**Seção 5: Introdução as Rotas, Controllers e Views**

**Aula 23**

Para inicar a execução do Laravel, deve-se executar php -S localhost:\*porta disponível\* no console e deixa-lo executando para ser possível a sua utilização.

**Aula 24**

Mostrado o artisan, script responsavel por garantir execução de comandos uteis de maneira mais fácil, como por exemplo, basta executar o comando php artisan serve para rodar a aplicação http do laravel instalado com a porta padrão 8000, podendo ser alterada com o parâmetro --port=\*porta disponível\*.

**Aula 25**

Introução as rotas, que no ambiente laravel são segmentadas em 4 partes: API, channels, console e web. Cada uma possui sua própria função dentro do sistema a ser desenvolvido e serão utilizadas de forma mais prática futuramente.

**Aula 26**

Utilizada a rota Web para criar duas novas rotas, a de /sobre-nos e a /contato. É utlizado o método Route::\*método http\*($uri, $callback). O método http pode ser get, post, patch, put, delete, etc. Já a $uri é o local que deseja ser mapeado como no caso /sobre-nos e /contato. O $callback se trata do que será realizado quando a local for acessado, geralemente sendo uma função que retorna uma view.

**Aula 27**

Explicação breve sobre controladores. Se tratam da parte do código responsável pela lógica do negócio, após ser tatados pelas rotas que levarão até determinada página do site.

**Aula 28**

Adicionado os controllers da aplicação. Com o artisan, é possível criar um novo controller com o comando php artisan make:controller, que irá criar um novo arquivo no diretório /app/http/controllers/. Após a criação dos controllers, no primeiro momento foram criadas uma função por controller somente exibindo a mesma mensagem que já estava sendo trasmitida em forma de callback nas rotas. Agora nas rotas deve ser passada uma string, que será interpretada de forma a esperar o nome da controller e uma ação a ser realizada na chamada desse controller, isso no laraval 7 para trás. A nova utilização deve-se ser utilizado como [\*diretório\*::class, ‘\*funcão\*’].

**Aula 29**

Explicação sobre view. Se trata da visão produzida no lado do servidor responsável por exibir os elementos para o usuário. Também é dito que este modelo se trata do modelo tradicional de criação de sites, existindo um modelo mais moderno, tratando back e front-end com softwares diferentes, unidos por uma api.

**Aula 30**

Feita a criação das views do projeto. Estão presentes dentro da pasta \app\resources\views, em que é criada a pasta site que irá conter as 3 páginas criadas até o momento. Essas views irão conter o que será de fato exibido para o usuário.

**Seção 6: Avançando com Rotas (Routes)**

**Aula 31**

Mostrado como receber parâmetros em rotas. Deve-se criar uma nova rota que pode ser o mesmo endereço, porém com mais uma separação com / e entre {} é passado o parâmetro que com uma função de callcabk poderá ser utilizado como uma variável.

**Aula 32**

Exibido como fazer com que os parâmetros sejam opcionais e não obrigatórios. Para realizar isso, basta definir um safe null operator no final do parâmetro presente na rota e na função de callback, junto da declaração da variável deve ser definido um valor, que será tratado com o valor padrão, caso nenhum seja passado no parâmetro. O laravel possui a limitação de que o parâmetro opcional só pode estar em falta da direita para a esquerda, pois caso alguma parâmetro não seja passado, porém e sequente seja passado, a página não será encontrada.

**Aula 33**

Feita a aplicação de expressões regulares para tratar os parâmetros passados. Para fazer o tratamento dos parâmetros, ao final do método get do route, deve-se por a expressão ->where(), onde serão passados o nome do parâmetro a ser tratado e depois separado por vírgula as expressões regulares desejadas, ambos sendo contidos em strings.

**Aula 34**

Criada uma lista com os links das rotas criadas até o momento.

**Aula 35**

Criadas 4 novas rotas para login, clientes, produtos e fornecedores.

**Aula 36**

Realizado o agrupamento das rotas presentes na aula passada utilizando o método prefix do Route. Route::prefix(‘/app’)->group, esse método recebe uma função que deverá possuir as rotas que serão agrupadas por esse prefixo.

**Aula 37**

Atribuídos nomes as rotas definidas até o momento, que facilitam e muito em suas chamadas em links por exemplo, fazendo com que a chamada seja a mesma independente do diretório da aplicação, ou caso a rota em si seja alterada. Para definir um nome, basta ultilizar o método name->(‘\*nome do site\*’) no final da linha da rota que se deseja nomear.

**Aula 38**

Mostrado como realizar redirecionamento de rotas. Para isso, outras duas rotas foram criadas somente para teste. Existem duas maneiras de realizar o redirecionamento. Uma delas é por meio da função redirect()->route(‘\*rota destino\*’), que deve ser passado dentro da função de callback da rota original. Outra maneira de redirecionar uma rota é utilizando o método redirect de Route (Route::redirect(‘/rota2’, ‘/rota1’);).

**Aula 39**

Criada uma rota de fallback, que é acessada toda vez que a rota não é encontrada. Sua sintaxe é Route::fallback(function() {\*ação da rota fallback\*});

**Seção 7: Avançando com Controladores (Controllers e Visualizações (Views)**

**Aula 40**

Mostrado como encaminhar parâmetros da rrota para o controlador. Basta que no momento em que seja passado o controller responsável por aquela rota, deve-se receber os parâmetros na função do controlador. Exem: parâmetros /{p1}/{p2}, a função do controller deve possuir function controller($p1, $p2) {} (o nome não precisa ser o mesmo, o importante é a ordem dos elementos a serem recebidos, da esquerda para a direita.

**Aula 41**

Mostrado como passar as variáveis de um controller para uma view. Há 3 maneiras de fazer isso. Após criar o retorno da view e indicar qual view será atribuída, pode-se criar um array contendo uma string com o nome desejado para a vaariável na view recebendo(=>) a variável presente no controller, para adicionar outras variáveis basta criar outros índices no array, o nome dessa técnica é array associativo. Existe também a função nativa do php compact, em que basta apenas usar o método compact dentro do método view, passando como parâmetros do primeiro strings sem o $ com o nome da variável, fazendo com que uma variável de mesmo nome seja passada diretamente na view. A ultima maneira é utilizando o método laravel ->with(‘\*nome desejado\*’, $\*variavel\*), contendo somente uma variável por utilização. Para passar outras variáveis, basta repetir a funcão novamente após o termino da anterior.

**Aula 42**

Explicado o blade. Se trata de uma tecnologia de renderização de views, possibilitando uma escrita mais enxuta do php no front-end. Além disso, o blade ainda permite que a escrita habitual do php seja utilizada, apesar de não ser muito usual.

**Aula 43**

Explicado como incluir comentários com blade e blocos de php puros. Para comentar em blade basta utilizar {{-- comentário --}}. E para abrir blocos de php, basta utilizar @php e @endphp para recerrar o bloco. {{ \*variavel ou string\* }} é sinônimo de <?= \*variável ou string\* ?>.

**Aula 44**

Instalado extensão de highlight da sintaxe blade no VSCode.

**Aula 45**

Feito if/else no blade. Basata utilizar @if, @else, @elseif e @endif para realizar o if. Diferente do php nativo, não é necessário abrir ou fechar tags para a programação, facilitando o trabalho.

