

Projeto Controle de Estudantes

Docupedia Export

Author:Ferro Alisson (CtP/ETS)

Date:12-Jun-2023 17:56

Table of Contents

1 Projeto Alvo:	3
2 Objetivo	4
3 Começando no FrontEnd	5
4 Começando o BackEnd	12
5 Partindo para o Microsoft SQL Server (MSSQL)	14
6 Conectando no SQL Server (MSSQL)	21
7 Inserindo Informações no Banco de Dados	25
8 Entendendo o Funcionamento do EJS	27
9 Exibir Todos os Dados na Página Principal	30
10 POST da Página Inicial	32
11 Enviando e Recebendo as imagens	35
12 Multer	36
13 Editar Informações dos Alunos	39
14 Desafios	46
15 Desafio dos Desafios	47
16 Erros/Problemas	48

1 Projeto Alvo:

- Criação de um site totalmente funcional para controle de Salas e seus respectivos Estudantes.
- Utilizando dos conhecimentos adquiridos ao longo do curso, iremos utilizar HTML, CSS, JavaScript e Banco de Dados (SQL Server).

2 Objetivo

- O site mostrará o Nome e a Idade de cada Aluno cadastrado na Sala de aula selecionada pelo usuário.
 - Sendo possível a modificação dos dados de cada aluno através do botão de Editar.
 - Cadastrar novos Estudantes e novas Salas.
-

3 Começando no FrontEnd

- Utilizando do HTML, CSS e Bootstrap, crie as páginas a seguir.
- Obs: Permitido modificar o design de acordo com sua escolha.

Página Principal


- Conter um Header com um título na **Esquerda** e duas opções na **Direita**, uma para cadastrar a **Sala** e a outra para os **Alunos**;
- Deve possuir um **Select/Option** para seleção das salas disponíveis;
- Possuir **Cards** (Disponível no Bootstrap), com uma foto e descrição;

CONTROLE de Estudantes


Cadastrar: Alunos | Sala

Salas de Aula: TI ▼


Alunos




Donathan
Idade: 19
[Editar](#)




Queila
Idade: 19
[Editar](#)



Fabio
Idade: 19
[Editar](#)



Andressa
Idade: 19
[Editar](#)

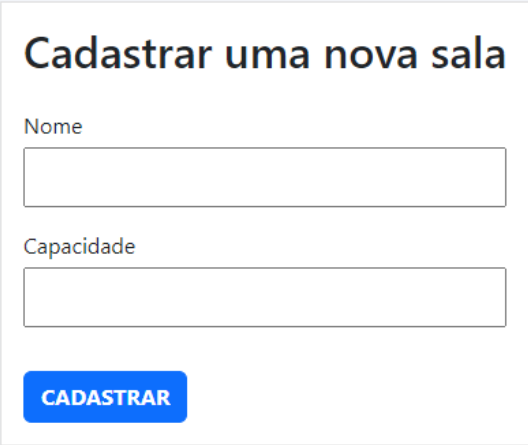


João
Idade: 19
[Editar](#)

Cadastro de Salas

Contendo o mesmo Header que na página principal, crie um formulário que possua os campos de **Nome** e **Capacidade** para preenchimento, e um botão de **submit** ao final.

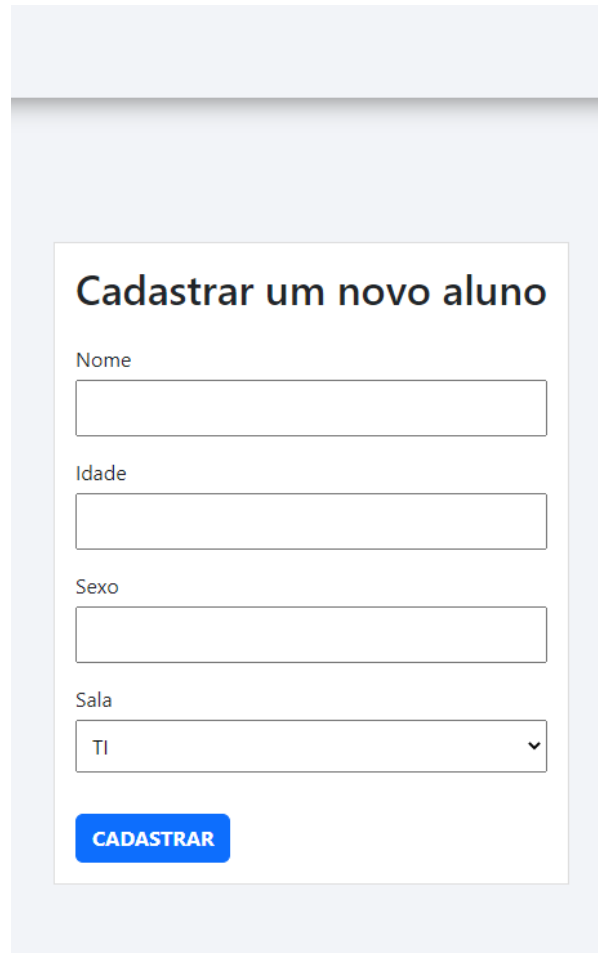
Importante: Todos os inputs do formulário deve possuir o parâmetro **name**, é o nome que será utilizado para puxar esses dados pelo Node.



The image shows a web form titled "Cadastrar uma nova sala" (Register a new room). The form is centered on a light gray background. It contains two text input fields: "Nome" (Name) and "Capacidade" (Capacity). Below these fields is a blue button with the text "CADASTRAR" in white capital letters.

Cadastro de Alunos

Seguindo a mesma lógica que o cadastro de salas, crie um formulário com os campos de **Nome**, **Idade** e **Sexo** do tipo **text**, possuindo um campo do tipo **Select** para selecionar uma sala existente.

A screenshot of a web form titled "Cadastrar um novo aluno" (Register a new student). The form is centered on a light gray background. It contains four input fields: "Nome" (Name), "Idade" (Age), "Sexo" (Gender), and "Sala" (Room). The "Sala" field is a dropdown menu with "TI" selected. Below the fields is a blue button labeled "CADASTRAR".

Cadastrar um novo aluno

Nome

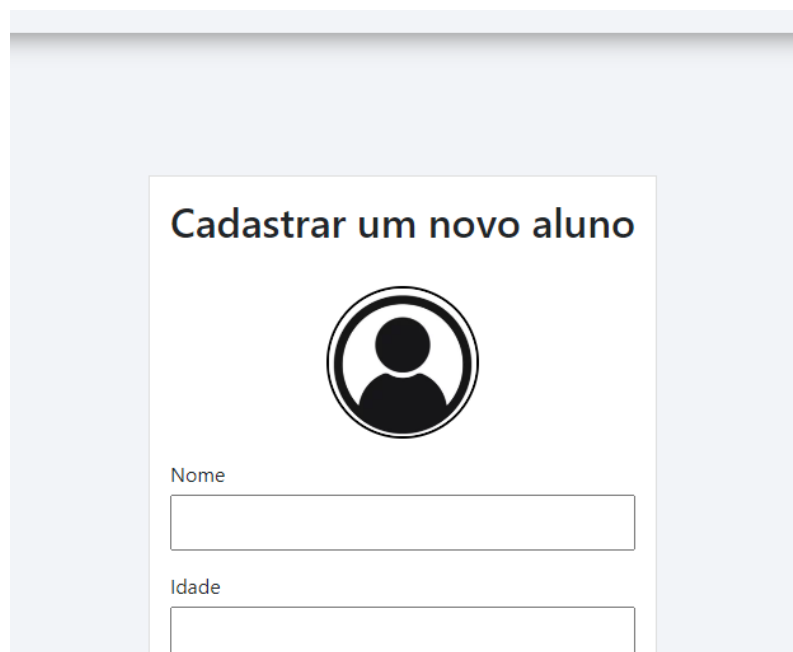
Idade

Sexo


Sala

CADASTRAR

Criação da foto de avatar



Cadastrar um novo aluno



Nome

Idade

Crie uma **tag img** com uma foto padrão de usuário como a imagem abaixo;

Crie um **input** do tipo **file** abaixo da imagem, passando o parâmetro **accept="image/*"** dentro do input para aceitar apenas imagens;

```
<input type="file" id="flImage" name="foto" accept="image/*">
```

