O que é Node.js?

- O **Node.js** é uma runtime de JavaScript;
- Ou seja, **uma biblioteca utilizada por um compilador** durante a execução do programa;
- Que está construída na **V8 engine** (escrita em C++) da Google;
- Possibilitando criar softwares em JS no lado do servidor;
- Temos então um código JS rodando em C++ para garantir **alta performance**;

O que é npm?

- O **npm** é um gerenciador de pacotes do Node;
- Vamos poder utilizar **bibliotecas de terceiros**, baixando elas pelo npm;
- E também **executar determinados scripts** no nosso programa;
- Dificilmente um software em Node.js não utiliza o npm;
- Os módulos externos ficam numa pasta chamada **node_modules**;
- Ela deve ser descartável, ou seja, a cada instalação do projeto baixamos todos os pacotes novamente;

Instalação Node Windows

- O download do Node.js é feito no site oficial: **nodejs.org;**
- Vamos baixar um arquivo **.msi**, que é o instalador;
- É interessante saber que o **npm** vem junto do Node;
- Após a instalação podemos testar o Node e o npm em um terminal, para validar a instalação;
- Note que os binários precisam estar no **PATH**, este passo está na instalação do Node;

Instalação Node Linux

- O download do Node.js é feito no site oficial: **nodejs.org;**
- Vamos baixar um executável que inicializa a instalação;
- Note que o **npm** vem junto do Node;
- Após a instalação precisamos validar em um **terminal**;

Instalação VS Code

- Ao longo do curso o editor utilizado será o **VS Code**;
- Ele tem algumas vantagens em trabalhar com Node também, como o **terminal integrado**;
- Além de ser **atualizado constantemente**, é mantido pela Microsoft;
- Muito utilizado no **ambiente corporativo**;
- Vamos instalar!

Terminal para Windows

- Caso seja necessário, vamos utilizar o **cmder** como terminal no Windows;
- Este software **não quer instalação**;
- E é **muito semelhante ao terminal do Linux**;
- Geralmente utilizamos **servidores em Linux**, então estar acostumado com este tipo de terminal é de grande ajuda;
- Além disso, algumas ferramentas como Node e Docker, por exemplo, utilizam muito o terminal;

Executando o Node

- Na maioria das vezes **estaremos executando o Node via arquivos** do nosso projeto;
- Porém é possível também **executá-lo via terminal;**
- Basta digitar: node
- O Node.js possui todas as **funcionalidades de JavaScript**;
- O nome do recurso é **REPL** (Read Evaluate Print Loop)
- Vamos experimentar!

Nosso primeiro programa

- Agora vamos criar algo mais sólido, um **programa simples baseado em um arquivo**;
- A extensão dos arquivos de Node serão **.js**
- Vamos executar o arquivo com o comando: **node <arquivo>**
- O código será interpretado e o programa executado;
- Vamos lá!

Utilizando um módulo

- Agora vamos **evoluir o nível de nossos programas**;
- Importaremos um módulo do Node: o **File System**;
- Este módulo serve para trabalhar com diretórios, arquivos e etc;
- E ele é um **Core Module**, ou seja, não é necessário instalar;
- Veremos mais sobre modules ao longo do curso;
- Podemos importar módulos com a instrução **import**;

Repositório do curso

- Todos os arquivos do curso estão no **GitHub**, em um repositório público;
- **É interessante que você faça o download** para poder acompanhar com o código e também resolver possíveis problemas;

Como tirar o melhor proveito

- **Faça todos os exercícios**, eles foram pensados em agregar no aprendizado e fixar conhecimento;
- **Crie o código** junto comigo!
- Tente **responder as dúvidas** de outros alunos;
- **Crie projetos práticos**, teste coisas novas;
- **Dica extra:** ouça e depois pratique;

Tarefa 01

1. Crie um novo projeto de Node.js;
2. Crie um arquivo para a aplicação com o nome programa;
3. No arquivo crie duas variáveis e imprima a soma delas;
4. Execute o arquivo e verifique a resposta no terminal;

Introdução

conclusão da sessão

Fundamentos de Node.js

introdução da sessão

O que são módulos

- Módulos são **scripts reaproveitáveis**, que utilizamos bastante programando em Node;
- Eles são divididos em três categorias;
- **Internos:** módulos que nós desenvolvemos; (criado por nós)
- **Core Modules:** módulos que vem com o Node.js; (utilizamos o file sytem no exemplo anterior)
- **Externos:** módulos que instalamos via npm; (feitos por desenvolvedores externos)

Módulos internos

- Os **módulos internos** são criados nas pastas do nosso projeto;
- Precisamos **exportar** o módulo;
- Podemos utilizar a instrução **module.exports**;
- E **importar** onde precisamos utilizar;
- Para importar vamos utilizar a instrução **require**;
- Vamos criar um módulo!

Export e Import

- Com o Node.js também é possível utilizar o **export e import do ES6**;
- São funcionalidades mais modernas de **importação e exportação**;
- Com **mais recursos** do que as que vimos anteriormente;
- Para isso precisamos modificar os nossos arquivos para a extensão **.mjs**;
- E então podemos exportar uma com **export default**;
- E importar com **import**, uma única função, caso seja necessário;

Core Modules

- No Node temos diversos **Core Modules**, que são os que vêm prontos para serem utilizados;
- Eles resolvem diversos problemas, como: **trabalhar com arquivos e diretórios**, **servir aplicações** e etc.
- **Precisamos importar** estes módulos no projeto para poder utilizar;
- Vamos utilizar um Core Module!

Ler argumentos

- O Node permite o **envio de argumentos via linha de comando**;
- Passamos eles após a instrução de execução do arquivo;
- Os argumentos ficam em um array chamado: **process.argv**
- Onde podemos **fazer um loop e resgatar os valores** enviados;
- Vamos ver na prática!

Módulos externos

- Os módulos externos podem ser instalados via **npm**;
- Para isso precisamos inicializar o npm no projeto, com: **npm init**;
- A partir daí os módulos ficam mapeados e podemos instalar módulos;
- Que são salvos na pasta **node_modules**;
- Podemos instalar módulos com **npm install <nome>**;
- Vamos ver na prática!

Algo prático com argumentos

- Podemos **utilizar os argumentos recebidos** para aplicar no nosso programa alguma lógica;
- Basta **encapsular em variáveis** e depois utilizá-los;
- Ou seja, podemos a partir do terminal, **executar também uma função de um módulo interno nosso**, por exemplo;
- Vamos ver na prática!

