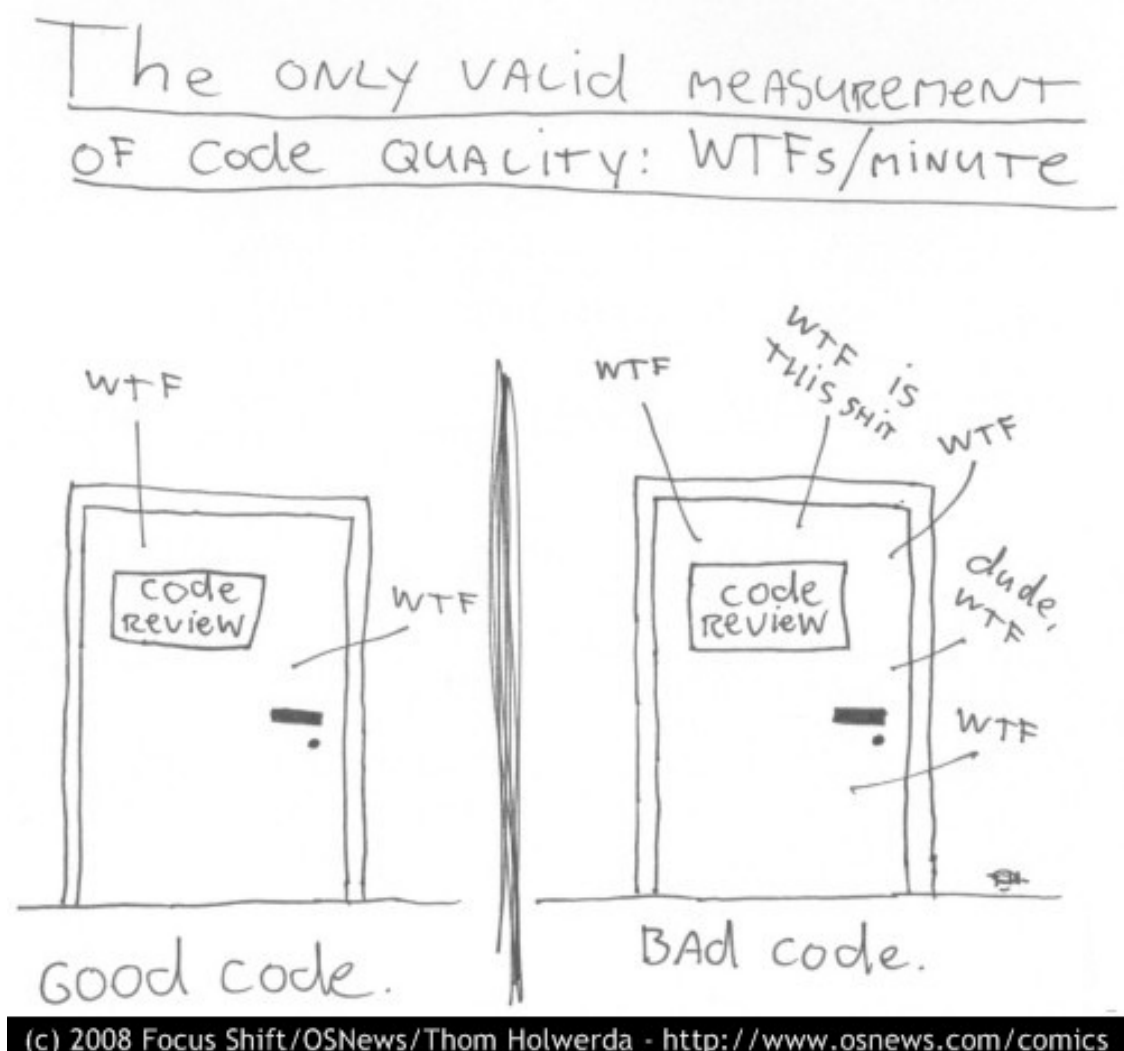
*Um ladrão rouba um tesouro, mas não furta a inteligência.*

*Uma crise destrói um herança, mas não uma profissão.*

*Não importa se você não tem dinheiro, você é uma pessoa rica,  
pois possui o maior de todos os capitais: a sua inteligência.*

*Invista nela. **Estude!***

**Augusto Cury**



A imagem acima ilustra o livro “Código Limpo” de Robert Cecil Martin também conhecido com Uncle Bob. O livro é um guia para se produzir código **legível**, **reutilizável** e **refatorável**. Assim, baseando-se nesses conceitos, a ideia é demonstrar o uso de alguns deles com a linguagem JavaScript.

Assim, citando um trecho do [Conceitos de Código Limpo adaptados em JavaScript](#) “aprender isto não irá lhe transformar imediatamente em um desenvolvedor(a) de software melhor, trabalhar com eles por muitos anos não quer dizer que você não cometerá erros. Toda porção de código começa com um rascunho, como argila molhada sendo moldada em sua forma final. Finalmente, talhamos as imperfeições quando revisamos com nossos colegas. Não se sinta culpado pelos primeiros rascunhos que ainda precisam de melhorias. Ao invés, desconte em seu código”.

## 7.1 O que é código limpo (Clean Code)

Uma das atividades que as pessoas que desenvolvem softwares realizam além da codificação é a manutenções em sistemas. Para isso, é preciso fazer a leitura e o entendimento de inúmeros códigos fontes de programas desenvolvidos por outras pessoas. Essa atividade pode ser muito mais fácil de se executar caso o desenvolvimento seja feito com base em boas práticas de programação.