**Aula 46**

Utilizado sintaxe blade @unless. Se trata da inversão do if, comparando se a informação passada é falsa, mesmo procedimento do if(!).

**Aula 47**

Explicada utilização do @isset. Essa sintaxe poupa a necessiade de abrir um if somente para verificar a existencia de uma variável. Para fechar, basta usar @endisset.

**Aula 48**

Definição do @empty, que verifica caso o valor atribuído a variável não é vazio. Valores vazios: ‘’, 0, 0.0, ‘0’, null, false, array(), $var.

**Aula 49**

Explicação de operador condicional ternário, no php puro mesmo.

**Aula 50**

Explicação do operador condicional de valor default no blade. Trata-se da utilização de ?? após a impressão de uma variável no blade, seguido pela definição do valor default. O operador condicional verifica como um isset e não como empty, então somente de estar setada, a variável não cairá no default.

**Aula 51**

Mostrado switch case no blade. @switch para abrí-lo, @case para inciar uma condição, @break para encerrá-la e @endswitch para finalizar o switch.

**Aula 52**

Mostrado for no blade. @for para iniciá-lo e @endfor para finalizá-lo. É possível usar um índice de um array como $i diretamente, exem: @for($i = 0; $array[$i]; $i++).

**Aula 53**

Mostrado while no blade. @while para iniciar e @endwhile para finalizar. É necessário criar uma variável de contador para contabilizar a execução do while, possívelmente sendo necessário a abertura da tag @php para criar o contador.

**Aula 54**

Exibido foreach no blade. @foreach para inicicar e @endforeach para finalizar.

**Aula 55**

Mostrado o forelse, tag exclusiva da sintaxe blade. Funciona como um foreach, mas caso o array percorrido esteja vazio, com o forelse é possível definir um @empty para definir o que será realizado com esse array vazio. Para fechá-lo, basta usar @endforelse.

**Aula 56**

Para escapar a tag de impressão do blade({{ }}), basta por um @ na frente da tag, o que fará com que o elemento inteiro seja imprimido e não interpretado.

**Aula 57**

Ao utilizar um loop for ou foreach, existe uma variável que contém informações sobre a execução, para acessá-la deve-se usar {{ $loop->\*informação\*}}. Alguns exemplos são ->iteration, ->first, ->last, ->count.

**Aula 58**

Disponibilizados novos arquivos com elementos feitos das páginas do diretório site.

**Aula 59**

Explicados o que são assets, basicamente sendo elementos que ajudam a composição de um elemento html, sejam css, arquivos javascript, etc.

**Aula 60**

Baixados e incluidos imagens presentes nos arquivos da aula anterior. Mostrado a tag {{ asset(‘\*diretorio\*’) }} que realiza a inserção de assets. Sua vantagem é que o diretório que a função asset irá acessar pode ser alterado a qualquer momento, então caso seja necessário uma mudançã de diretórios, configurando em apenas um local irá mudar em todos os outros automáticamente.

**Aula 61**

Feita a inclusão do css contido na página em um arquivo externo, dentro da pasta public.

**Aula 62**

Criado um template da parte de fora da body dos sites e colocada em outra view. A view contendo a head do site pode ser acessada pelos outros por meio do @extends e as views com o conteúdo podem ser renderizadas na view head por meio do @section(‘\*nome do conteudo\*’) e depois finalizada com @endsection. Depois, no site com a head foi usado o @yield(‘\*nome do conteudo\*’), fazendo com que o conteúdo das outras páginas sejam gerados sem repetição e uma alteração no template da head irá alterar em todas as páginas.

**Aula 63**

Feito uma alteração no título por meio da section, que pode ser mandada sem o @endsection, passando somente uma informação. @section(‘\*variavel\*’, ‘\*conteudo\*’) irá passar a variável que pode ser vista com o @yield.

**Aula 64**

Feita a inclusão do menu superior das páginas no arquivo basico, esndo criada a parte \_partials, que deverá conter partes do html que podem ser reutilizadas, se tratando de elementos visuais e não somente da head. @include irá incluir tudo que estiver no determinado arquivo, no local em que for chamada a sintaxe.

**Aula 65**

Passada a ação e o método do formulário presente na página de contato. A ação foi passada usando {{ route(‘site.contato’) }}, pois o objetivo é enviar o formulário para seu próprio controller.

**Aula 66**

Modificado o formulário fazendo com que ele seja enviado por post. Para fazer isso, é necessário criar uma nova rota utilizando o método post. Todos parâmetros da rota podem ser iguais, porém usando o método get ou post, somente um ou outro serão recebidos, fazendo que nesse caso os dois sejam necessários. Também é necessário adicionar ao formulário uma linha contendo @csrf, que se trata de um token necessário para que o laravel aceite qualquer formulário enviado, por questões de segurança.

**Aula 67**

Explicada a importância do token csrf. O mesmo impede que requisições maliciosas sejam realizadas com informações armazenadas nos cookies de um site para o envio de formulários falsos, como o de envio de dinheiro em um internet banking, por exemplo. O token csrf é mandado pelo servidor para a página web autêntica do formulário e é enviada novamente com os dados preenchidos do formulário. Caso qualquer formulário que for enviado para o servidor não possuir o token, será imediatamente recusado.

**Aula 68**

Mostrado a sintaxe @component para retirar o formulário que estava se repetindo em duas páginas para que o mesmo seja utilizado de forma dinâmica. Foi criada uma nova pasta no diretório layout, nomeada components. Nela foi criada um arquivo que contém o formulário.

**Aula 69**

É possível passar parâmetros por meio do @component, que será recebido pela variável @slot diretamente no que estiver escrito entre os @components. Com o @component é possível passar uma ou mais variáveis que serão passadas e serão utilizáveis diretamente no componente. Exem: @component(‘site.teste’, ‘\*variavel\*’ => ‘\*valor\*’).

**Seção 8: Models, Migrations, Seeders, Factories, Banco de Dados, Tinker e Eloquent ORM**

**Aula 70**

Criado a Model SiteContato com o comando php artisan make:model SiteContato -m. O comando -m cria junto da model, uma migration, que será explicada mais a frente no curso.

**Aula 71**

Utilização e explicação da migration. Se trata de um arquivo que possui informmações escritas em php, que irão realizar a criação de elementos no banco de dados, facilitando assim por exemplo, o desenvolvimento em equipe, pois no código pode conter as informações necessárias para o criação de tabelas e elementos em um banco de dados. Acessar <https://laravel.com/docs/10.x/migrations> para a documentação com os comandos das migrations.

**Aula 73**

Mostrado a execução das migrations. Para realizar, primeiro é necessário conferir o arquivo database.php, que contem informações sobre vários tipos de bancos de dados diferentes. Neste arquivo, existe a utilização do método env em várias linhas. Esse método se refere as infromações do environment, que deve ser deifinido no arquivo .env, na raiz do projeto. Neste arquivo, é necessário conferir se o banco setado é o que deseja ser utilizado. Também é interessante criar um arquivo de mesmo nome do que está escrito no elemento database\_path(método esse que irá procurara no diretório database), no arquivo database.php, no caso do sqlite. Para que a leitura do database\_path aconteça, é necessário excluir a linha com DB\_DATABASE no arquivo .env.