No CSS deve informar que o input do tipo File deve possuir um **display: none;**

Utilizando JavaScript

Será criado um JavaScript, que quando for clicado na imagem, executará a função **click()** do input file, ou seja, estará chamando o input através da imagem que se comportará como um botão;

Para isso o JavaScript precisará receber os elemento **img** e **input** através do seu ID;

Criando uma função que executará quando houver um **click** na imagem;

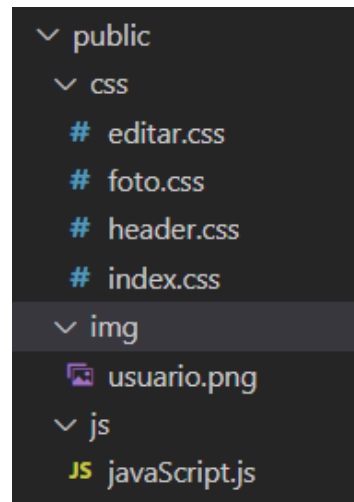
```
let photo = document.getElementById('imgFoto');  
let file = document.getElementById('flImage');
```

```
photo.addEventListener('click', () => {  
  file.click();  
});
```

Agora utilize uma função para quando o input file receber algo, ele irá mudar automaticamente a foto do elemento img para que o usuário possa visualizar;

```
file.addEventListener('change', () => {  
  
  // Sem essa verificação, ele irá dar erro quando o usuário clicar em cancelar  
  // pois enviará uma "imagem" vazia  
  if (file.files.length == 0) {  
    return;  
  }  
  
  // Inicializando a função que pega o caminho da imagem  
  let reader = new FileReader();  
  
  // Está pegando o caminho da imagem  
  reader.readAsDataURL(file.files[0]);  
  
  // Coloca o caminho da imagem no Source da tag IMG  
  reader.onload = () => {  
    photo.src = reader.result  
  }  
  
});
```

Crie as pastas **css**, **img**, **js** dentro de uma pasta chamada **public**, organizando os arquivos criados anteriormente em suas respectivas pastas.



O arquivos **HTML** serão os arquivos **EJS** que vimos anteriormente, crie a pasta **src** no mesmo nível da **public**, e coloque os arquivos dentro de uma pasta chamada **views**.

4 Começando o BackEnd

Instalar as bibliotecas.

- npm init -y
- npm install express mssql sequelize nodemon ejs

1. Fora do src, na pasta raiz do projeto, crie os arquivos **server.js** e **routes.js** para configurar a parte principal do servidor.

server.js

```
const express = require('express');
const routes = require('./routes');

const app = express();

app.use(express.urlencoded({ extended: true }));

// Static files
app.use(express.static('public'));

// EJS
app.set('views', './src/views');
app.set('view engine', 'ejs');

app.use(routes);

app.listen(3000, () => console.log('Acesse: http://localhost:3000/'));
```

2. Dentro do **src**, crie a pasta **controllers** e dentro dele, crie o arquivo **home.js**, onde será passado o caminho do **ejs** para que sejam renderizados.

```
module.exports = {
  async pagInicialGet(req, res){
    res.render('./views/index');
  }
}
```

3. Configure o **Routes.js** para que seja possível o acesso da página pelo localhost, importando o controllers.

routes.js

```
// Iniciando Route do Express
const express = require('express');
const route = express.Router();

// Importando os Controllers
const home = require('./src/controllers/home');

// Iniciando as rotas
route.get('/', home.pagInicialGet);

module.exports = route;
```

4. Ajuste o link do css, js, e img para o caminho correto a partir do public.

```
// Para a página principal
// localhost:3000/css/index.css
<link rel="stylesheet" href="css/index.css">

// Para as outras páginas
// Colocamos o ../ para voltar uma pasta, pois o css não está em localhost:3000/editar/css/editar.css
<link rel="stylesheet" href="../css/editar.css">
```

Nesse ponto já é possível acessar a página principal do projeto.

5 Partindo para o Microsoft SQL Server (MSSQL)

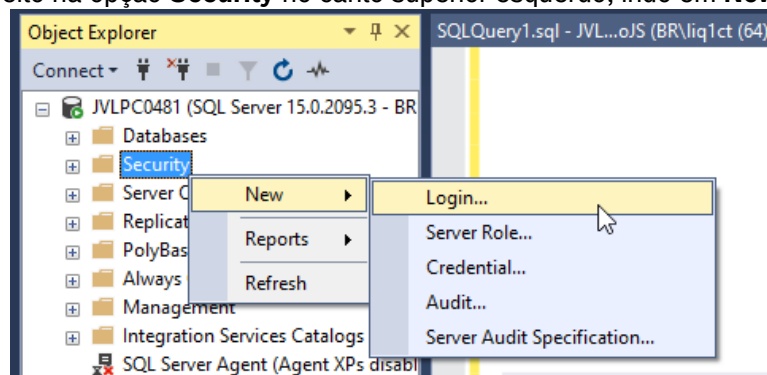
Para acessar o SQL Server remotamente, é preciso mudar algumas configurações que vem como padrão quando instalado.

O Sequelize solicita o **database**, **login** e a **senha** para se conectar com o SQL, então o database deve ser criado manualmente, e depois vinculado um usuário que será criado posteriormente.

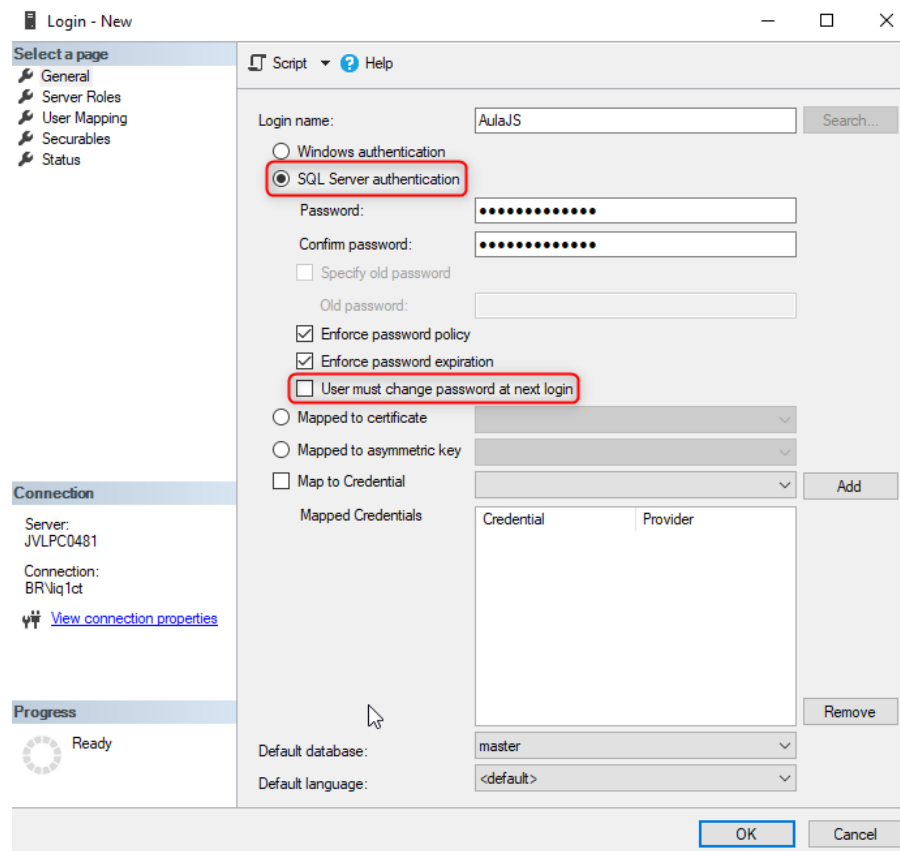
1. Crie um database com o nome desejado.