Explorando o console

- Temos uma grande variedade de métodos no **console**, como o console.log;
- Podemos **imprimir mais de uma variável** por vez;
- Inserir **variáveis entre strings**;
- Há um método para limpar as mensagens de console;
- Vamos ver na prática!

Melhorando a visualização

- Há um módulo externo chamado **chalk**;
- Ele pode deixar a **visualização do console** mais agradável;
- Fazendo com que seja possível expressar um **feedback com base em cores**;
- Vamos ver na prática!

Lendo entrada de dados

- Podemos ler dados do usuário com o módulo **readline**, um Core Module;
- Neste caso utilizamos o método **question**, que faz uma pergunta a ser respondida pelo usuário;
- Depois podemos **processar a resposta** e entregar um retorno;
- Vamos ver na prática!

Melhorando a leitura de dados

- Há um módulo externo chamado **inquirer**;
- Que é muito mais completo para **resgatar e lidar** com o input do usuário;
- Além disso, é **baseado em Promises**, o que torna sua utilização mais simples;
- Vamos ver na prática!

Event Loop

- O **Event Loop** é um recurso da arquitetura do Node;
- O **Node.js executa uma linha por vez**, e de cima para baixo do código escrito;
- Isso nos ajuda a **evitar problemas de concorrência**, garantindo a execução do código;
- Precisamos apenas cuidar com **bloqueios no fluxo**, como loops infinitos;
- Vamos ver na prática!

Event Emitter

- O **Event Emitter** se comporta como os eventos do navegador;
- Podemos **ativar um código em alguns pontos da aplicação**, como o início;
- É um Core Module chamado **events**;
- Precisamos instanciar a **classe EventEmitter** que vem deste módulo;
- E então utilizar os métodos para atingir nosso objetivo;
- Vamos ver na prática!

Síncrono e Assíncrono

- Em Node.js temos duas opções ao executar métodos;
- Conhecidas como sync e async;
- **Forma síncrona:** o código espera ser totalmente executado para prosseguir;
- **Forma assíncrona:** o código continue progredindo e em um ponto futuro obtém a resposta da execução assíncrona;
- Vamos ver na prática!

Erros no Node

- Temos duas formas principais para **gerar ou evidenciar erros** em Node.js;
- **throw**: uma forma de encerrar um programa, gerando um novo erro;
- **try catch**: uma forma de evidenciar algo que deu errado em um bloco de código e exibir a mensagem de erro;
- Vamos ver na prática!

Tarefa 02

1. Crie um novo projeto que aceite pacotes externos;
2. Instale o inquirer e o chalk;
3. Utilize o inquirer para receber o nome e a idade do usuário;
4. Apresente esta resposta com uma cor de fundo amarela e texto preto;
5. Dica: Você pode utilizar bgYellow e black!
6. Insira um tratamento para um possível erro do inquirer com o catch;

Fundamentos de Node.js

conclusão da sessão

Core Modules

introdução da sessão

Sobre os Core Modules

- Vamos ver diversos Core Modules em detalhes, que são **fundamentais para criação de softwares em Node**;
- **http:** módulo para criar servidores HTTP;
- **path:** extrair informações de paths (caminhos) de arquivos;
- **fs:** file system, leitura e escrita de arquivos;
- **url:** módulo para trabalhar com URLs;

O módulo http

- Podemos criar um **servidor HTTP** com este módulo;
- Ou seja, receber uma requisição e **enviar código HTML** como resposta, por exemplo;
- Vamos utilizar alguns métodos como **createServer**, para criação do servidor;
- E também **listen**, para determinar a porta;
- Vamos ver na prática!

Parando o serviço

- Há alguns serviços do Node que **ocupam a aba do terminal** para continuar rodando;
- Um destes é o módulo http;
- Para parar este serviço basta utilizar: **ctrl + c**;
- Isso **é útil quando há um problema no código** também;

Retornando HTML com o http

- Para retornar HTML precisamos **implementar mais recursos**;
- Podemos adicionar um **status code** no retorno, com a propriedade statusCode;
- Mudar os headers para **text/html**;
- E retornar o HTML pelo **método end** do http;
- Vamos ver na prática!

Atualizações de projeto

- Para emitir as atualizações do projeto **precisamos sempre reiniciar o servidor**;
- Ou seja: salvar, encerrar e reiniciar, este é o processo;
- Isso é **terrível para a produtividade**! xD
- Nas próximas aulas **veremos como contornar este problema**;
- Mas primeiro vamos ver o que acontece!

Módulo url

- O módulo **url** serve para decompor uma URL que passamos para o método **parse**;
- Podemos resgatar: **host, path, search, query** e etc;
- A partir destas informações podemos alterar a lógica do nosso código;
- Vamos ver na prática!

Unindo os módulos http e url

- **Podemos trabalhar com estes módulos juntos** e ter um resultado interessante;
- Com o **http criamos nosso server** e alteramos a resposta baseado na URL acessada;
- **Com url processamos os parâmetros** que vem pela query string, para alterar a lógica do http;
- Vamos ver na prática!

Módulo fs

- O módulo **fs** (File System) serve para trabalhar com arquivos e diretórios;
- Este é também um **Core Module**;
- Podemos **ler e escrever em arquivos**, por exemplo;
- Uma utilização interessante: **logs do sistema**;
- Vamos ver na prática!

Escrevendo em arquivos

- Podemos criar e escrever em arquivos também, utilizando o método **writeFile**;
- Esta escrita pode estar associada a um conjunto de operações;
- Como o **envio de informações de um usuário**, por exemplo;
- Vamos unir mais uma vez os módulos na prática!

Atualizando um arquivo

- O **writeFile** substitui tudo que está em um arquivo;
- **E se quisermos atualizar?**
- Para este fim utilizamos o **appendFile**;
- Que tem a mesma utilização que write, porém nos permite unir conteúdo;
- Vamos ver na prática!

Removendo um arquivo

- Para remover um arquivo com o fs utilizamos o método **unlink**;
- Precisamos **passar o arquivo como parâmetro**;
- Temos a possibilidade de checar se houve algum erro, a partir da **callback** retornada;
- Vamos ver na prática!

Renomeando um arquivo

- Para renomear um arquivo com o fs utilizamos o método **rename**;
- Precisamos **passar o arquivo como parâmetro**;
- E também o novo nome!
- Neste método também podemos verificar algum eventual erro, pela **callback**;
- Que pode ser ativado quando o arquivo alvo não existe;
- Vamos ver na prática!