**Aula 74**

Instalação do mysql server.

**Aula 78**

Mudada a migration para os paramêtros mysql, presente no arquivo .env.

**Aula 79**

Executada a migration.

**Aula 80**

No caso da criação de uma model sem o -m no final, fará com que uma migration não seja criada automáticamente, porém, para criar depois caso necessário, basta digitar o comando php artisan make:migration \*nome\_da\_migration\* (no caso create\_fornecedores\_table). Criada a model Fornecedor e sua migration com a coluna string nome.

**Aula 81**

Criada uma migration de alteração de tabela (alter\_fornecedores\_novas\_colunas), informando as novas colunas a serem informadas para a mesma tabela utilizada na migration anterior, fazendo com que caso novas colunas sejam adicionada sem que o conteúdo anterior das tabelas seja perdido, criando outro documento de migrate poderá resolver o problema. No caso, o objeto Schema chama a função table, que é responsável por selecionar uma tabela e não criá-la.

**Aula 82**

Explicação da diferença do método up e down das migrations. O método up é responsável por realizar ações no banco de dados, já o método down é responsável por desfazer as ações do método up. Para executar o método down de uma migration, é necessário usar o comando php artisan migrate:rollback. Esse comando executa a uma rollback(down) da mais recete para a mais antiga, então caso existam 6 batches de migrations, somente a sexta será revertida. É possível alterar qual batch será revertida adicionando --step=\*batch desejada\* ao comando migrate:rollback.

**Aula 83**

Descrição de como criar colunas que aceitam null e valores default para colunas nas migrations. Para null, basta adicionar o método ->nullable() ao final da criação de uma coluna. Para valores default, basta adicionar o método ->default(\*valor default\*), utilizando ‘’ para strings.

**Aula 84**

Criadas chaves estrangeiras por meio de migrations. Para realizar tal, foi criada uma nova migration da tabela produto\_detalhes, na qual irá receber o id da tabela produtos. É criada uma constraint $table->foreign(‘produto\_id’)->references(‘id’)->on(‘produtos’). Essa linha cria uma chave estrangeira produto\_id, ligada a coluna id da tabela produtos, criando assim a ligação. Também é utilizado $table->unique(‘produto\_id’), para que os registros das tabelas sejam únicos, já que o objetivo é uma relação um para um.

**Aula 85**

Criados relacionamentos um para vários com migrations. A diferença em usa implementação é ma criação é pela ausência do método unique. Também é interessante criar a coluna foreign como unsinegnedBigInteger, pois essa é a criação padrão de uma id, sendo útil definir da mesma forma para uma foreign key.

**Aula 86**

Criado um relacionamento de vários para vários. Esta ligação é feita com base na criação de uma tabela auxiliar, totalizando 3 tabelas em que as duas tabelas exteriores fazem uma ligação de um para vários, dessa forma totalizando uma ligação de vários para vários. É criada uma nova migration criando as duas tabelas, a tabela filial que irá ter uma relação de vários para vários com a tabela produtos, para isso é criada a tabela auxiliar produtos\_filiais. É então criada na última duas foreign keys, uma de cada tabela, estabalecendo assim o relacionamento desejado.

**Aula 87**

Mostrado o modificador after. Sua função é definir o local em que uma coluna deve ficar, após sua criação, quando é feita em uma tabela pré-existente. Sua sintaxe é ->after(‘\*coluna anterior desejada\*’).

**Aula 88**

Mostrado 4 comandos uteis com o migrate: status; reset; refresh e fresh. O comando status exibe todos as migrates e seus batches, além de exibir caso elas foram migradas ou não. O comando reset executa os comandos down de todas as migrates, mas nada além disso. Já o refresh executa todos os comandos down das migrates e em seguida executa os comandos up, refazendo o banco de dados. O fresh é bem parecido com o refresh, porém ao invéz de executar o comando down, ele dropa todas as tabelas e executa o comando up em seguida.

**Aula 89**

Introdução ao eloquent ORM, que ajudará a trazer o paradigma de orientação a objetos para os bancos de dados relacionais.

**Aula 90**

Introdução ao tinker. Auxilia no teste das classes criadas e models por exemplo, funcionando como um atalho, pois não será necessário criar uma interface web para visualizar o funcionamento das classes.

**Aula 91**

Mostrado como realizar inserções com o tinker. Após abrir o terminal do tinker (php artisan tinker), foi criado um novo objeto da classe \App\SiteContato() e foram passados os respectivos parâmetros para o objeto. Depois para inserir o objeto no banco de dados, basta usar o modificador ->save(), que o objeto será inserido normalmente no banco de dados.

**Aula 92**

Mostrado que o eloquent tem um padrão para a tradução do nome da model para o nome do banco de dados. Primeiro, as letras maiusculas terão um undeline adicionados a sua esquerda (SiteFornecedor -> Site\_Fornecedor). Depois, as letras maíusculas serão transformadas para minusculas (Site\_Fornecedor -> site\_fornecedor). Por último, algo que pode apresentar problemas. É colocado um s no final da última letra, para dar o sentido de plural (site\_fornecedor -> site\_fornecedors). Porém em alguns casos como o do exemplo, somente o s não completa o sentido da palavra, provavelmente diferenciando do banco que está criado. Para isso é possível definir o nome da tabela alvo do eloquent, usando protected $table = ‘\*nome da tabela\*’. A variável protegida $table é definida pelo padrão de tradução do eloquent automáticamente.

**Aula 93**

No tinker, também é possível inserir registros por métodos estáticos, sem depender da instanciação de um objeto. Para isto, deve-se usar \*model\*::create([\*array de parâmetros a serem inseridos\*]);. Dessa forma, um novo registro será inserido no banco referente a model colocada. Porém, esse comando inicialmente apresentará erro. Isso é por que na model é necessário que exista a definição de quais colunas podem ser preenchidas com o comando protected fillable = [\*colunas preenchiveis\*]. Após essa adição para a model, o método estático create irá funcionar corretamente.

**Aula 94**

Mostrado o método estático ::all();. Esse método recupera todos os elementos cadastrados em um banco de dados, facilitando a sua visualização. No tinker, é possível declarar use \*namespace completo da model\* e nas próximas utilizações da model, não é necessário o namespace completo, somente o nome da model.

**Aula 95**

Utilizado método estático ::find();. Por padrão este método pesquisa em um banco de dados pela id passada do registro. Caso o id passado não seja encontrado, simplesmente não haverá retorno. É possível passar um array de ids, que retornará uma coleção de registros, assim como utilizando o método all().

**Aula 96**

Introduzido o método estático ::where(). Com ele é possível fazer pesquisas simples e avançadas sem a necessidade de pesquisar no banco. Após declarar a model a ser pesquisada, são necessários 3 parâmetros para a utilização do where: ::where(‘\*nome da coluna\*’ , ‘\*operador de comparação(<, >, like)\*’ , ‘\*valor a ser comparado\*’). Quando o operador de comparação for =, é possível omitir o segundo parâmetro, contendo o parâmetro do nome da coluna e do valor a ser comparado. Será sempre retornado um conjunto de registros, mesmo resultando em um ou nenhum. Além disso, somente executando o método estático, será retornado um builder, sem o resultado. Para isso, é necessário a utilização do modificador ->get() ao final do comando.