2. Criando um usuário.

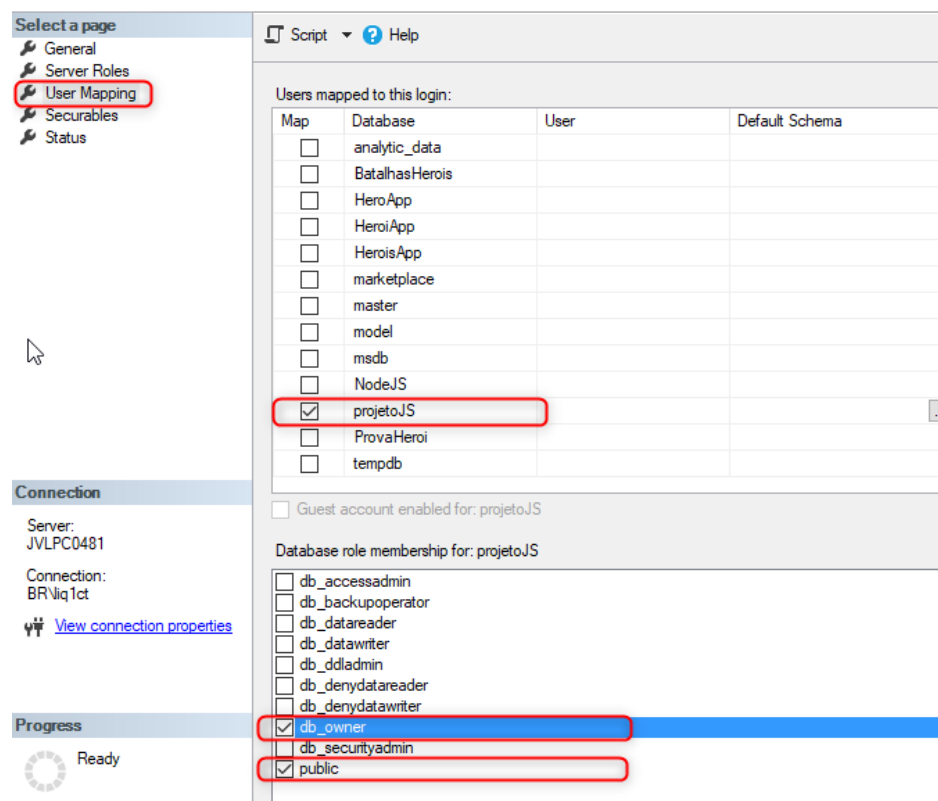
Para criação de um usuário, clique com o botão direito na opção **Security** no canto superior esquerdo, indo em **New**, e depois em **Login...**



1. Quando aparecer a janela seguinte, clique em **"SQL Server authentication"**, para habilitar a criação do usuário.
2. Crie o **"Login name"** e o **"Password"** ao lado.
3. Desabilite a terceira opção já marcada, onde está escrito **"User must change password at next login"**, para que não precise mudar a senha depois.



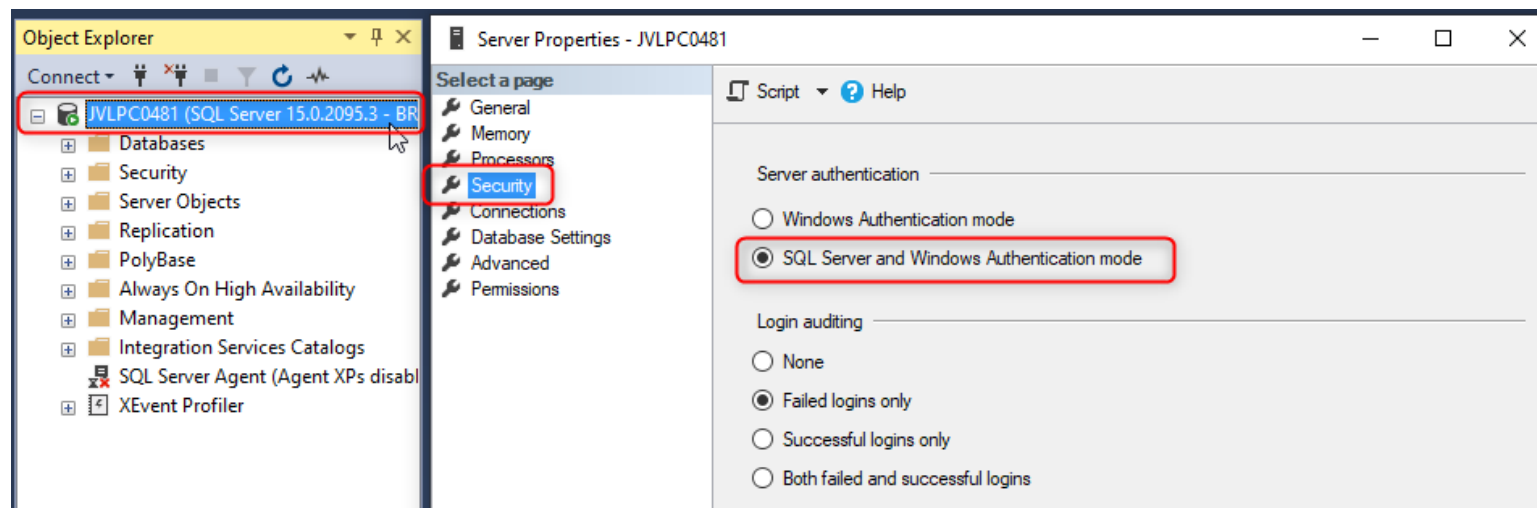
1. Após isso, clique a opção **"User Mapping"** no canto superior esquerdo.
2. Selecione o database que você acabou de criar.
3. Deixe selecionado abaixo as opções **"public"** e **"db_owner"**, para que esse usuário possa fazer o CRUD nesse database.
4. Após isso, pode clicar em OK.



Depois de todos esse passos, ainda é preciso modificar a configuração que permite a conexão remota com o SQL.

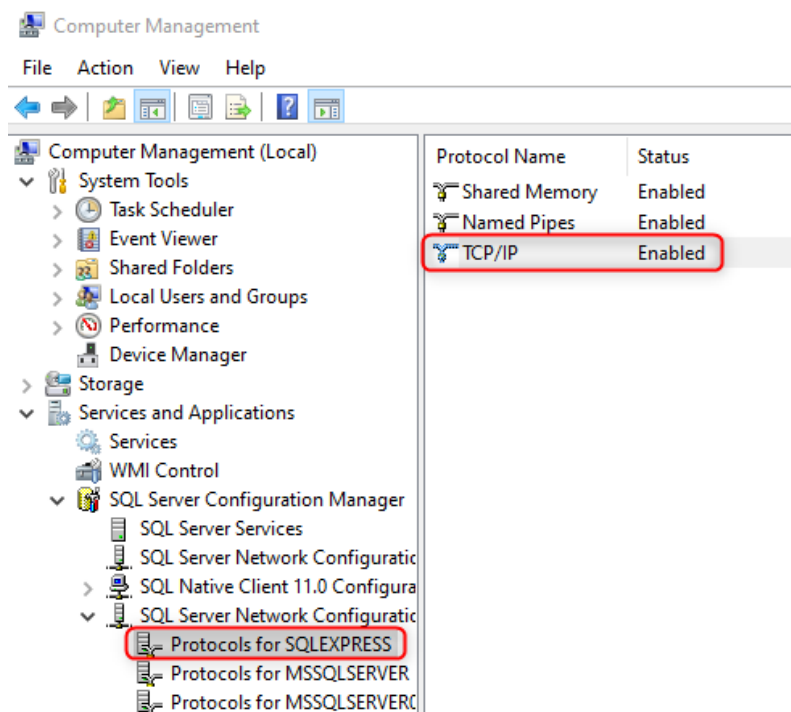
3. Configurando acesso pelo usuário.

1. Lá no início do **"Object Explorer"** no superior esquerdo, clique com o botão direito no nome do servidor e selecione "Properties" (Propriedades).
2. Na nova janela, selecione **Security** na esquerda, e habilite a opção **"SQL Server and Windows Authentication mode"**.



3. Habilitando TCP/IP.

Acesse o "Computer Management" ou o "SQL Server Configuration Manager" como **administrador** para habilitar a opção TCP/IP e podermos conectar com o SQL.