Rotas com Node.js puro

- Podemos criar um **sistema de roteamento** simples com Node.js e seus Core Modules;
- As rotas são basicamente as **páginas que acessamos** nos sites;
- Vamos falar mais sobre esse recurso em outros módulos;
- A ideia é **identificar os arquivos acessados pela URL** e retorná-los, se existirem;
- Vamos ver na prática!

Detalhes de arquivos

- Podemos **saber mais sobre os arquivos** que temos acesso;
- Utilizamos o método **stat** de fs;
- Com ele temos informações de: tamanho, data de criação, se é arquivo ou diretório e etc;
- Vamos ver na prática!

Módulo path

- Com o **path** conseguimos extrair diversas informações sobre caminhos e arquivos;
- Este também é um **Code Module**;
- **Algumas infos possíveis são:** nome do diretório, nome do arquivo, extensão do arquivo e etc;
- Vamos ver na prática!

Path absoluto e formar path

- Com a função **resolve** é possível saber qual o path completo até o arquivo alvo;
- E com a função **join** é possível formar um path dinâmico, com variáveis e valores fixos;
- São duas funções muito importantes;
- Vamos ver na prática!

Trabalhando com diretórios

- Com o módulo fs também podemos **trabalhar com diretórios** (pastas);
- O método **exists** pode evidenciar se um diretório existe ou não;
- E o método **mkdir** pode criar um diretório;
- Vamos ver na prática!

Módulo os

- Com o módulo **os** podemos extrair informações do sistema operacional;
- Este também é um **Core Module**;
- Vamos ver na prática!

Core Modules

Conclusão da sessão

npm Fundamental

Introdução da sessão

O que é npm?

- É o principal **gerenciador de pacotes** do Node.js;
- A sigla significa: **Node Package Manager**;
- A maioria dos projetos que vamos trabalhar em Node, **também terá atuação do npm**;
- Podemos não só instalar pacotes, mas também **configurar o projeto e rodar scripts** por meio do npm;
- A criação de um projeto gera sempre um arquivo, o **package.json**;

Criando um projeto com npm

- Para iniciar um projeto, podemos utilizar o comando **npm init**;
- Seremos questionados para **configurar algumas opções iniciais**;
- Como o **nome do projeto**;
- E então um **package.json** condensando as informações é criado;
- Isto caracteriza o início do projeto com npm e Node;
- Vamos ver na prática!

Instalando um pacote

- Para instalar um pacote vamos utilizar o comando **npm install <nome>**
- Onde **<nome>** será substituído pelo nome do pacote;
- Quando fazemos desta maneira a instalação uma pasta **node_modules** é criada;
- Nela **todos os arquivos de módulos** de terceiro são armazenados;
- Sempre que rodarmos o comando npm install, a pasta node_modules é recriada com todos os módulos do package.json;

Instalando um pacote como dev

- Há uma possibilidade de **instalar pacotes apenas para o ambiente de desenvolvimento**;
- Utilizamos a flag **--save-dev**;
- Isso faz com que ele seja **separado no package.json dos demais**;
- E então na build de produção não instalaremos estes módulos;
- Um exemplo: servidor para ambiente local, como o Nodemon;
- Vamos ver na prática!

Atualização de pacotes

- Constantemente **os pacotes do npm são atualizados** por seus desenvolvedores;
- Temos então o comando **npm update**;
- Que vai fazer a atualização de todos os pacotes instalados no package.json;
- É possível atualizar um pacote específico com **npm update <nome>**

Criando scripts

- É possível **criar rotinas com o npm** também;
- Ou seja, executamos uma série de comandos com apenas um;
- Utilizamos **npm run <script>**
- Onde **<script>** é o nome da sequência de comandos que configuramos;
- Vamos ver na prática!

Instalando pacote global

- Um **pacote global** não fica salvo na pasta node_modules do projeto;
- Ele fica salvo no computador do usuário;
- A vantagem é que **podemos acessá-lo em qualquer local** via terminal;
- Utilizamos a **flag -g** em node install;
- Vamos ver na prática!

Executando scripts com npx

- **Alguns pacotes são scripts executáveis**, que resultam em alguma ação no nosso computador;
- Como por exemplo a **instalação do React**, que é feita pelo **npx**;
- Desta maneira uma série de processos são simplificados por este executor;
- Vamos ver na prática!

Remover pacote com npm

- Para remover um pacote utilizamos o comando: **npm uninstall <nome>**
- Substituindo **<nome>** pelo nome do pacote;
- Isso faz com que o pacote seja **removido do package.json** também;
- Vamos ver na prática!

npm Fundamental

Conclusão da sessão

Express

Introdução da sessão

O que é Express?

- Um **framework para Node.js** muito utilizado;
- Serve para criarmos **aplicações web**;
- Nele podemos criar rotas, renderizar HTML, conectar a um banco de dados;
- **O Express torna a criação de apps muito mais simplificada**, do que com os Core Modules;
- Vamos instalá-lo!

O que são rotas?

- Rota é um conceito **super importante e presente** em aplicações web;
- Basicamente são as **URL's que acessamos**;
- Se criamos uma rota **/produtos**, podemos acessar através da URL www.nossosite.com/**produtos**;
- **Quando o usuário acessa podemos acessar várias lógicas**, como carregar produtos do banco de dados;
- Ou seja, rotas **são uma ponte** entre o usuário e a lógica da aplicação;

Primeiros passos

- O **setup inicial do Express é simples**, mas precisamos seguir alguns passos;
- **Importar o Express** e invocá-lo;
- **Definir uma porta base** para a aplicação;
- **Criar uma rota** (URL que será acessada);
- **Executar o método listen** na porta especificada;
- Vamos ver na prática!

Renderizando HTML

- Para enviar HTML como resposta utilizamos o método **sendFile**;
- Isso faz com que o arquivo seja **renderizado no navegador**;
- Precisamos acessar o arquivo por meio do diretório base, isso requer o módulo **path**;
- Vamos ver na prática!

Problema de atualização

- Precisamos toda vez **reiniciar o servidor** para receber as atualizações, isso é muito custoso;
- Vamos então instalar o módulo **Nodemon**;
- Que **a cada vez que o arquivo é salvo reinicia o projeto**, facilitando nossa vida;
- Vamos salvar como **dependência de desenvolvimento** (--save-dev);
- Vamos ver na prática!

Middlewares

- Middlewares são códigos que podem ser executados no **meio/entre (middle) de alguma ação e outra**;
- Por exemplo: **verificar se usuário está logado**, podemos ter um para esta verificação;
- O método que nos dá acesso a utilizar middlwares é o **use** no Express;
- Vamos ver na prática!