**Aula 97**

Mostrados os métodos estáticos ::WhereIn() e ::WhereNotIn(). Ambos recebem dois parâmetros, o primeiro sendo o nome da coluna a ser buscada e o segundo, o(s) valor(es) a ser(em) buscado(s). Em caso de mais de um, pode ser utilizado um array, sendo passado os valores. No caso do whereIn, serão retornados os registros que atenderem os valores do segundo parâmetro, já o whereNotIn irpa retornar o oposto. Pode ser utilizado para datas, integers e strings.

**Aula 98**

Os métodos ::whereBetween() e ::whereNotBetween funcionam de maneira parecida com os da aula anterior, porém, além dos próprios valores a serem comparador por igualdade, também serão retornados os valores entre esses registros no caso do whereBetween. No caso do whereNotBetween o oposto será retornado.

**Aula 99**

Mostrado como utilizar mais de um where no tinker. Para realizar tal, basta que ao final do primeiro where, seja passado um próximo utilizando -> e em seguida o where desejado.

**Aula 100**

Quando é utilizado mais de um where em uma busa e espera-se que a operação entre os dois ou mais wheres seja de or e não and (operação padrão), pode-se inserir o or na frente da função de qualquer um dos wheres(orWhere, orWhereIn, orWhereBetween), fazendo com que a operação utilizada seja a de or.

**Aula 101**

Mostrado o método de pesquisa ::whereNull, que retorna os valores nulos, de acordo com a model e coluna selecionada. Logo, a função possui somente um parâmetro, que é o da coluna a ser pesquisada.

**Aula 102**

Exibido os comandos de pesquisa que envolvem datas. Eles são: whereDate(‘\*coluna\*’, ‘\*data\*’), whereDay(‘\*coluna\*’, ‘\*dia\*’), whereMonth(‘\*coluna\*’, ‘\*mês\*’), whereYear(‘\*coluna\*’, ‘\*ano\*’), whereTime(‘\*coluna\*’, ’\*operador de comparação\*’, ‘\*horário\*’). Lembrando que para que as pesquisas de datas funcionem, as colunas devem ser do tipo data e estarem preenchidas corretamente, pois o laravel não tem inteligência de interpretar horários diferentes do padrão.

**Aula 103**

Mostrado o comando whereColumn, que funciona de maneira muito parecida com o where, porém de o terceiro parâmetro ser um valor para a comparação, o whereColumn o compara com outra coluna do mesmo registro, podendo assim comparar se o horário criado da coluna é maior, igual ou menor que o horário de modificação, por exemplo.

**Aula 104**

Mostrado como fazer pesquisas mais específicas utilizando o eloquent. Existem casos em que a buscas podem possuir ‘camadas’: select \* from tabela where (... or ...) and (... and ...), retornando resultados diferentes caso não houvesse os parenteses. Para isso, é possível realizar consultas no eloquent utilizando o método where, passando uma função de callback que possui como parâmetro a prória query, sendo ela a utilizada para realizar as consultas de uma camada, por exemplo: $consulta = Model::where(function($query) { $query->whereIn(...)->orWhereDate(...);})->orWhere(function($query){ $query->whereBetween(...)->whereNull(...);});

**Aula 105**

Para organizar a ordem dos resultados no eloquent, deve ser usar o método ::orderBy(‘\*coluna\*’, ‘\*asc ou desc\*’). Por padrão o modo ascendente é selecionado e pode ser omitido caso for o desejado. É possível também stackar orderBy(), sendo que será ordenado de acordo com o primeiro e denstro dessa organização será pela segunda ordem. É possível utilizar filtros where também em conjunto ao orderBy.

**Aula 106**

Introdução as collections do laravel, sendo possível encontrar a documentação das mesmas no site do laravel.

**Aula 107**

Após realizar o get(), será retornado uma collection, que possui vários métodos, 3 deles são: ->first(), que irá retornar o primeiro elemento da collection; ->last(), que irá retornar o último e ->reverse que irá retornar a collection inteira porém invertida.

**Aula 108**

Mostrados os métodos ->toArray() e ->toJson() do eloquent. Fazem exatamente o que parece. Após utilizar esses dois métodos, não será possível usar os métodos de collections do laravel.

**Aula 109**

Mostrado o método ->pluck(‘\*coluna\*’, ‘\*coluna associativa\*’), que irá retornar somente a coluna inserida. É possível passar um segundo parâmetro que é o da coluna associativa, que assumirá como identificador para os elementos da coluna, exemplo: ->pluck(‘email’, ‘nome’) irá retornar uma collection com [‘santhiago’ => ‘santhiago@contato.com’].

**Aula 110**

Falado um pouco mais sobre a documentação laravel.

**Aula 111**

Mostrado o método save() no eloquent. É possível alterar registros no banco que já foram feitos. Para isso, basta recuperar o registro que se deseja modificar com o método ->find(), depois basta alterar os valores desejados, exemplo: $registro->nome = ‘novo nome’;. Após isso, basta utilizar o método save() que as alterações serão salvas no banco.

**Aula 112**

Usado o método ->fill(). Sua funcionalidade é modificar vários elementos de um registro de uma vez, por meio de um array, sua sintaxe é: $registro->fill([‘\*coluna1\*’ => ‘\*valor1\*’, ‘\*coluna2\*’ => ‘\*valor2\*’, ‘\*coluna3\*’ => ‘\*valor3\*’])->save(). Pronto, registro atualizado.

**Aula 113**

Utilizado o método ->update([‘\*coluna\*’ => ‘\*valor\*’]). Nele os valores passados serão substituídos de acordo com as colunas e os valores do array passsado. É possível também aplicar as alterações de acordo com um where que for utilizado, para várias mudanças ao mesmo tempo, por exemplo.

**Aula 114**

Mostrados os métodos delete() e destroy(). O método delete, serve para deletar um registro de acordo com as clausulas de where utilizadas para receber os registros. Já o método destroy(\*id1\*, \*id2\*, ...) é mais curto, porém serve somente para ids. Uma observação é que ao utilizar qualquer um dos dois métodos, caso seja uma variável que armazena os registros, será retornado um true ou false informando o sucesso ou não da operação, porém, caso esses métodos sejam utilizados em diretamente na model, será então retornado um int informando quantos elementos foram excluídos.

**Aula 115**

Mostrado o softDelete. Faz com que quando um registro seja deletado, na verdade ele seja desativado, tendo uma coluna no banco de dados informando a data da desativação, evitando a exclusão do registro. Para sua implementação, basta que a model possua outra herança, algo que normalmente não seria possível, mas o laravel contorna o problema com o use. Primeiro deve-se passar a classe do softDelete, que fica no diretório Illuminate\Database\Eloquent\SoftDelete, depois devese usar a sintaxe use SoftDelete na model desejada. Após isso deve-se criar outra coluna em uma migration de preferência, utilizando a sintaxe $table->softDeletes(); e $table->dropSoftDeletes() no método down. Pronto, quando algum registro for excluido na determinada tabela, somente será adicionado a data na coluna deleted\_at. Ainda é possível excluir um registro de verdade utilizando a função forceDelete(), com ela o registro será completamente removido da tabela.