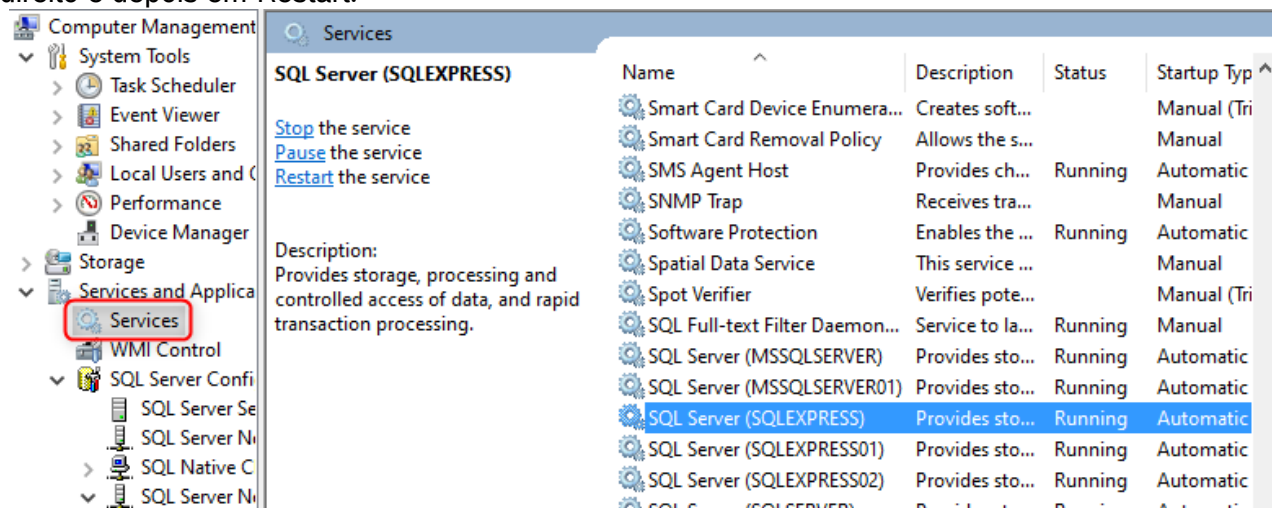
Em "Protocols for SQLEXPRESS", habilite a opção TCP/IP como na imagem acima.

4. Encontrar a porta de conexão.

Para encontrar a porta que será utilizada para conectar, é preciso:

Clicar duas vezes na opção que foi habilitada, clique em "IP Addresses" e vá até as últimas opções. **Guarde** a porta que estiver lá, pois será utilizada na programação para a conexão com o banco.

3. Clique com o botão direito e depois em Restart.



6 Conectando no SQL Server (MSSQL)

1. Conectar com o SQL

Agora que está tudo configurado no SQL, será o momento de conectar com a aplicação no JavaScript.

Dentro da pasta **src**, crie uma pasta chamada **config**, nela, conecte a aplicação com o SQL através de um arquivo chamado **db.js**. Não esqueça de importar essa pasta no **server.js**.

db.js

```
const sequelize = require('sequelize');

//configurações da base de dados
const database = new sequelize('projetoJS', 'AulaJS', 'JSqL0123_QWE',
{
  dialect: 'mssql', host: 'localhost', port: 1433
});

database.sync();

module.exports = database;
```

2. Criação das tabelas

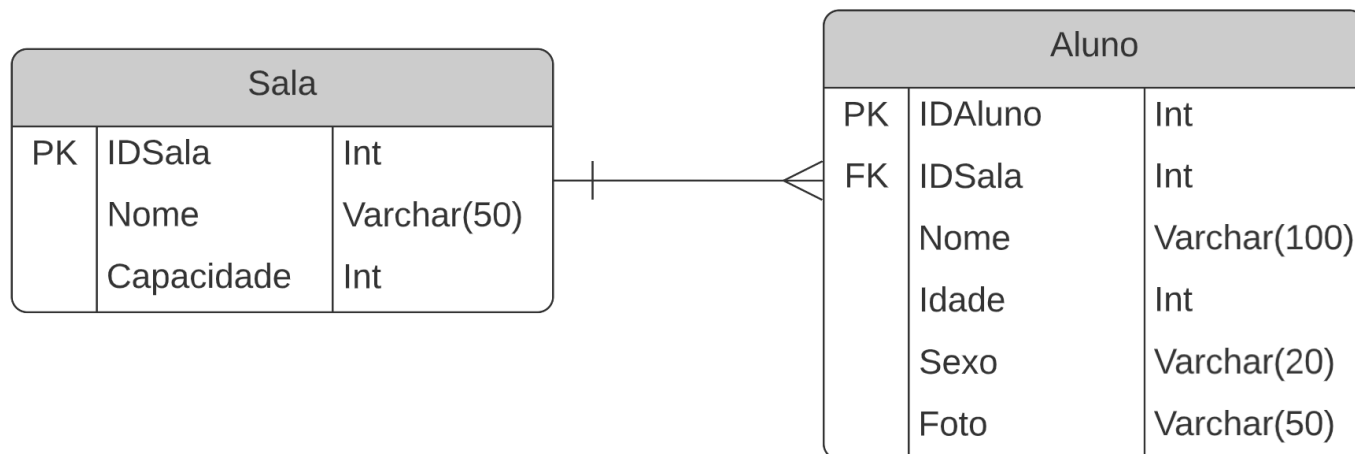
Vamos agora criar as tabelas que será utilizado no banco de dados através do Sequelize.

Uma coisa boa de utilizar o Sequelize para criação de tabelas, é que ele irá verificar se essas tabelas já existem no SQL, se não existirem ele irá criar, se já existirem ele apenas irá utilizar as já existentes.

Depois não há porque de se preocupar com a criação dessas tabelas.

Crie então a pasta **model**, que será onde será guardado os códigos para criação das tabelas.

Seguindo o diagrama a seguir, crie dois arquivos, o **aluno.js** e **sala.js**.



1. Começando pela tabela "sala.js".

```
// Importação
const Sequelize = require('sequelize');
const database = require('../config/db');

// Criando a tabela Sala
const sala = database.define('Sala', {
  IDSala: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    allowNull: false,
    primaryKey: true
  },
  Nome: {
    type: Sequelize.STRING(50),
    allowNull: false
  },
  Capacidade: {
    type: Sequelize.INTEGER,
```

```
        allowNull: false
      }
    });

    // Exportando essa tabela
    module.exports = sala;
```

2. Agora faça por conta própria o "aluno.js".

Só fica a questão de que não é preciso criar o **IDSala**, pois quando for realizada a integração das tabelas, ele irá criar automaticamente essa coluna.

aluno.js

```
const Sequelize = require('sequelize');
const database = require('../config/db');

const aluno = database.define('Aluno', {
  IDAluno: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    allowNull: false,
    primaryKey: true
  },

  Nome: {
    type: Sequelize.STRING(100),
    allowNull: false
  },

  Idade: {
    type: Sequelize.INTEGER,
    allowNull: false
  },

  Sexo: {
    type: Sequelize.STRING(20),
    allowNull: false
  },
```

```
    Foto: {
      type: Sequelize.STRING(50),
      allowNull: false
    }
  });

module.exports = aluno;
```

2. Integrar tabelas.

Para integrar as tabelas, primeiro volte ao código do arquivo **aluno.js** e importe a **sala.js**.

Ainda no arquivo **aluno.js**, digite o seguinte código antes do **module.exports**.

```
aluno.belongsTo(sala, {
  constraint: true, //Garantir integridade referencial
  foreignKey: 'IDSala'
});
```


7 Inserindo Informações no Banco de Dados

1. Inserindo as Salas e os Alunos.

Crie o arquivo **cadastro.js** dentro da pasta **controllers**, fazendo o mesmo que a **home.js** para acessar as páginas de cadastro de **Salas e Alunos**. Importe o **cadastro.js** para o **routes.js** para que elas possam ser acessadas.