Parâmetros por URL

- Podemos resgatar os parâmetros da URL por meio do **req**;
- Acessamos **req.params.<nome>**;
- Onde nome **deve ser o que está definido na URL do Express**;
- Que fica desta forma: **/users/:id**
- Neste caso estaríamos buscando o usuário no banco de dados pelo id;
- E o parâmetro vem pela URL;

Enviando dados por POST

- Para enviar os dados vamos precisar **criar um form e mandar os dados via POST** para alguma URL;
- No Express precisamos colocar alguns **middlewares** como o **express.json** para ler os dados do body;
- E também uma roda que vai receber estes dados, utilizando o **método post** do Express;
- Vamos ver na prática!

Módulo de rotas

- Podemos **unir várias rotas em um módulo**, isso vai deixar nosso código mais organizado;
- Normalmente **criamos uma pasta ou arquivo** que contém estas rotas;
- Que representam uma entidade em comum, como **users**;
- Vamos utilizar um novo objeto chamado **Router**, e colocar as rotas nele;
- Depois precisamos **exportá-lo e importar no arquivo principal**;

Colocando CSS

- Para inserir CSS nas páginas e arquivos estáticos **vamos precisar de um middleware**;
- Que é o **express.static**;
- **Precisamos colocar um diretório base**, que normalmente é o public;
- E criar os estáticos a partir desta pasta;
- No HTML podemos **acessar o caminho relativo após a pasta definida**, e pronto!

Criando uma página 404

- Para criar uma página 404, ou seja, **quando o usuário acessa uma página que não existe**;
- Basta criar um novo middleware **abaixo de todas as rotas**, que responde com este status;
- E **enviar um arquivo de template** referente a este página;
- Vamos ver na prática!

Tarefa 03

1. Crie um novo projeto com Express;
2. Adicione nodemon e coloque um script chamado serve para rodar o projeto na porta 5000;
3. Crie duas páginas da sua escolha;
4. Adicione CSS as páginas, mude a cor de fundo e a cor da fonte, pelo menos;
5. Separe as rotas no recurso de Router do Express;

Express

Conclusão da sessão

Template Engine

Introdução da sessão

O que é Template Engine?

- Uma forma de deixar o **HTML dinâmico**, **inserindo variáveis** do back-end no front-end;
- Podemos também criar **layouts**, que são reaproveitados;
- É essencial para projetos usam banco de dados, que **não são estáticos**;
- Temos diversos disponíveis: **EJS, Pug e Handlebars**, por exemplo;
- **Todos atingem o mesmo objetivo**, porém há algumas diferenças de setup e funcionalidades;

Conhecendo o Handlebars

- O **Handlebars** é uma das template engines mais utilizadas;
- Colocamos os dados dinâmicos no HTML entre **{{ }}** para serem impressos;
- Podemos **criar condicionais e também loops** no template;
- **Conhecido pela sua sintaxe limpa no front,** nos força a não executar lógica no HTML;
- O nome do pacote é **express-handlebars**;

Instalando o Handlebars

- Vamos precisar instalar o **Express e o Handlebars**, para o correto funcionamento;
- Podemos também utilizar o **Nodemon**, para nos ajudar;
- No index precisamos **importar** os pacotes instalados;
- E também adicionar ao Express a **engine** do Handlebars;
- Criaremos uma **view** no diretório views, com a **extensão handlebars**;
- Utilizamos o método **render** para enviar esta view para a requisição;

Criando layouts

- Podemos criar layouts para **reaproveitar código** entre páginas;
- Vamos criar uma **pasta chamada layouts** em views;
- E nela **criamos o template** que será repetido;
- Colocamos uma tag especial **{{ body }}**;
- Para que neste local o ‘corpo’ do site seja inserido;
- Em **todas as nossas views** agora o layout é repetido;

Passando dados para a view

- Vamos passar os dados por meio do **método render**;
- Enviamos **um objeto** com chaves e valores;
- E isso nos possibilita acessar estes **dados no template**;
- Vamos utilizar a sintaxe de **{{ dado }}**;
- E o dado será impresso!

Utilizando condicionais

- Utilizar uma **estrutura condicional** nos permite mais flexibilidade no layout;
- Podemos utilizar o **if** no Handlebars;
- A sintaxe é **{{#if algumacoisa }}** … **{{/if }}**
- **Só imprime o que está entre as condicionais**, se o resultado dela for verdadeiro;
- Precisamos encaminhar o dado a ser validado **pelo backend**;

Utilizando o else

- O else é um **complemento do if**;
- Utilizamos no Handlebars para a exibição de outra parte do layout, **caso a condição seja falsa**;
- Isso nos dá mais flexibilidade ainda!
- A sintaxe é: **{{#if alguma coisa}}** … **{{else }}** … **{{#/if }}**

Estrutura de repetição

- As estruturas de repetição no Handlebars são feitas pelo operador each;
- A sintaxe é **{{#each }}** … **{{/each }}**
- Em um array podemos chamar os itens com: **{{this }}**
- Então cada um dos itens é acessado na view;
- Como o Handlebars prega **um template mais limpo**, devemos mandar apenas o necessário para o front-end;

Utilizando o with

- O **with** nos permite **abstrair um objeto**;
- Ou seja, podemos acessar as propriedades **sem nos referenciarmos sempre ao objeto antes**;
- A sintaxe é: **{{#with objeto}}** … **{{/with}}**
- Desta maneira nossa código fica ainda mais simples;

Conhecendo os partials

- **Os partials são como mini templates**, que precisam ser repetidos em diversos locais da nossa aplicação;
- Precisamos realizar algumas **modificações na implementação do Handlebars**;
- Os partials geralmente ficam em **views/partials**;
- E utilizamos a sintaxe: **{{> partial}}** para chamá-lo no projeto;

CSS com Handlebars e Express

- A inclusão de CSS no Handlebars **é muito semelhante a que realizamos apenas com Express**;
- Precisamos **definir a pasta dos arquivos** estáticos;
- **E vamos linkar o CSS com o nosso layout** em comum para todas as páginas;
- Isso torna possível a estilizar os nossos projetos;

Tarefa 04

1. Crie um projeto com Express, Handlebars e Nodemon;
2. Crie também uma rota para uma home, que exibe alguns produtos de um array de objetos;
3. Utilize o recurso de layout do Handlebars para a base do HTML;
4. Adicione CSS e modifique alguns estilos a sua escolha;
5. Os produtos precisam ter um link, que exibe as páginas individuais de cada um, dica: URL dinâmica do Express, aproveite o array da home;

Template Engine

Conclusão da sessão

Node.js com MySQL

Introdução da sessão

O que são bancos relacionais?