**Aula 116**

Mostrado como selecionar e restaurar elementos excluídos com softDelete. Para visualizar os elementos excluídos, pode-se usar o método withTrashed(), que irá retornar tanto os registros excluidos quanto os não excluidos. Utilizando o onlyTrashed, será retornado somente os elementos excluidos. Para restaurá-los, é necessário utilizar um dos comando anteriores, pois somente eles capturam os elementos excluidos e armazenar a collection retornada em uma variável. Após isso, é necessário selecionar a index do registro desejado como se fosse um array, utilizando o método restore(), que irá colocar o valor nulo na coluna deleted\_at, restaurando o registro, exemplo: $collection[0]->restore();.

**Aula 117**

Mostrado os seeders. São classes responsaveis por popular tabelas a fim de testes. Para criar uma seeder, basta utilizar o comando artisan make:seeder \*nome do seeder\*. Após isso acessando a nova classe seeder criada, é necessário instanciar a model que se deseja popular com o método use. Então deve-se criar os registros para popular a tabela, são mostrado 3 modos diferentes. O primeiro é instanciando o objeto utilizando $variavel = new Model () e preenchendo o registro e utilizando o método save no final. O segundo foi feito utilizando o método estático create (Model::create([‘\*array com informações\*’]). O ultimo foi utilizando o método insert direto da classe DB (DB::table(‘\*tabela\*’)->insert([\*array com informações\*]). Também é necessário passar qual model será executada pelo artisan no arquivo DatabaseSeeder.php, informando o comando $this->call(Model::class). Após isso basta executar o comando php artisan db:seed e pronto, tabelas populadas.

**Aula 118**

Criado um novo seeder, porém dessa vez, um para a model SiteContato. Após a criação do seeder,

Para não executar novamente o criado na aula anterior, é possível passar diretamente o nome da seeder desejada, exemplo: php artisan db:seed --class=MinhaSeeder.

**Aula 119**

Criada uma factory, responsável por criar automaticamente quantos registros forem desejados. Para criar uma, basta utilizar php artisan make:factory \*nome da factory\* --model=\*model desejada\*, após cria-la é preciso passar os elementos a serem criados. As factorys do laravel utilizam a biblioteca faker, a documentação está em <https://fakerphp.github.io>. Após isso, deve-se definir o que será passado nas colunas por exemplo: ‘nome’ => $this->faker->name. Após isso deve-se passar chamar a factory criada na seeder da respectiva model com factory (\App\Models\Model::factory()->count(\*vezes desejadas\*->create()). Dessa maneira, na próxima vez que a seeder for executada, os registros serão criados automaticamente.

**Seção 9: Trabalhando com formulários**

**Aula 120**

Explicação sobre como a request funciona. Ao carregar a página, é feita uma requisição que que contém as informações a serem carregadas, básico. Porém o laravel consegue pegar a requisição de formulários enviados e armazaná-las em uma variável, que pode ser acessada passando Request $request como parâmetro no controller que se deseja receber a request. Existem alguns métodos que podem ser eutilizados com a request: $request->all(), que recupera o array associativo das informações; $request->input(‘\*name desejado\*’), que irá imprimir o valor da name desejada.

**Aula 121**

Realizadas inserções no banco com base na request realizada no site. Existem algumas maneiras de realizar a inserção: criando uma variável objeto da model SiteContato, passando a coluna com base no método input da request ($contato->nome = $request->input(‘nome’)) e no final utilizando método ->save() na variável contento um array associativo dos itens; Outra maneira é utilizando o método fill na variável contendo um objeto da model SiteContato ($contato->fill($request->all())); Mais uma maneira é utilizando o método create ($contato->create($request->all()). O método fill irá passar somente os registros que forem definidos na variável $fillable na model.

**Aula 122**

Para que não seja retornado um erro ao tentar acessar a página por conta das operações de inserção realizadas, foi alterada a rota post site.contato, para que dentro do controller, a requisição seja passada para a função salvar (muito massa!). Após isso, são passadas as operações para essa função para que sejam realizadas somente quando uma requisição do tipo post for realizada. Mas antes que seja feita a inserção no banco é necessário realizar a valizadação dos campos. Para isso o laravel possui a variável $errors que está presente em qualquer view, que contém informações sobre erros que acontecem na página. Além disso existe um metódo chamado ->validate(), que recebe um array associativo contendo o name dos campos do formulário enviado na requisição, que recebem como valores a validação a ser realizada, que no caso é de not null podendo ser escrita como ‘required’. Dessa forma, as informações nulas recebidas irão cair no validate, que irá realizar a request anterior ao receber o erro, pois na aula, a função salvar não renderiza nenhuma view, mas a página é exibiad do mesmo jeito após o erro na função por conta do valor nulo e isso é justamente por essa inteligencia do método validate, que retorna a requisição anterior, que no caso é o get no site.contato, que retorna o método contato(), que renderiza a view.

**Aula 123**

Mostradas as validações min e max do validate. Para mais informações das validações possíveis, podem ser lidas no link [https://laravel.com/docs/9.x/validation#available-validation-rules](https://laravel.com/docs/9.x/validation" \l "available-validation-rules). Quando deseja-se mais de uma validação por campo, elas devem ser deparadas por um pipe (|). Os métodos min e max esperam um parâmetro, o da quantidade de caracteres, que devem ser passados após um : (min:3|max:40).

**Aula 124**

É possível fazer com que os valores que foram submetidos no formulário sejam retornados aos campos em que estavam preenchidos. Isso se dá pois além de capitar os erros nas variável $errors, o laravel também possui a função old que pode ser utilizada em qualquer view, contendo os valores do formulário que foi submetido por último. Para imprimir os valores anteriores, basta fazer uma colocar a função old(‘\*name\*’). No caso do textarea é necessário verificar se o valor era diferente dde nulo, pois já havia a mensagem padrão escrtita.

**Aula 125**

Feita a recuperação do campo select e também foi diminuido o código das options do select, transformando sua informações em um array assciativo que for criado na model contato, contendo o value da option, recebendo a nome da operação como valor (Dúvida, Elogio, Reclamação). Após isso o array é passado para a view como parâmetro na renderização da view, que ddepois é passado para o componente também como parâmetro. Após o array estar no componente, é feito um foreach, que imprime uma option, que recebe dinamincamente o value e o conteúdo, além de realizar a comparação como valor old, que imprime selected caso o valor seja diferente de vazio, ficando: <option value=”$key” {{ old(‘motivo\_contato’) == $key ? ‘selected’ : ‘’ }}> {{ $motivo\_contato}}</option>.

**Aula 126**

Feito um ajuste na tela principal, que não estava recebendo o array que deveria estar presente no formulário. Para isso, o array presente no controller SiteContato foi copiado para o controller Principal também e em seguida foi passada view, novamente para o componente, resolvendo o erro.