Quando for realizada a inserção das informações pelo **form** do **front-end**, o JS irá receber as informações pela **requisição**, e posteriormente serão inseridas no banco.

```
// Importando as tabelas do DB
const sala = require('../model/sala');
const aluno = require('../model/aluno');

module.exports = {
  async sala(req, res){
    res.render('../views/cadastroSala');
  },

  async salaInsert(req, res){

    // Recebe as informações do front-end
    const dados = req.body;

    // Criando sala no banco de dados
    await sala.create({
      Nome: dados.nome,
      Capacidade: dados.capacidade
    });

    // Redirecionar para a página principal
    res.redirect('/');
  }
}
```

Agora faça o mesmo para os **alunos**, mas não se esqueça de que os alunos devem ser associados a uma sala, então acesse e mostre todas as salas criadas no banco.

Então no mesmo arquivo, crie o **"async aluno"**.

```
async aluno(req, res){  
  
  // Encontrando todas as salas disponíveis no SQL  
  const salas = await sala.findAll({  
    raw: true, // Retorna somente os valores de uma tabela, sem os metadados.  
    attributes: ['IDSala', 'Nome']  
  });  
  
  // Renderizando e passando o nome das salas para o front  
  res.render('../views/cadastroAluno', {salas});  
  
}
```

8 Entendendo o Funcionamento do EJS

Como dito anteriormente nas aulas, o EJS é uma engine de visualização, que com ele conseguimos de uma maneira fácil e simples transportar dados do BackEnd para o FrontEnd, onde é possível utilizar códigos JavaScript no HTML das páginas.

Para referenciar no código que determinada parte do arquivo não é mais HTML e sim JavaScript, é necessário deixar com a seguinte sintaxe:

Código HTML

```
<% for () { ou if () { %>
```

Código HTML

```
<% } %>
```

Código HTML

Quando queremos que uma informação seja inserida no HTML, utilizamos o "=" depois do primeiro sinal de porcentagem.

```
<option value='<%= Dado do JS %>'> <%= Dado do JS %> </option>
```

Quando o **JS** for inserido no **EJS**, ficará ruim a visualização, pois o código ainda não sabe que está sendo utilizado duas linguagens no mesmo arquivo.

É possível arrumar isso instalando uma extensão do Visual Studio Code referente ao EJS.

A extensão utilizada no exemplo foi o **EJS language support**.

Utilizando o EJS para criação do Aluno

Crie um **option** dentro de um **for**, assim é possível listar todas as salas do banco de dados.

Coloque o **ID da sala** dentro do **value**, e poder saber qual a sala que o usuário selecionou quando esses dados retornarem.

Não esqueça de colocar o "select" como **required**, para que não possa ser criado um aluno sem uma sala.

ejs

```
<% for (let i=0; i<salas.length; i++) { %>
  <option value='<%= salas[i].Nome %>' > <%= salas[i].Nome %> </option>
<% } %>
```

Inserir os alunos no SQL

Da mesma maneira que foi enviado os dados para criação da sala no banco de dados, repita o processo para os alunos.

- Receba os dados enviados pelo **body**;
- Envie todos os dados para o banco de dados;
- No campo foto coloque '**usuario.png**', mais pra frente quando o usuário enviar uma imagem, iremos trocar o nome;

alunoInsert

```
async alunoInsert(req, res){

  // Recebendo as informações pelo Body
  const dados = req.body;

  // Nome padrão da foto
  let foto = 'usuario.png';

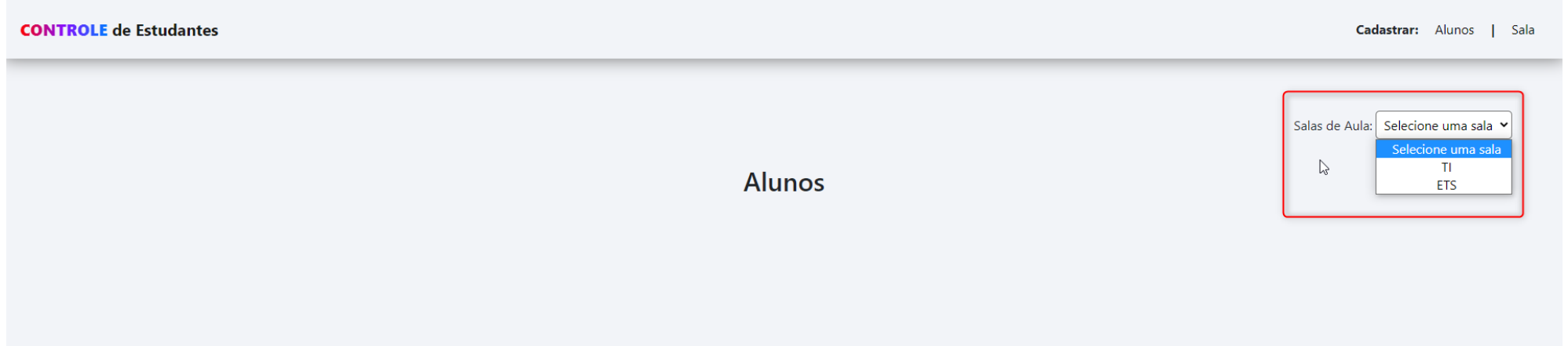
  // Criando aluno no banco de dados
  await aluno.create({
    Nome: dados.nome,
    Idade: dados.idade,
    Sexo: dados.sexo,
    IDSala: dados.sala,
    Foto: foto
  });

  // Redirecionar para a página principal
  res.redirect('/');
```

```
}
```

9 Exibir Todos os Dados na Página Principal

Na primeira página que o usuário acessar, apenas terá que mostrar as salas disponíveis para que ele possa escolher. Sem a necessidade de mostrar nenhum Card.



1. Então encontre no SQL, o **ID** e o **Nome** de cada sala, sendo o **Nome** para exibir ao usuário no **select** do HTML, e o **ID** para o retorno dentro do **value**.
home

```
async pagInicialGet(req, res){  
  
  const salas = await sala.findAll({  
    raw: true,  
    attributes: ['IDSala', 'Nome']  
  });  
  
  res.render('../views/index', {salas});  
  
}
```

2. Faça um **for** que mostre todas as salas no **select** para que o usuário possa escolher qual sala ele deseja consultar.
3. Adicione um **option** com **value** vazio (") em acima do for, essa será a opção que não irá exibir nenhum aluno, para que seja a opção de quando o usuário entrar no site.
4. Para selecionar a sala sem a necessidade de clicar em um botão **submit**, digite o seguinte código no **select**.

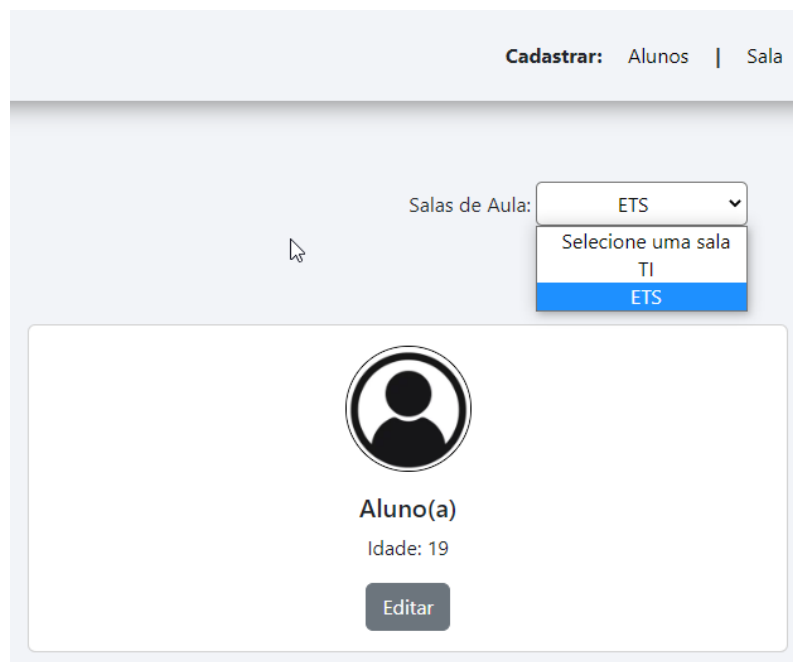
```
<select id="sala" onchange="this.form.submit()" name="nome">
```

Pois quando houver uma mudança de valores ele mesmo irá ativar o submit do formulário.