- Os **bancos de dados** relacionais são constituídos por algumas entidades;
- **Banco de dados:** Um banco para guardar os dados do projeto;
- **Tabelas:** Onde categorizamos os dados e os inserimos;
- **Colunas:** Onde separamos os dados em uma tabela;
- **Dados:** O que é inserido, modificado, atualizado e removido em uma tabela;
- **Relacionamentos:** Ligações entre as tabelas;

O que é MySQL?

- O MySQL é um **SGBD** (Sistema Gerenciador de Banco de Dados);
- Que vai nos ajudar a trabalhar com **bancos relacionais**;
- Também **é o mais utilizado atualmente** em sistemas e aplicações;
- Muitos **projetos de Node.js utilizam MySQL**;
- Precisamos **instalar** o software **e deixar ele em execução** para que o Node possa se conectar aos bancos que temos disponíveis;

Integração do MySQL e Node.js

- Primeiramente vamos precisar **instalar o driver**, que é um pacote chamado **mysql**;
- Vamos **instalá-lo com o npm**!
- Depois **precisamos conectar ao nosso banco** de dados;
- Vamos inserir informações como: **host, user, password e o banco**;

Criando a nossa tabela

- Para manipular os dados do sistema **vamos precisar criar uma tabela**;
- Faremos isso via **Workbench**, criando uma tabela chamada **books**;
- Onde vamos cadastrar livros;
- Estes livros vão precisar de **duas informações**: título e número de páginas;
- Vamos criar a tabela!

Inserindo dados

- Para inserir dados no banco vamos precisar **criar e executar uma query**;
- A query em si **é uma string**, seguindo os padrões do MySQL;
- Já para executar vamos utilizar o **método query** do pacote mysql;
- E nesta aula faremos o uso da instrução **INSERT**, que insere dados;
- Vamos ver na prática!

Resgatando os dados

- Para resgatar dados vamos precisar criar uma query, que será um **SELECT**;
- **Podemos escolher quais dados são retornados**, determinando as colunas;
- E podemos imprimi-los nas nossas **views**;
- Vamos ver na prática!

Resgatando os dados específicos

- Para resgatar um dado específico vamos precisar utilizar o **WHERE**;
- Desta maneira conseguimos **filtrar por uma coluna e um valor**;
- Poderemos então achar os livros pelos seus ids;
- Vamos ver na prática!

Editar dados primeiro passo

- Para **editar** algum dado temos antes **alguns preparos a realizar**;
- Primeiramente vamos resgatar o dado, como na aula anterior;
- E normalmente **preenchemos o formulário** de dados com os dados que foram resgatados (**SELECT**);
- Isso faz com que o usuário lembre dos dados cadastrados e possa escolher o que editar;
- Vamos lá!

Finalizando a edição

- Para concluir esta etapa precisamos **criar uma nova rota como POST**;
- Isso porque o **navegador só consegue interpretar dois verbos atualmente** (GET ou POST);
- E então faremos uma query de **UPDATE** para processar a atualização;
- Note que precisamos **passar o id** do livro neste formulário também;
- Vamos para a prática!

Remover itens

- Para remover um item vamos utilizar a query **DELETE**;
- Precisamos enviar para a rota **um POST com o id do item** a ser removido;
- Este processo podemos fazer em **uma única etapa**;
- Vamos ver na prática!

Connection Pool

- **Connection Pool** é um recurso para otimizar as conexões, **criando um cache** e permitindo sua reutilização;
- O **driver mysql** tem este recurso desenvolvido, podemos aplicá-lo;
- Este recurso também **controla as conexões abertas**, fechando assim que se tornam inativas;
- Vamos aplicá-lo!

Preparando a query

- Uma forma de nos defendermos de **SQL Injection**;
- Podemos utilizar **interrogações** em vez dos valores;
- **E substituir através de um segundo passo**, para a query ser executada corretamente;
- **Esta técnica deve ser utilizada** em qualquer programa que vá para a produção;
- Vamos ver na prática!

Node.js com MySQL

Conclusão da sessão

Sequelize

Introdução da sessão

O que é ORM?

- **Object Relational Mapper** (ORM);
- **Abstrai a complexidade das queries**, para trabalharmos com métodos;
- Nos **concentramos na regra de negócios** e não nos comandos SQL;
- Uma ORM muito utilizada para Node.js é a **Sequelize**;
- Em alguns casos **uma ORM pode trazer prejuízos de performance**;
- A **query pura executa mais rapidamente** do que a ORM;
- E temos código “gerado”, ou seja, não vemos por baixo dos panos;

O que é Sequelize?

- **Uma ORM** para Node.js;
- Ela é **baseada em promises** (then, catch);
- Tem integração para **vários bancos relacionais**, como o MySQL;
- **Precisamos sempre criar um Model**, que é o reflexo da nossa tabela em uma classe;
- Precisamos instalar o pacote, já que é um **módulo externo**;

Instalando o Sequelize

- Para instalar o Sequelize utilizamos o pacote **sequelize**;
- E para conectar precisamos passar os mesmos dados que no outro pacote: **banco, usuário e senha**;
- **Instanciando** a classe Sequelize;
- É possível checar a conexão com o **método authenticate**;
- Vamos ver na prática!

Criando um Model

- Para criar um **Model** temos que instanciar uma classe que representará uma tabela;
- Um Model User **cria uma nova tabela chamada users**;
- Colocamos os **campos e os tipos** dele como propriedades do Model;
- Futuramente ele será utilizado para as **operações entre aplicação e banco**;
- O método **sync** faz a criação das tabelas baseada nos models;

Inserindo dados

- Para inserir um dado **vamos precisar do Model** que criamos, ou seja, importar ele no arquivo de execução do comando;
- O método a ser utilizado é o **create**;
- Ele leva como parâmetro **todos os campos**, e insere o registro na tabela;
- Vamos ver na prática!

Lendo dados

- Para ler os dados de uma tabela vamos utilizar o **método fetchAll**;
- Que também requer o **model**, no nosso caso o **User**;
- Os dados vem em um **objeto especial**, para transformar em um array de objetos **temos que inserir um parâmetro**;
- Que é o **raw** como true;
- Vamos ver na prática!

Utilizando o WHERE

- Para filtrar dados com o Sequelize utilizando o WHERE, **precisamos inserir um novo parâmetro**;
- Que será o **where**, um objeto onde colocamos a propriedades e valores que queremos filtrar;
- E para retornar apenas um resultado podemos utilizar o método **findOne**;

Removendo itens

- Para remover itens utilizando o método **destroy**;
- A função vai ficar bem parecida com a de resgatar um usuário;
- Só que será um **POST**, e que efetua a remoção, depois redirecionamos;
- **Precisamos também criar um formulário** no front-end;
- Vamos ver na prática!