**Aula 127**

Feita alteração para que o array de motivo contatos não seja feita a mão, mas sim por uma tabela no banco de dados, que pode ser acessada e utilizada de maneira dinâmica em quaquer parte da aplicação. É então criada uma nova model MotivoContato (-m para criar uma migration também). Na migration então. Além da criação da tabela, é criada a coluna motivo\_contato, que recebe no meu caso no máximo 40 caracteres. Em seguida, é criada uma seeder MotivoContatoSeeder, para popular com os 3 valores criados até o momento. Após realizada a migrate e o db:seed, os valores foram criado e podem ser recebidos nas variáveis já criadas de motivo\_contato, recebendo o método estático ::all() de um objeto MotivoContato. Porém o array recebido nesse método é bem mais elaborado, contendo diversas outras informações da busca no banco, então no formulário, é necessário modificar a exibição dos elementos: No lugar de $key, deve ser utilizado $motivo\_contato->id; No lugar de $motivo\_contato, deve ser utilizado $motivo\_contato->motivo\_contato. Dessa maneira, estão sendo recebidos os valores das determinadas colunas da tabela, realizando a mesma operação, porém de forma dinâmica e com fácil alteração e incrementação.

**Aula 128**

Feita a conexão entre a tabela motivo\_contatos com a tabela site\_contatos, já na segunda existe uma coluna com a mesma função da primeira, existem várias maneiras corretas de resolver o problema, mas a adotada foi a seguinte: Criar a coluna motivo\_contatos\_id, passar todos os dados para a coluna recém criada, depois criar uma fk na tabela motivo\_contatos referenciando o id, então excluir a coluna motivo\_contato em seguida. Para isso, é criada uma nova migration que irá realizar as ações com o nome alter\_table\_site\_contatos\_add\_fk\_motivo\_contatos. Nela é feita toda a operação descrita anteriormente. Para passar os dados de uma coluna para a outra é introduzida uma nova função, a DB::statement, em que é possível passar uma query a ser executada em SQL, que no caso foi (UPDATE site\_contatos SET motivo\_contato = motivo\_contatos\_id). Em seguida é realizado o método down da migration, realizando o inverso das operações realizadas pelo up().

**Aula 129**

Mostrado o método de validação email na função validate, utilizada no controller do Contato. Faz com que seja necessario que um email válido seja submetido. Já realiza a função de required também.

**Aula 130**

Feito o redirecionamento de rotas após a inserção de um objeto SiteContato para que seja mandado para a tela inicial utilizando como retorno da função salvar redirect()->route(‘site.index’). Após isso, somente são alterados os nomes da coluna anterior de motivo\_contato para o atual.

**Aula 131**

Utilizado o método de validação unique na função validate. Como parâmetro, deve-se passar a tabela no banco de dados em que o campo será comparado. Na aula foi colocado no nome, somente para fins de testes.

**Aula 132**

Feito um if para validar se existe algum erro na variável $erro, utilizando ->any(). Após isso, foi recuperado com o método ->all(), um array somente com as frases dos erros, que foram imprimidas com um foreach, melhorando levemente a exibição dos erros.

**Aula 133**

Melhorado mais um pouco a exibição dos erros, fazendo com que cada erro específico seja exibido debaixo do seu campo. Para isso foi criado um if ternario embaixo de cada campo, validando se há algum erro para aquele name específico, com a função ->has(‘\*name\*’). Caso exista, irá então imprimir o primeiro erro daquele name (pois existe mais de uma validação para vários desses campos), utilizando $errors->first(‘\*name\*’).

**Aula 134**

É possível passar mensagens personalizadas de acordo com a validação feita. O método validate() aceita dois parâmetros, o primeiro com os campos e as validações e o sugundo, um array com as mensagens para as determinadas validações. Para definir uma mensagem específica para um campo, deve-se usar \*campo\*.\*validação\* (nome.required). Também é possível passar uma mensagem genérica, definindo somente a validação, sendo possível por meio de :attribute, retornar de forma dinâmica, o name que está retornando o erro. Também foi colocado ambos os arrays em variáveis fora do método e depois passadas as variáveis contendo os arrays, de modo a deixar o código mais elegante.

**Seção 10: Middlewares**

**Aula 135**

Introdução ao middlewares. Basicamente são códigos que irão captar a request ou a response de uma página web, permitindo que ações sejam realizadas antes do usuário acessar o core da aplicação, verificando se o usuário está autenticado por exemplo, ou também, recebendo informações sobre qual usuário acessou a página ou se o usuário aceitou ou não os termos, bloqueando o acesso. Também é possível padronizar as responses quando se está trabalhando com CORS(Cross-Origin Resource Sharing), padronizando as responses para que possam ser tratadas por APIs, por exemplo.

**Aula 136**

Criação de uma middleware, recomendado acessar a documentação <https://laravel.com/docs/9.x/middleware> em caso de dúvidas. Para criar um middleware basta utilizar o comando php artisan make:middleware \*Nome do Middleware\*, sendo recomendação colocar middleware no final do nome. A pasta dos middlewares estão localizadas no diretório app\Http\Middleware. Para que um middleware seja executado, é necessário que ele seja passado nas rotas, utilizando o método estático middleware(\*Middleware\*::class), sendo necessário utilizar o método use no cabeçalho do código. Já dentro do middleware, a resposta é realizada com um return Response (\*conteúdo\*). Somente com essa resposta, a requisição será finalizada no middleware, não passando adiante.

**Aula 137**

Criada uma nova model, que será responsável por criar logs com os ips e as rotas acessadas, para isso foi realizado php artisan make:model LogAcesso -m criando já a migration também, que recebeu a string log com um máximo de 200 caracteres. Após isso foi definido a coluna logs no variável fillable para que possa ser preenchida em massa, então utilizado o create no middleware. Para recuperar o ip e a rota acessada, deve-se utilizar a variável $request, que possui essas informações. Para recuperar o ip, deve-se utilizar $request->server->get(‘REMOTE\_ADDR’). Já a rota pode ser acessada por $request->getRequestUri(). Ambas foram armazenadas em variáveis e passadas no create, informando: O ip $ip acessou a rota $rota.

**Aula 138**

Feita a implementação do middleware criado em uma controller. No caso foi criada no controller sobre nós. Para realizar a implementação, basta utilizar o método use no cabeçalho do código e criar o método contrutor \_\_construct() {}, passando a função do middleware dentro, no caso sendo: $this->middleware(LogAcessMiddleware::class);

**Aula 139**

É possível passar os middlewares em todas as rotas de uma maneira mais simples. Isso é possível acessando o o código Kernel.php no diretório app\Http\. Neste arquivo possui um trecho referente aos middlewares que serão executados ou na parte web, ou na parte de apis. No caso, queremos alterar somente a parte do web, bastando introduzir o middleware no array, de maneira padronizada: \App\Http\Middleware\LogAcessoMiddleware::class.

**Aula 140**

Colocado um apelido para o middleware criado. O nome pode ser definido no kernel também, estando na variável protegida $routeMiddleware. Nesse array deve-se passar a string com o nick do middleware, recebendo o diretório do mesmo. Após isso, quando for necessário a utilização do middleware, basta utilizar o nick passado, não sendo necessário utilizar o use ou o diretório do middleware, pois o laravel já terá a inteligência de utilizar o código desejado.