select

```
<select id="sala" onchange="this.form.submit()" name="nome">
  <option>Selecione uma sala</option>
  <% for (let i=0; i<salas.length; i++) { %>
    <option value='<%= salas[i].IDSala %>'> <%= salas[i].Nome %> </option>
  <% } %>
</select>
```

10 POST da Página Inicial



1. Pegue o **ID da Sala** que veio no **body** da requisição, envie junto nos parâmetros do render, pois será utilizado para defini-lo a sala como padrão no **"option"**.
2. Encontre no Banco, todos os dados necessários dos usuários para fazer os Cards (**ID do Aluno, Nome, Idade, Foto**), filtrando todos que pertencem ao **ID da Sala** recebida no **body**.
3. Encontre novamente o **ID** e o **Nome das salas**, pois mesmo que uma sala esteja selecionada, é possível mudar para outra.
4. Adicione os três parâmetros acima com o render da página.

homePost

```
async pagInicialPost(req, res){

  const id = req.body.nome;

  const alunos = await aluno.findAll({
    raw: true,
    attributes: ['IDAluno', 'Nome', 'Idade', 'Foto'],
    where: { IDSala: id }
  })
```



```
});  
  
const salas = await sala.findAll({ raw: true, attributes: ['IDSala', 'Nome'] });  
  
res.render('../views/index', {salas, alunos, id});  
  
}
```

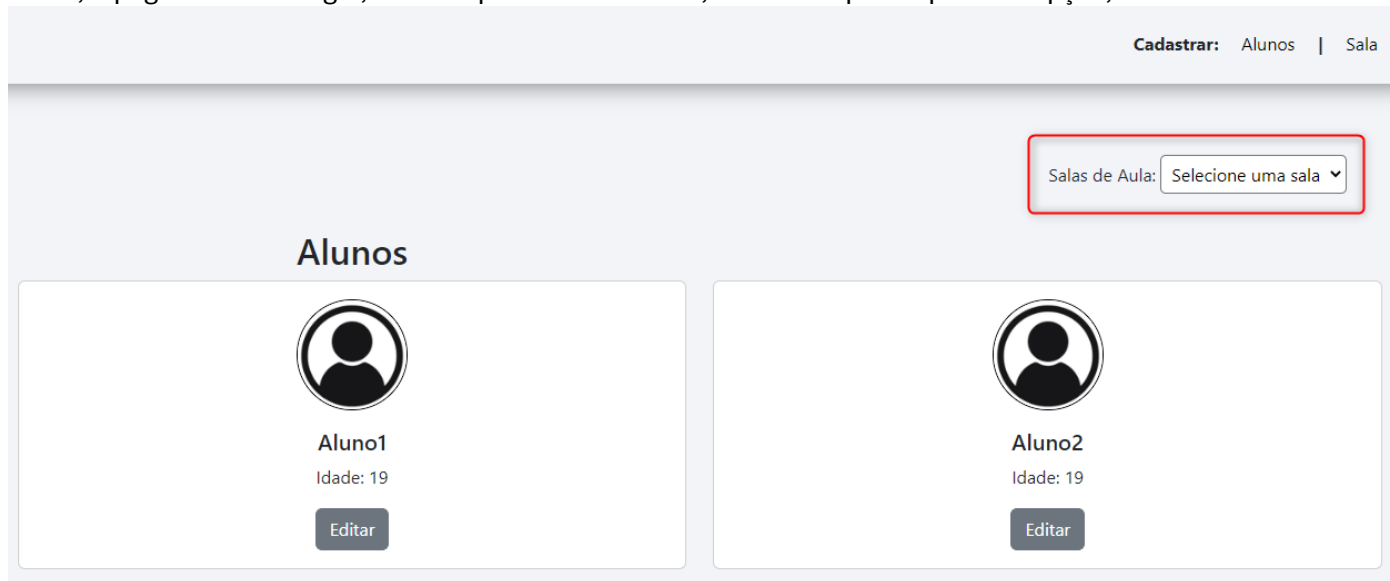
Os mesmos parâmetros que foi passado no **Post da página principal** será passado quando o usuário entrar pela primeira vez (**Get**), pois como está sendo utilizado a mesma página html, ele irá solicitar essas informações.

Então vá no **Get da página principal** e adicionamos esses valores como **Nulos**.

```
res.render('../views/index', {salas, alunos: '', id: ''});
```

Exibir a Sala no Select


Quando foi selecionado a sala, a página irá recarregar, mas na questão de estética, ele voltará para a primeira opção, e não fica visual a sala que está sendo exibida.




Cadastrar: Alunos | Sala

Salas de Aula: Selecione uma sala ▼

Alunos



Aluno1
Idade: 19
Editar



Aluno2
Idade: 19
Editar

Então utilize o EJS para ativar o atributo **selected** com o seguinte código.

Criando um **IF** que se for verdadeiro, terá o **selected** entre **aspas** para selecionar essa opção.

```
<option value='<%= salas[i].IDSala %>' <%= id == salas[i].IDSala ? 'selected' : '' %>> <%= salas[i].Nome %> </option>
```

Agora ele automaticamente irá deixar selecionado a sala em que os alunos pertencem.

E por último, crie um Card contendo as informações de cada aluno. Lembrando que já foram filtrados os alunos de acordo com a sala.

Não esqueça de que o botão **Editar** é um **submit** de um form, redirecionando para a página de edição.

Desta vez, utilize o método **Get** para dar uma treinada.

Então no **action** desse form, coloque o **endereço** normal da outra página, uma **barra**, e a informação que se deseja passar, que nesse caso é o **ID do aluno**.

card

```
<% for (let i=0; i<alunos.length; i++) { %>

  <div class="col">
    <div class="card h-100">
      
      <div class="card-body">
        <h5 class="card-title"><%= alunos[i].Nome %></h5>
        <p class="card-text">Idade: <%= alunos[i].Idade %></p>
        <form action="/editarAluno/<%= alunos[i].IDAluno %>" method="get">
          <button type="submit" class="btn btn-secondary">Editar</button>
        </form>
      </div>
    </div>
  </div>

<% } %>
```

11 Enviando e Recebendo as imagens

Você já deve ter percebido que estamos conseguindo enviar todas as informações dos alunos em forma de texto para realizar o cadastro, mas não estamos recebendo nenhuma foto, mesmo que selecionarmos uma.

Para fazer isso, primeiramente temos que habilitar a opção no HTML de enviar arquivos.

Então passe o parâmetro **enctype="multipart/form-data"** no **form** para dizermos que estaremos enviando um arquivo junto ao formulário.

```
<form action="/cadastroAluno" method="post" enctype="multipart/form-data">
```

12 Multer

Agora, estamos enviando o arquivo, mas o **body** está vindo **vazio**, sem qualquer dado. Para resolver isso, temos que **utilizar o Multer**.

O Multer é um **middleware**, ou seja, ele irá interceptar os dados enviados, fazendo a ponte entre o Front e o Back, com ele é possível receber a imagem, nomeá-la e salva-la em um arquivo local do servidor, que no nosso caso é o nosso próprio computador.

Nós iremos armazenar as fotos na pasta **public/img**.

Pois para salvar uma imagem no SQL Server, teríamos que transformá-la para base 64, e dependendo do tamanho da imagem, podemos ter mais de 1 milhão de caracteres em cada imagem.

E com o tempo, iríamos encher muito rápido nosso Banco, o que facilmente deixaria o SQL muito lento.

Então temos que instalá-lo para começarmos a usar.

npm install multer

Criando as Configurações do Multer

Na pasta config, crie um arquivo chamado **multer.js**, é nessa pasta que iremos alterar o nome e colocar o local onde iremos salvar a imagem.

```
// Importando Multer
const multer = require('multer');

// Configuração de armazenamento
const multerConfig = multer.diskStorage({

  // Criar destino de armazenamento
  destination: (req, file, cb) => {
    cb(null, 'public/img'); // (Caso de erro, Local de destino)
  },

  // Renomear arquivo
  filename: (req, file, cb) => {

    // Criando um novo nome para o arquivo (Data em milisegundos - nome original)
    const fileName = `${new Date().getTime()}-${file.originalname}`;

    // Alterando efetivamente o nome
    cb(null, fileName); // cb = Callback
  }
});
```

```
    }  
  });  
  
  // Exportando configurações  
  module.exports = { storage:multerConfig };
```

Inserindo Multer nas Rotas

Para que o Multer seja chamado para executar, devemos voltar ao **routes.js**.