Editando itens

- O primeiro passo da edição é **resgatar os dados do item**;
- Com isso podemos **preencher o formulário**, para depois fazer o UPDATE;
- Vamos utilizar o método **findOne**, igual na rota de visualização;
- E fazer o preenchimento do form na nossa **nova view**;
- Vamos a prática!

Atualizando o dado no banco

- Para realizar a query de UPDATE vamos utilizar o **método update** do Sequelize;
- Nele passamos o **objeto atualizado** do item;
- E também filtramos por meio do **atributo where**, que item vamos atualizar;
- Vamos ver na prática!

Refazer as tabelas

- Podemos forçar a **reconstrução das tabelas**;
- No método **sync**, onde são sincronizados os **models e as tabelas**;
- Vamos colocar um atributo: **force** como true;
- Exemplo: `sync({ force: true })`
- Note que os dados são perdidos neste processo;
- Vamos ver na prática!

Relacionamentos

- Em bancos relacionais podemos criar **relacionamentos entre as tabelas**;
- Para concretizar isso no Sequelize vamos precisar de **dois Models**, ou seja, precisamos criar mais um no nosso projeto;
- Depois precisamos **inserir um método de relacionamento em algum dos models** que vai criar a relação;
- Após o sync **uma coluna** que faz a relação entre as tabelas será criada;
- Que representa a **FOREIGN KEY**;

Adicionando dado relacionado

- Para adicionar o dado relacionado **o fluxo é quase o mesmo**;
- O grande detalhe é que precisamos passar o **id do item** que o relaciona;
- Podemos fazer com um input do tipo **hidden**;
- Dentro do **form** que contém os dados do novo item;
- E enviando para uma **nova rota** no nosso sistema;
- Vamos ver na prática!

Resgatando dados relacionados

- Precisamos definir as **relações entre os dois Models**, podemos fazer isso no Model de endereços;
- Depois basta utilizar o **operador include** com o nome do Model, onde estamos resgatando o dado;
- Isso faz com que os **registros associados** também venham na seleção;
- Como há dados relacionados, precisamos **remover o raw**;
- Vamos ver na prática!

Removendo relacionados

- Para remover itens relacionados utilizaremos o **mesmo processo de remoção de itens**;
- Criaremos um **formulário** que envia o id do item;
- E uma rota para receber estas informações e executar a remoção, utilizando o **método destroy**;
- Vamos ver na prática!

Sequelize

Conclusão da sessão

MVC

Introdução da sessão

O que é MVC?

- Um acrônimo de **Model View Controller**;
- É um padrão de **arquitetura de software**, que pode deixar nossas aplicações mais organizadas;
- A aplicação é dividida em camadas, cada sua **responsabilidade**;
- Teremos uma **nova estrutura de arquivos e pastas**;
- E a aplicação obedece um novo fluxo, que se repete para todas as ações;
- Aplicações em MVC tendem a ter uma manutenção/evolução mais fácil;

Camada do Modelo (Model)

- É uma camada onde vamos **interagir com o banco de dados**;
- Normalmente interage com os **arquivos do Controller**;
- Responsável por **resgatar, atualizar, remover e criar** dados;
- É comum que **cada tabela seja um Model**, assim como fazemos com o setup do Sequelize;
- Os Models são quem **controlam a arquitetura do sistema**, é fácil entender a regra de negócios analisando eles;

Camada de visualização (View)

- É onde **apresentamos os dados** que estão no banco;
- Geralmente a view **interage com o Controller**, que é o meio de campo;
- E também nas views temos a **interação com o usuário**, como formulários para inserir dados no sistema;
- É correto não haver **lógica/regra de negócios** na view, ou o mínimo possível;
- Normalmente a exibição é feita **através do HTML**;

Camada de controle (Controller)

- É onde temos a **interação entre Model e View**;
- Podemos definir qual view será impressa, processar dados que foram enviados para o banco ou para a view;
- Os Controllers terão um **código parecido com os das rotas**, que estamos criando até então no curso;

Nossa estrutura com MVC

- Nossa estrutura será composta por:
- **controllers**: pasta que ficam os arquivos de Controller;
- **models**: pasta que ficam os arquivos de Model;
- **views**: pasta que ficam os arquivos de View;
- **routes**: pasta que ficam os arquivos de rotas;
- **index.js**: arquivo que inicializa a aplicação;
- Vamos criar nosso app!

Criando o Model

- Vamos criar nosso Model dentro da pasta **models**;
- Precisamos também **sincronizar para criar a tabela**, então vamos importar no index.js;
- Este é o **primeiro passo do nosso MVC**, declarando uma entidade que faz parte da regra de negócios;
- Vamos ver na prática!

Criando o Controller

- Vamos agora criar nosso **Controller**, que ficará dentro da pasta controllers;
- Será uma classe que contém as **funções com a lógica de cada rota**;
- Algumas só encaminharão as views, outras vão processar dados e passar para os Models;
- Por isso **vamos importar o Model** que o controller utiliza;
- Vamos ver na prática!

Criando a View

- Agora que as **ligações foram feitas**, podemos criar views;
- Vamos adicionar sempre uma **função no controller**, que retorna a view;
- E também **linkar esta função com alguma rota**, em routes;
- Assim nós estaremos aplicando o **fluxo MVC**;
- Vamos ver na prática!

Criando as Rotas

- Criaremos as rotas com ajuda do **router**;
- Onde cada arquivo será responsável por um controller;
- E em cada rota vamos utilizar uma das **funções do Controller**;
- Por isso vamos **importar o controller responsável pela lógica** das rotas;
- O router precisa ser importado no **index da aplicação**;
- Vamos ver na prática!

Dando uma cara ao projeto

- Vamos aproveitar este início de estrutura e **criar um projeto completo em MVC**;
- Primeiramente vamos dar alguns **estilos iniciais**;
- E depois voltamos as outras partes da aplicação para **avançar no nosso projeto**;
- Vamos lá!

Salvando dados

- Agora é a hora de **interagir com o banco**;
- Ou seja, criar um vínculo entre um **Controller e um Model**;
- Criaremos uma **nova função para tratar os dados** e enviaremos para o banco;
- Como o **Sequelize** tem alguns métodos prontos, o trabalho do nosso Model fica mais simples;
- Vamos ver na prática!