**Aula 141**

É possível realizar um encadeamento de middewares, antes que a página atinga o código principal. Isso na realizade, já era fato desde o inicío da utilização do laravel, que já realiza a execução de vários middlewares. Foi criado um novo middleware, que será responsável pela autenticação do usuário (não realizará isso de fato ainda). Após isso, foi passado o nick ‘autenticacao’ para esse middleware, e foi colocado para ser executado na rota app\, que será a parte administrativa do site. Em seguida, foi passada a operação return $next($request) no middleware de LogAcesso, para que a requisição passe por ele, chegando após isso na outra middleware criada, chegando até ela e informando a informação desejada.

**Aula 142**

Passado o middleware criado no grupo presente no código. Basta usar o método middleware() na frente da chamada do grupo.

**Aula 143**

Mostrado como passar parâmetros para middlewares. Para realizar isso, basta colocar : após o nick do middleware, passando diretamente uma string por exemplo. Esses parâmetros irão ser recebidos no método handle pelas suas respectivas variáveis. Para passar mais variáveis, basta passar elas separadas por vírgulas.

**Aula 144**

Utilizando o $next($request), é possível será recebido a resposta do site, quando este trecho é armazenado em uma variável, sendo possível alterá-la. Na aula, foram alterados o status code e o texto do status, utilizando o método estático ->setStatusCode(\*código\*, ‘\*mensagem\*’) na variável que recebe a response do site.

**Seção 11: Autenticação de Usuários (revisando e praticando os assuntos abordados)**

**Aula 145**

O método de autenticação desenvolvido será o normal, utilizando nome e senha. Alterada a rota get de login que irá chamar o LoginController@index, em seguida, foi criado uma nova controller chamada LoginController e nela foi renderizada uma view(‘site.login’). Após criado a view com esse diretório, foi feito a cópia do formulário do contato. Foram apagados os conteúdos da classe informacao-pagina e foi criado um formulário que possui como action o método blade route, passando o nome do diretório da view (site.login) com o método post. Foi então criada uma nova rota com o método post, para receber o formulário.

**Aula 146**

Recebidos os valores passados no formulário e criadas as variáveis $regras e $feedback para poder utilizar o método validate na variável $request, recuperada na função autenticar. Também passadas as mensagens de erro nos campos, além de recuperar os valores old.

**Aula 147**

Colocados os valores recuperados em variáveis separadas para email e senha. Em seguida for criado um objeto da model User, que já é criada automáticamente pelo laravel. É criada outra variável para receber a busca no banco, procurando registros no banco em que o email e senha sejam os mesmos que os inseridos no formulário (um registro de teste é criado anteriormente). Após o ->get(), é adicionado o método ->first para recuperar somente o primeiro registro, pois é o único necessário. Após isso é realizada a verificação com um if, verificando a existência do ->name, caso exista exibindo ‘usuário existe’ e caso não ‘usuário não existe’.

**Aula 148**

Fieto um redirect para o login, no caso em que o usuário não exista, passando um parâmetro [‘erro’ => 1], para que seja tratado. Em seguida, é adicionado o parâmetro na rota, recebendo /{erro?} com o operador opcional, pois não é obrigatório o recebimento do valor. Após isso é necessário receber a request na função index também. Em seguida, é criada a variável $erro que recebe o get de erro da request e é passado a variável como parâmetro pra view renderizada e lá é feita uma validação que imprime o valor da variável caso ela exista e não seja vazia.

**Aula 149**

Alterado o que será caso o usuário exista, agora irá iniciar uma sessão e serão definidos o nome e a senha para a sessão e é realizado um redirect para a página que precisa de autenticação, no caso a rota clientes. Também é alterado o middleware autenticação, que agora irá iniciar uma sessão e verificar se o campo email está presente e se ele é preenchido. Caso seja true, será passado com o $next($request), caso contrário sendo redirecionado para a página de login com o erro 2 que se trata da necessidade da autenticação para acessa a página.

**Aula 150**

Criados os controllers e views para as demais rotas da parte do app, sendo elas: /home (HomeController), /sair (LoginController@sair), /cliente (ClienteController), /fornecedor (FornecedorController), /produto (ProdutoController). Após a criação das controllers e views dessas rotas, foram copiados os templates basico.blade.php e topo.blade.php e feitos os ajustes para cada página, deixando elas funcionais.

**Aula 151**

Adiciona a função de sair na função sair() do login controller. Apenas destrói a sessão e redireciona para a página inicial.

**Seção 12: Finalizando o projeto Super Gestão**

**Aula 152**

Inicio da implementação do cadastro de fornecedores. Basicamente foram criadas outras duas rotas, sendo a já existente uma de pesquisa de fornecedores e a nova de cadastro de fornecedores, nenhum conhecimento novo introduzido. A outra rota criada é responsável por mostrar uma lista com os fornecedores buscados.

**Aula 153**

Realizada a implementação da adição de novos fornecedores com o formulário criado, também realizada a validação dos campos, nada de novo.

**Aula 154**

Feita a pesquisa dos elementos da tabela por meio de um encadeamento de pesquisas, que funciona de maneira bem eficaz, na medida em que não é necessário que todos os campos estejam preenchidos, mas somente um deles e caso nenhum seja preenchido, retorna todos os fornecedores criados.

**Aula 155**

Feita a edição dos campos. Criada uma nova rota editar que irá receber o id como parâmetro (funciona por get, pois é essa a operação realizada quando é clicado em um link (a)). É chamada uma nova operação editar no controller. Nele é realizada uma pesquisa com o id passado no get e também é onde é aberto o mesmo formulário de cadastro, porém recebendo uma variável que contém o registro que se deseja modificar, fazendo com que quando o formulário de cadastro for enviado, seja verificado se há um id passado por meio de um input hidden, para determinar se é um cadastro ou edição. Algo interessante é que após a edição foi utilizado um redirect para a rota da edição do item novamente, pois com esse redirecionamento, é realizada busca no elemento atualizado. Existem formas mais eficientes de realizar o que está sendo feito, principalmente utilizando as rotas update ou put por exemplo que são facilmente gerenciadas pelo laravel, mas por hora somente o get e post foram utilizados.

**Aula 156**

Realizada a páginação na tabela de consulta. No laravel, a paginação pode ser realizada ao invés de chamar o método get na variável que contém os registros do bando, utilizar o método ->paginate(\*qnts registros por página\*). Porém somente isso não irá apresentar os botões de troca de página. Para isso, deve-se ir na view e lá passar {{ $\*variável com os registros paginados\*->links() }}, onde será criado uma lista desordenada com os links das páginas, que deverão ser alteradas com css para melhor visualização. Após isso outro problema surgirá, quando for trocada a página será apresentado um erro, pois a requisição realizada nesse momento será get, sendo que a rota atual no momento só recebe post. Após criar a rota get, o problema será resolvido e a páginação irá funcionar. Outro problema é que quando for realizada alguma pesquisa com filtro, após trocar de página, o filtro será perdido. Para resolver, é necessário encaminhar para a view a variável $request->all(), pois ela é quem armazena os filtros e na view deve-se usar o método estático ->append($request), antes do ->links, para que o filtro seja considerado na páginação do elemento. E assim, de maneira simples a paginação está completa.