Lá, devemos fazer a importação do Multer como **biblioteca**, e a importação do arquivo **multer.js** que acabamos de criar.

Adicionando essas informações como parâmetro em cada rota que iremos efetivamente utiliza-lo.

```
// Iniciando Multer  
const multer = require("multer");  
  
// Recebendo arquivo do multer que criamos  
const config = require('./src/config/multer');  
  
// Cadastro de aluno irá receber um arquivo com o "name" do HTML chamado de "foto"  
route.post('/cadastroAluno', multer(config).single('foto'), cadastro.alunoInsert);
```

Inserindo Caminho da Foto no SQL

Quando criamos o aluno, deixamos **'usuario.png'** como padrão. Quando for enviada alguma foto, nosso arquivo **multer.js** já irá adicionar a imagem na pasta **img**, mas também devemos alterar esse caminho no SQL.

```
// Nome padrão da foto  
let foto = 'usuario.png';  
  
// Verificando se foi enviada alguma foto  
if (req.file) {  
  // Pegar novo nome da foto  
  foto = req.file.filename;  
}
```



13 Editar Informações dos Alunos

Nós ainda não criamos uma tela em HTML que seja para editar as informações de algum aluno, isso porque podemos utilizar a tela de cadastro.

A única questão é que as informações passadas pro **EJS são diferentes**, ou seja, teremos que **duplicar a tela de cadastro** para modificarmos algumas coisas específicas.

A tela "editar aluno", virá **preenchido** com as informações do SQL, e tudo que for **modificado** será enviado pro Banco.

Editar informações do aluno



Nome

Idade

Sexo

Sala

ATUALIZAR

Temos que informar para o **routes.js** que junto a URL, será enviado um valor, que nesse caso é o ID do aluno que queremos modificar.

```
route.get('/editarAluno/:id', editar.alunos);
```

No controller **editar.js**, devemos receber o ID passado na URL.

Quando queremos receber algo na URL, deixamos de acessar pelo body e acessamos pelo **params**.

```
const parametro = req.params.id;
```

Agora que recebemos o ID:

1. Utilize o **findByPk** para encontrar todos os dados do aluno que iremos modificar.

Obs: **findByPk(ID, { raw, attributes })**

2. Encontre o ID e o Nome de todas as salas do Banco.

3. Faça o render com os dois parâmetros acima.

editar

```
async alunos(req, res){  
  
  // Recebendo o id da URL  
  const parametro = req.params.id;  
  
  const alunos = await aluno.findByPk(parametro, {  
    raw: true, //Retorna os somente os valores de uma tabela, sem os metadados  
    attributes: ['IDAluno', 'Nome', 'Idade', 'Sexo', 'Foto', 'IDSala']  
  });  
  
  const salas = await sala.findAll({ raw: true, attributes: ['IDSala', 'Nome'] });  
  
  res.render('../views/editarAluno', {salas, alunos});  
  
}
```

4. No EJS:

- Insira todas as informações do Aluno nos campos.
- Deixe o select com a sala na qual ele pertence, retornando o **ID da sala**.
- Vamos utilizar o método **Post** e também passar o **ID do Aluno na URL**.

ejs

```
<div class="edit-box">
```


<h2>Editar informações do aluno</h2>

<form action="/editarAluno/<%= alunos.IDAluno %>" method="post" enctype="multipart/form-data">

<div class="max-width">

<div class="image-container">

</div>

<input type="file" id="flImage" name="foto" accept="image/*">

</div>

<div class="user-box">

<label>Nome</label>

<input type="text" value="<%= alunos.Nome %>" name="nome">

</div>

<div class="user-box">

<label>Idade</label>

<input type="text" value="<%= alunos.Idade %>" name="idade">

</div>

<div class="user-box">

<label>Sexo</label>

<input type="text" value="<%= alunos.Sexo %>" name="sexo">

</div>

<div class="user-box">

<label>Sala</label>

<select id="sala" name="sala">

<% for (let i=0; i<salas.length; i++) { %>

<option value='<%= salas[i].IDSala %>' <%= alunos.IDSala == salas[i].IDSala ? 'selected' : '' %>> <%= salas[i].Nome %> </option>

<% } %>

</select>

</div>

<input class="btn btn-primary" type="submit" value="Atualizar" />

```
</form>
</div>
```

Atualizando Aluno no Banco de Dados

1. Receba o **ID** do aluno na **URL** e os dados do **form** pelo **body**.
2. Dê o **update** dos novos dados no **SQL**.

```
async adicionar(req, res){

  const dados = req.body;
  const id = req.params.id;

  // Dando upgrade nas novas informações
  await aluno.update({
    Nome: dados.nome,
    Idade: dados.idade,
    Sexo: dados.sexo,
    IDSala: dados.sala
  },
  {
    where: { IDAluno: id }
  });

  res.redirect('/');

}
```

3. Adicione o **post** do **editar** no **route.js**

```
// Iniciando Route do Express
const express = require('express');
const route = express.Router();

// Iniciando e importando Multer
const multer = require("multer");
```

```
const config = require('./src/config/multer');

// Importando os Controllers
const home = require('./src/controllers/home');
const cadastro = require('./src/controllers/cadastro');
const editar = require('./src/controllers/editar');

// Iniciando as rotas
route.get('/', home.pagInicialGet);
route.post('/', home.pagInicialPost);

route.get('/cadastroSala', cadastro.sala);
route.post('/cadastroSala', cadastro.salaInsert);

route.get('/cadastroAluno', cadastro.aluno);
route.post('/cadastroAluno', multer(config).single('foto'), cadastro.alunoInsert);

route.get('/editarAluno/:id', editar.alunos);
route.post('/editarAluno/:id', multer(config).single('foto'), editar.adicionar);

module.exports = route;
```

Update das fotos

Nos ainda precisamos fazer o update das fotos no SQL, o que faremos somente se o usuário mudar a foto do aluno. Então no código do arquivo que acabamos de criar:

1. Faça uma verificação para saber se o usuário fez o envio de alguma foto.
2. Dê o update do nome da nova foto enviada pelo usuário.

foto

```
// Se foi enviado alguma foto
if (req.file) {

    // Update da nova foto no DB
```

```
    await aluno.update(  
      { Foto: req.file.filename },  
      { where: { IDAluno: id } }  
    );  
  
  }  
}
```

Excluindo as antigas fotos

Agora que estamos armazenando as novas fotos que foram enviadas pelo usuário, temos que excluir as antigas da nossa pasta, pois não será mais útil. Utilizaremos a biblioteca **fs(File System)**, que dispensa a necessidade de instalação.

Então vamos importa-la no início do nosso código.

Vamos receber o nome da **antiga foto** do SQL, utilizando o **ID** do aluno recebido.

E por último, vamos utilizar a função **unlink(caminho, erro)** para excluir a foto, desde que não seja nossa foto padrão, a **usuario.png**.

```
// Se foi enviado alguma foto  
if (req.file) {  
  
  // Recebendo a antiga foto do aluno  
  const antigaFoto = await aluno.findAll({  
    raw: true,  
    attributes: ['Foto'],  
    where: { IDAluno: id }  
  });  
  
  // Excluindo a foto da pasta  
  if (antigaFoto[0].Foto !== 'usuario.png') fs.unlink(`public/img/${antigaFoto[0].Foto}`, ( err => { if(err) console.log(err); } ));  
  
  // Update da nova foto no DB  
  await aluno.update(  
    {Foto: req.file.filename},  
    {where: { IDAluno: id }}  
  );  
  
}
```


14 Desafios

Agora que o site está funcional, vamos adicionar alguns recursos novos, colocando em prática tudo que foi visto até agora.

1. Crie uma página onde seja possível **editar as salas** criadas, alterando seu nome ou sua capacidade.
2. Mostre a **quantidade de vagas disponíveis** na página principal de cada sala que o usuário selecionar.
3. Mostre no cadastro dos alunos, apenas as salas que **não preencheram** toda a sua **capacidade**.
4. Crie a opção de **excluir as salas ou os alunos** que o usuário desejar.