Resgatando dados

- Para resgatar dados o **processo é semelhante**;
- Vamos criar uma **rota get** que acessar uma função do Controller;
- Nesta função traremos os dados através do **Model**;
- E assim podemos **imprimir no HTML**;
- Vamos ver na prática!

Removendo dado

- Para remover um dado vamos utilizar o **método destroy** do Sequelize;
- Condicionado por um **POST** que vai até a rota determinada pelo router;
- Ativando uma **função do Controller**;
- E depois **direcionamos para a home** novamente;
- Vamos ver na prática!

Editando dado

- O primeiro passo da edição nós já sabemos: **resgatar dado e colocar no formulário em uma nova view**!
- Para isso vamos precisar de uma **função no Controller**, pegando os dados via **Model**;
- E então criar a **rota que corresponde a um id de uma Task**, neste caso;
- **Preenchendo os valores dos inputs** com o que foi resgatado do banco;

Enviando dados para editar

- Agora no passo final, precisamos **enviar os dados**;
- Vamos criar uma **nova função no Controller** que processa o que veio do form;
- Esta mesma função **chama o Model** e executa a função de update;
- Retornando assim o usuário para onde ele possa ver a **Task atualizada**;
- Vamos ver na prática!

Completando tarefas

- Realizaremos uma ação do nosso sistema para **completar ou “descompletar”** as tarefas;
- Basicamente vamos **criar uma rota de update** que só atualiza o atributo done de Task;
- Precisamos de um **form na lista** para fazer o envio e também criar a rota;
- Vamos lá!

MVC

Conclusão da sessão

Node.js com MongoDB

Introdução da sessão

O que é NoSQL?

- São bancos de dados focados em **documentos (documents)**;
- A modelagem de dados com relacionamentos é **opcional**;
- **Não utilizamos queries** e sim métodos de classes para trabalhar com os dados;
- As tabelas não existem, temos as **collections**;
- **Não precisamos definir a estrutura** da collection previamente;
- **MongoDB** é um banco NoSQL;

O que é MongoDB?

- Um banco de dados **orientado a documento**;
- Os dados ficam salvos em **estruturas parecidas com JSON**;
- Frequentemente utilizado com **Node.js**;
- Diferente do SQL temos **índices primários e secundários**;
- Utilizaremos um **driver** para conectar a nossa aplicação com o MongoDB, semelhante ao MySQL e Sequelize;

Conectando ao MongoDB

- Primeiramente vamos precisar instalar o **driver do MongoDB**, que é **mongodb** (um pacote de npm);
- Depois criaremos a conexão, baseada em uma URL com o **protocolo mongodb://**
- Através da classe **MongoClient**
- Vamos ver na prática!

Criando MVC

- Antes de trabalhar com MongoDB vamos criar uma **estrutura MVC**;
- Isso vai **fortalecer nossos conhecimentos** com a arquitetura;
- E também poderemos comparar o trabalho com **SQL x NoSQL**;
- Vamos lá!

Inserindo dados

- Primeiramente vamos **criar um Model**, onde este será uma classe de JavaScript, para seguir o MVC;
- Depois utilizaremos o **Model** para criar o **método save**, que executa o **insertOne** de MongoDB;
- Por fim é necessário **criar o formulário** que interage com a rota do sistema;
- Vamos ver na prática!

Resgatando dados

- Para resgatar os dados de uma collection vamos utilizar o **método find** de MongoDB;
- Os dados vem em um cursor, para **converter em array** utilizamos **toArray**;
- Depois é só passar os dados para o controller e **exibir na view**;
- Vamos ver na prática!

Resgatando um dado

- Para resgatar um dado vamos utilizar o método **findOne**;
- Onde podemos **filtrar por um campo**, que no nosso caso vai ser o _id;
- Vamos **enviar o dado para o Controller** e depois para a View;
- Vamos ver na prática!

Excluindo dados

- Para remover um dado do banco, vamos utilizar o método **deleteOne**;
- Que **recebe um filtro** como no de resgatar dados, utilizaremos o campo **_id**;
- Basta executar o método no Controller e **redirecionar após a remoção**;
- Vamos ver na prática!

Editar dado - formulário

- Para editar um dado, primeiramente vamos criar uma view que mostra o **formulário preenchido**;
- A ideia é parecida com a de **getProduct** do nosso projeto;
- Vamos utilizar o **atributo value do input** para preencher o campo com o valor salvo;
- Vamos ver na prática!

Editar dado - salvando

- Para salvar um dado editado será necessário criar um método no **Model**;
- Que utiliza o método **updateOne** do MongoDB;
- Onde passamos o **id e também os dados** para atualizar;
- Criaremos também uma função no Controller e a rota;
- Vamos ver na prática!

Node.js com MongoDB

Conclusão da sessão

Mongoose

Introdução da sessão

O que é ODM?

- Um acrônimo para **Object Data Mapping**, ou seja, mapeamento de dados por objetos;
- Em **MongoDB** utilizamos ODMs para deixar mais ágil o processo de trabalho com os dados;
- Basicamente um **Model** ficará responsável pelas interação com um BD;
- A ODM mais utilizada para MongoDB é a **Mongoose**;
- **ORM e ODM** tem funções e utilizações muito semelhantes;

O que é Mongoose?

- Mongoose é a **ODM** mais utilizada para MongoDB;
- Onde declaramos um **Model**, que faz as interação com a **collection** que ele corresponde;
- Definimos nesse Model os **dados e seus tipos**, como estamos esperando que a collection fique;
- Isso se assemelha ao SQL, e ajuda na **organização/manutenção**;
- Como nas ORMs temos **métodos prontos** para a interação com o DB;

Instalando o Mongoose

- Nesta sessão vamos **adaptar o projeto** de MongoDb para utilizar o **Mongoose**;
- O primeiro passo é **instalar o pacote** e fazer a conexão com o banco;
- O nome do pacote é **mongoose**;
- Vamos precisar fazer algumas alterações no nosso **arquivo conn.js**, que é a ponte entre nossa aplicação e o DB;
- Vamos ver na prática!

Criando o Schema

- Agora vamos criar uma parte muito importante das aplicações que tem o Mongoose: o **Schema**;
- Que é um **esqueleto do nosso elemento** a ser inserido na **collection**;
- Vamos criá-lo no **Model** da aplicação;
- Vamos ver na prática!

Inserindo dados

- Para inserir dados vamos utilizar o **método save** de Mongoose;
- Este método é da própria biblioteca, ou seja, **não vamos utilizar o do nosso Model**, que usamos no MongoDB;
- O restante fica bem parecido, **passamos os dados a serem inseridos em um objeto** para save;
- Vamos ver na prática!