**Clean Code**, que significa código limpo, é uma técnica que cria um conjunto de práticas e orientações sobre como desenvolver códigos que sejam facilmente entendidos, escritos e mantidos pelas pessoas desenvolvedoras. O objetivo é garantir o desenvolvimento de aplicações com códigos de qualidade que possam ser facilmente reutilizados.

## 7.2 Variáveis de Ambiente

Se existe uma coisa que vários projetos de desenvolvimento tem em comum são dados sensíveis, como dados de acesso a banco de dados, chaves de API's, *Secret Keys* entre outras dados de configuração necessárias. Como são dados críticos, eles não podem ter acesso público e, de forma alguma, podem estar em um repositório de dados aberto.

Outro ponto são os ambientes de desenvolvimento. Quando o projeto está sendo executado e acessado por usuários dizemos que eles está em um **Ambiente de Produção - Production**. Já, quando ele é executado pelos desenvolvedores dizemos que ele está em um **Ambiente de Desenvolvimento - Development**. Ambos possuem características semelhantes, mas executam de forma diferente. Assim, o banco de dados, por exemplo, que roda em **produção** tem acesso diferente ao banco que roda em **desenvolvimento**, logo, possui acesso (usuário e senha) diferente. Logo, nosso projeto, deve possuir essas características, e para isso vamos usar as **Variáveis de Ambiente**.

### 7.2.1 Dotenv

O **Dotenv** é a ferramenta utilizada para separar as variáveis ambientes de um projeto. O nome dela sugere o arquivo em que as informações ficarão, **dot** que é ponto em

inglês acrescido de **env**, então temos o arquivo **.env** que é composto de chaves e valores. Assim, para começar a usar, vamos instalar as dependências necessárias.

```
1 yarn add dotenv-safe
```

Após a instalação, vamos criar um arquivo na raiz do projeto chamada **.env.exemplo**. Esse arquivo conterá as variáveis que vamos usar em nosso projeto. Todas elas devem, por padrão, serem em **maiúsculo** seguido do sinal de igual.

```
1 # Variáveis para o banco de dados
2 USUARIO=teste
3 SENHA=teste
```

Já, para acessar-mos os dados das variáveis, em nosso arquivo principal, no topo, vamos adicionar o seguinte código que incorporará o Dotenv em nosso código.

```
1 require("dotenv-safe").config();
```

Com o arquivo **.env** configurado é necessário executar o Dotenv que irá chama-lo para ler as variáveis e adiciona-las ao processo que o executou, sabendo que para acessar as informações usem:

```
1 process.env.[NOME_DA_CHAVE]
2 process.env.SENHA
3 process.env.USUARIO
```

Algumas observações para a criação das variáveis:

- As chaves são em caixa alta, por padrão e não exigência;
- As chaves não podem ter espaço;
- Os valores podem ser quaisquer tipo, que será retornado sempre uma string;
- Pode haver espaçamentos, porém é feito um trim na string;
- Pode existir chave sem valor, que retorna uma string vazia;

### 7.2.2 Boas práticas e cuidados

Por se tratar de informações sensíveis é importante que esses dados fiquem em seu ambiente de desenvolvimento, então se pretende compartilhar seu código, lembre-se de remover esse arquivo. Caso utilize o github basta adicionar ao `.gitignore` o arquivo `.env` para ele fazer esse trabalho para você.

Uma boa prática também é criar um arquivo de exemplo com as chaves que seu projeto está utilizando, sem os valores sensíveis, assim quem clonar o repositório ou ter acesso ao seu fonte não ficará perdido. Crie um **`.env.example`** e deixe apenas as variáveis de ambiente sem seus respectivos valores.

## 7.3 Aplicação de código limpo em variáveis, funções e classes



## 8 Autenticação e autorização com JWT

## Referências

- ELMASRI, R. et al. *Sistemas de banco de dados*. Pearson Addison Wesley São Paulo, 2005. Citado na página 24.
- MAGAZINE, N. *ORM - Object Relational Mapping - Revista Easy .Net Magazine* 28. 2020. Disponível em: <<https://www.devmedia.com.br/orm-object-relational-mapping-revista-easy-net-magazine-28/27158>>. Acessado em: 18/04/2022. Citado 2 vezes nas páginas 38 e 39.
- MURAMATSU, A. *Introdução a ORM no Node.js com Sequelize + exemplo prático*. 2020. Disponível em: <<https://ezdevs.com.br/introducao-a-orm-no-node-js-com-sequelize/>>. Acessado em: 11/04/2022. Citado na página 38.
- PARA iniciantes. Disponível em: <<https://tableless.github.io/iniciantes/manual/js/o-que-framework.html>>. Citado na página 16.
- SANCHES, A. R. *Disciplina: Fundamentos de Armazenamento e Manipulação de Dados*. 2005. Acessado em: Disponível em: <<https://www.ime.usp.br/~andrers/aulas/bd2005-1/aula7.html>>. Citado 3 vezes nas páginas 26, 27 e 28.
- SONG, H.; GAO, L. Use orm middleware realize heterogeneous database connectivity. In: IEEE. *2012 Spring Congress on Engineering and Technology*. [S.l.], 2012. p. 1–4. Citado na página 38.
- TAKAI, O. K.; ITALIANO, I. C.; FERREIRA, J. E. *Introdução a banco de dados. Departamento de Ciências da Computação. Instituto de Matemática e Estatística. Universidade de São Paulo. São Paulo*, 2005. Citado 3 vezes nas páginas 24, 25 e 26.