15 Desafio dos Desafios

Desafio

Ao invés do usuário inserir a idade, peça a **data de nascimento**, e continue mostrando sua **idade na página principal**.

Adicione o **campo** de idade **mínima** ou **máxima** na criação das Salas.

Apenas será aceito os Alunos que forem de **idade compatível**.

16 Erros/Problemas

Erro de proxy para instalação das bibliotecas

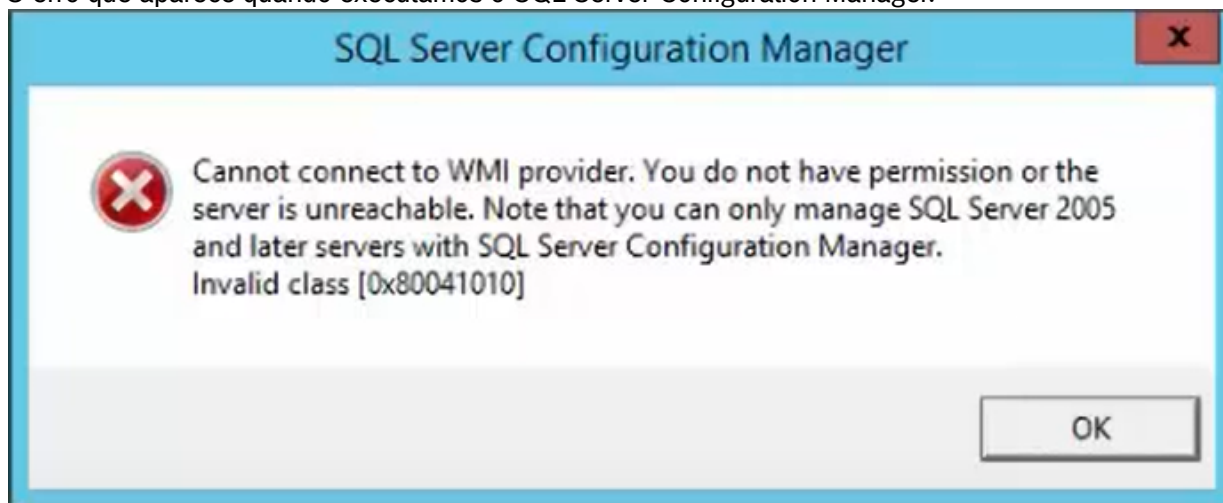
Digite os comandos abaixo no terminal do VSCode ou no CMD como administrador

```
npm config set proxy http://USUARIO_DE_REDE:SENHA_DE_REDE@10.224.200.26:8080  
npm config set https-proxy=http://USUARIO_DE_REDE:SENHA_DE_REDE@10.224.200.26:8080  
npm config set http-proxy=http://USUARIO_DE_REDE:SENHA_DE_REDE@10.224.200.26:8080
```

Computer Management sem as config do SQL

Esse erro ocorre quando ele não encontrou o **WMI**, que é utilizado para configurar essa parte do SQL.

O erro que aparece quando executamos o SQL Server Configuration Manager.



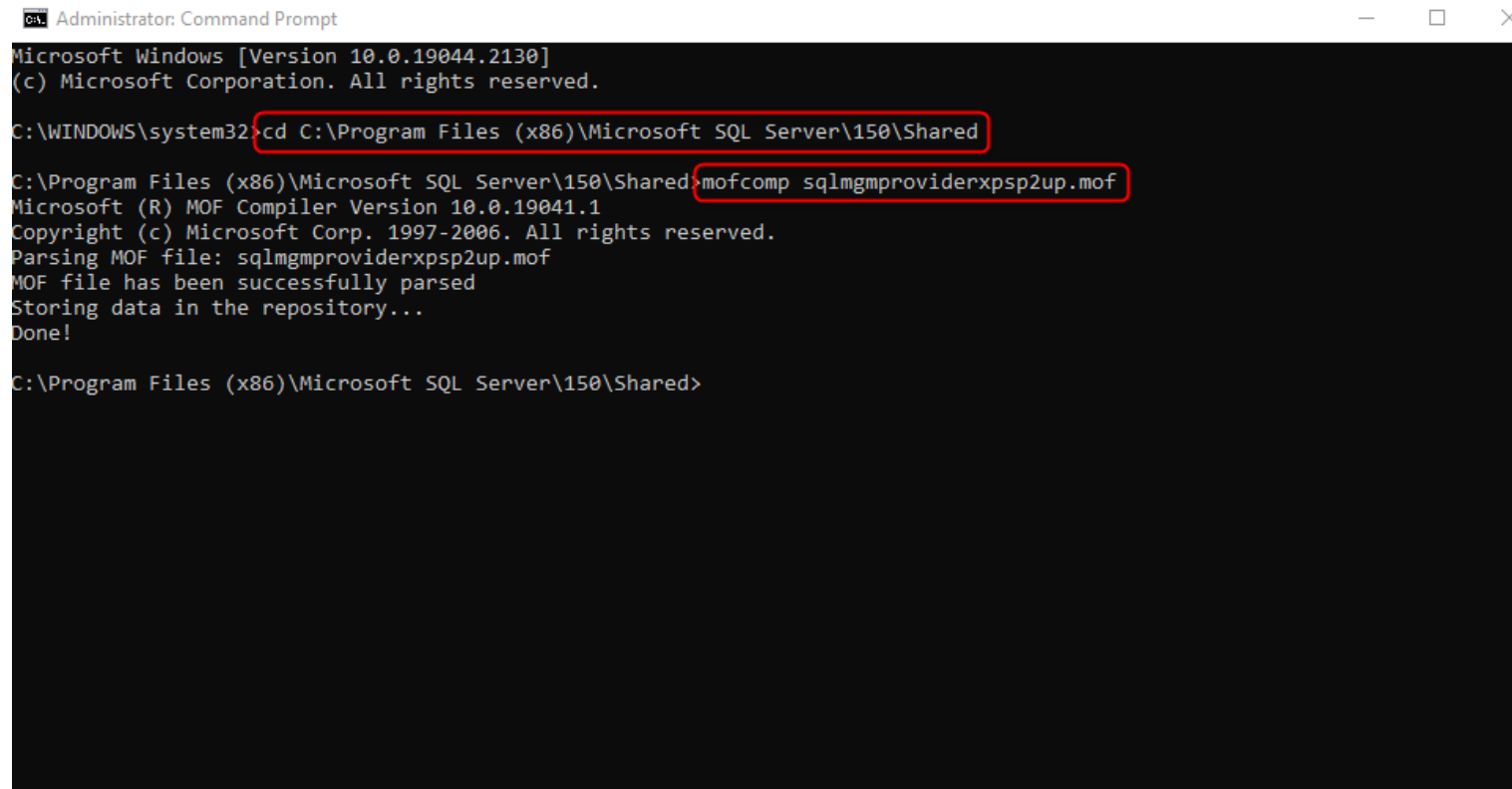
Para resolvermos esse erro, encontre o arquivo "**sqlmgmproviderxpsp2up.mof**".

Abaixo está um exemplo de local desse arquivo, pois pode estar em outra pasta perto desse fornecido abaixo:

C:\Program Files (x86)\Microsoft SQL Server\150\Shared

1. Execute o CMD como administrador.
2. Vá até esse local acima com o **cd** do **cmd**.

3. Execute o comando **mofcomp sqlmgmproviderxpsp2up.mof**



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.19044.2130]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd C:\Program Files (x86)\Microsoft SQL Server\150\Shared

C:\Program Files (x86)\Microsoft SQL Server\150\Shared>mofcomp sqlmgmproviderxpsp2up.mof
Microsoft (R) MOF Compiler Version 10.0.19041.1
Copyright (c) Microsoft Corp. 1997-2006. All rights reserved.
Parsing MOF file: sqlmgmproviderxpsp2up.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!

C:\Program Files (x86)\Microsoft SQL Server\150\Shared>
```

Vídeo de exemplo - Fazendo os passos acima



Sorry, the widget is not supported in this export.
But you can reach it using the following URL:
<https://www.youtube.com/watch?v=1plejl1RL-k>