Resgatando dados

- Para resgatar os dados vamos utilizar o **método find** do Mongoose;
- É necessário o **método lean** também para formatar os dados, de forma que o Handlebars consiga utilizar;
- Apenas com estas alterações já estamos prontos;
- Vamos ver na prática!

Resgatando dado individual

- Para encontrar um único item vamos utilizar o **método findById**;
- Neste caso **não precisamos converter o id** para o padrão do MongoDB, o Mongoose faz isso para nós;
- E também utilizaremos o **lean** novamente;
- Vamos ver na prática!

Edição de dado - form

- Para preencher o formulário de edição vamos utilizar novamente o **findById**;
- O **método lean** nos ajuda a receber os dados para o Handlebars;
- E pronto, nosso código já se adapta aos outros processos;
- Vamos ver na prática!

Edição de dado - post

- **Obs:** para resolver um warning vamos colocar as rotas em /products;
- Depois basta utiliza o **método updateOne**, onde passamos um **filtro** para atingir o registro correto;
- E também um **objeto com os dados** de atualização;
- Vamos ver na prática!

Removendo dado

- Para remover um dado utilizaremos o **método deleteOne**;
- Que leva simplesmente o **filtro como argumento**, neste caso utilizaremos o **_id**;
- Depois basta redirecionar o usuário;
- Vamos ver na prática!

Mongoose

Conclusão da sessão

Iniciando com APIs

Introdução da sessão

O que é uma API?

- **API** é um acrônimo para Interface de Programação de Aplicações (Application Programming Interface);
- Uma forma simples de **comunicação entre aplicações**;
- Não necessita de telas, respostas geralmente são em **JSON**;
- **Independente** do front-end, a API não possui ligação com o front;
- Baseadas em **requisição e resposta**;
- Podemos criar uma **API com Express**!

O que é REST e RESTful?

- **REST** significa Representational State Transfer ou Transferência Representacional de Estado;
- Um **estilo de arquitetura**, que define como as APIs devem ser criadas;
- Seguindo todos os padrões a API é considerada **RESTful**;
- Alguns dos pontos são: respostas uniformes, sem ligação de estado, cache de respostas e outros;
- Ao longo do curso focaremos em **seguir estes padrões**!

O que é uma SPA?

- **SPA** significa Single Page Application;
- É quando a aplicação possui um **front-end separado do back-end**;
- Ou seja, uma **API** para o back-end;
- E um framework **front-end JS**, como: React, Vue ou Angular;
- Aplicações deste tipo estão **dominando o mercado**;
- E as sessões seguintes tem como objetivo trabalhar **focadas em SPA**;
- A arquitetura anteriormente utilizada é conhecida como **Monolith**;

Verbos HTTP

- Os **verbos HTTP** andam de mãos dadas com as APIs, e **fazem parte do REST**;
- Temos métodos como: **GET**, **POST**, **PUT**, **DELETE**, **PATCH** e etc;
- Cada um **representa uma ação na API**, ou seja, são essenciais;
- É muito importante que as nossas funcionalidades usem os **métodos corretos**;
- Enviamos o método **através da requisição**;

Criando uma API com Express

- Para criar a **API** teremos uma tarefa bem simples, comparada a estrutura com Handlebars;
- Basta instalar o **Express**, ele fará tudo sozinho neste ponto;
- Depois **criaremos uma rota que responde em JSON**, este é o dado de comunicação entre aplicação e API;
- É importante definir o **verbo correto**, como **GET** ou **POST**, por ex;
- Vamos ver na prática!

Conhecendo o Postman

- O **Postman** é um client para testes de API;
- Podemos então **criar o back-end antes ou separado do front** só com a ajuda deste software;
- Como você deve imaginar, é **fundamental** quando estamos criando APIs;
- Podemos **simular verbos**, **corpo de requisição**, inserir **headers**, tudo que é possível com uma aplicação web;
- Vamos instalar!

Testando rota com Postman

- Para acessar uma rota com o Postman precisamos **configurar o client**;
- Devemos **inserir o verbo correto** para a rota;
- E também **configurar o endpoint**, que é a **URL** onde nossa rota foi estabelecida;
- Enviando a requisição, **receberemos a resposta**;
- Vamos ver na prática!

Criando uma rota de POST

- Para criar a rota de POST vamos utilizar o **método post** do Express;
- Podemos extrair os dados da requisição, acessando **req.body**;
- Da mesma maneira que no get, podemos **retornar uma resposta como JSON** pela API;
- Vamos ver na prática!

Rota com POST na API

- Para acessar a rota de POST precisamos **mudar o verbo e a URL do endpoint** da API;
- Depois precisamos **alterar o corpo da requisição** com os dados solicitados pela API;
- E então enviar o request;
- Vamos ver na prática!

Adicionando status na resposta

- Os status podem **ajudar no desenvolvimento da nossa aplicação**;
- **Exibindo mensagens de sucesso ou erro**, por exemplo;
- Precisamos entre res e o método json definir um **número de status HTTP**;
- Isso é feito por meio do **método status**;
- Vamos ver na prática!

A realidade sobre as APIs...

- A API é **desenvolvida de forma muito semelhante** a quando criamos projetos com Handlebars, por exemplo;
- Temos algumas alterações leves, como: **resposta apenas por JSON**;
- Mas isso acaba até **simplificando as coisas**, separando as responsabilidades;
- Ou seja, **reaproveitamos** todo o conhecimento visto durante o curso;

Iniciando com APIs

Conclusão da sessão

Conclusão

Fechamento e próximos passos

API REST com Node.js e MongoDB

- A API será baseada no framework **Express** para tratar as requisições e enviar as respostas aos endpoints;
- Vamos utilizar o **MongoDB** como banco, porém com a ODM **Mongoose**;
- Realizaremos as operações no **MongoDB Atlas**;
- O pacote **Nodemon** vai ajudar a atualizar o código em tempo real da aplicação;
- A API será testada pelo software **Postman**;

O que vamos precisar?

- **Node.js instalado na máquina**, para rodar o projeto e poder utilizar o **npm** para gerenciar os pacotes da aplicação;
- Editor de código (sugestão: **VS Code**);
- **Postman**, para teste de API;
- Uma **conta no MongoDB Atlas** ou MongoDB instalado na máquina;

O nosso objetivo

- **Criar uma API** nos padrões RESTful, que terá um **CRUD**;
- Os endpoints serão criados com os **verbos HTTP** que correspondem a ação do mesmo;
- As respostas serão baseadas em **JSON**, retornando também o **status** correto;
- Aplicaremos **validações** simples, para simular o ‘mundo real’;