**Aula 157**

Mostrado alguns comandos que contém informações específicas sobre a busca paginada: ->count() exibe a quantidade de resgistros por página; ->total() mostra a quantidade total de registro encontrados na busca; ->firstItem() mostra o número do primeiro item da página atual; ->lastItem() mostra o último item da página atual.

**Aula 158**

Feita a exclusão de registros. Para isso foi criada uma nova rota para excluir que recebe o id como parâmetro e chama a função excluir no controller de fornecedor. Dentro dessa função é realizada a pesquisa do registro e em seguida utilizado o comando delete para realizar o softdelete do fornecedor. É possível utilizar o forceDelete para a exclusão de fato do banco, mas não é necessário em minha opinião.

**Aula 159**

Mostrada cração de um controller com o resource, técnica do laravel que já cria as funções que realizam as principais ações de um crud de maneira padronizada e estruturada: index(), que exibe a lista de registros; create() que exibe o formulário de criação do registro; store() que recebe o formulário de criação do registro; show() que mostra um registro específico; edit() que mostra o formulário de edição do registro; update() que recebe o formulário de edição do registro; destroy() que recebe os dados para a exclusão do registro. Para utlizar o resource, basta especificar -r na instrução do controller a ser criado. Também é possível criar a model junto da controller, especificando --model=\*nome da model\*. No caso, o controller de produtos foi excluido e criado novamente com resource, a linha de criação usada foi: php artisan make:controller -resource ProdutoController --model=Produto.

**Aula 160**

É possível criar as rotas de maneira unificada quando se trata de uma resource. Basta definir Route::resource(‘\*model\*’, \*controller\*::class); após importar o controller, que todas as rotas utilizadas terão sido criadas, de forma dinâmica, com os métodos http específicos para a operação como delete, put e patch.

**Aula 161**

Explicação básica dos métodos http, de forma superficial, sem acrescentar nada no que já sabia antes. Recomendado site <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods> para consulta sobre os métodos HTTP.

**Aula 162**

Iniciada a implementação do cadastro de produtos, com a index, que irá exibir a lista de produtos cadastrados, de maneira bem parecida com a consulta de fornecedores. Criada uma variável que recebe paginate de produto com 10 registros. Definido também quais colunas seriam fillables na model de produto para poderem ser preenchidos em massa.

**Aula 163**

Feito o próximo método do controller produto, create. Foi criada uma nova página que irá exibir o formulário de cadastro. O formulário foi pego de base com o de fornecedor, somente realizado ajustes para que possua as colunas do produto, sendo que a unidade foi feita dinâmicamente com um foreach da tabela unidades.

**Aula 164**

Realizado o store de produto. Realizado de maneira muito simples, somente utilizando create($request->all()).

**Aula 165**

Feita a validação dos campos da mesma maneira realizada até este ponto do curso. Foi utilizado um método diferente dessa vez, pois era necessário no select que a option escolhida fosse uma presente na tabela de unidades. Para isso, foi utilizado o método de validação exists: que espera dois parâmtros: o primeiro a tabela do banco, com o nome da tabela conforme o banco e o segundo é a coluna a ser comparada, no caso unidade\_id.

**Aula 166**

Implementado o show, que exibe um produto em específico em detalhes. Para isso foi criado uma tabela(html kkk) exibindo as informações recuperadas com base no id que deve ser passado como parâmetro e recebido no método.

**Aula 167**

Implementado o método edit, que exibe o formulário de edição. No caso foi criado um novo formulário que recebe os calores do id passado, porém é possível e mais interessante fazer um componente com o formulário de produtos, para então utilizá-lo dinamicamente nos tanto na inserção e edição, com os devidos parâmatros, algo que será feito mais adiante.

**Aula 168**

Feita a implementação do método update, que recebe o formulário de edição e persiste os dados no banco. Esse método utiliza ou método http put ou patch. A diferença entre os dois é que o put está relacionado a edição de **TODOS** os valores do objeto, já o patch é referente a alteração de **ALGUNS** atributos do objeto. Como no caso o formulário de edição tem o potencial de editar todos os atributos do objeto, o put será utilizado com a definição de @method(‘PUT’) logo abaixo do @csrf no formulário, para indicar que os dados devem ser interpretados como put pelo backend. Após isso, feito a recuperação e persistência dos dados normalmente, sem nada de novo até o momento.

**Aula 169**

Realizado o método destroy. Como sua execução é baseada no método post, é necessario criar um formulário dentro da tabela que está exibindo os registro e sua action é a rota destroy. Dentro desse fomulário, além da definição do csrf, tabém é necessário definir o método http que no caso é o delete. Após isso basta recuperar o id do produto com base no formulário enviado e excluir o registro, redirecionando novamente para a index.

**Aula 170**

Melhorada a utilização do edit, que agora utilizará também o formulário de criação de produtos. Passado a fazer uma verificação se foi passada para a view o id do produto, presente na variável $produto, que é instanciada no edit e não no create.

**Aula 171**

Refatorada novamente a edição e criação dos produtos, dessa vez, transformando o formulário em um componente, que será compartilhado entre as views. Criado o diretório \_components dentro da pasta de produtos, que recebeu o formulário utilizado. Passado então o component nas duas views com seus respectivos dados, os mesmos que são enviados até a própria view de edição e criação.

**Aula 172**

Explicação sobre os relacionamentos entre tabelas em um banco de dados. Relações um para um acontecem entre duas tabelas, uma principal e outra secundária, onde a secundária possui uma constraint unique, para que os dados registrados na tabela secundária sejam relacionadas a somente um registro da tabela principal, não podendo ser referenciada novamente por outro registro. Já o relacionamento um para muitos não possui a constraint. Já o relacionamento muitos para muitos se trata da utilização de uma terceira tabela, que irá associar registros das outras duas tabelas.

**Aula 173**

Feito a criação de uma model, controller, resource e duas views para a tabela produto detalhes, que irá conter detalhes como largura e altura dos produtos cadastrados.

**Aula 174**

Criados os métodos create e store, baseados nas páginas do produto, com os ajustes feitos.

**Aula 175**

Criados os métodos edit e update, baseados nas páginas do produto, com os ajustes feitos.

**Aula 176**

Feita a implementação das informações contidas em produto\_detalhes na view index de produto, para isso foi criado um foreach que passa por cada um dos objetos da collection recuperada pelo paginte deito na variável $produtos. Dentro do foreach é então pesquisado com o where, na tabela produto detalhe onde a coluna produto\_id é igual ao id do produto, então com o método ->first() é armazenado o primeiro produto encontrado, para que não seja retornado uma collection mas sim um produto. Após isso é verificada a existencia da variável que armazena a busca anterior e com ela existindo, é então passado para a variável $produtos[$key][‘\*medida\*’] = $produtoDetalhe->\*medida\*, isso sendo realizado para comprimento, altura e largura. A sintaxe utilizada referente a $produtos se faz necessaria pois $produto que é criado no foreach é apenas uma cópia do array(collection no caso) original, então setando esses valores não irão realizar alterações no original, que no caso é o que se deseja alterar. Na aula seguinte será mostrado uma maneira mais eficiente e elegante de realizar o mesmo processo.

**Aula 177**

Na model produto, é criada uma função chamada produtoDetalhe, que retona $this->hasOne(‘App\Models\ProdutoDetalhe’). Essa função verifica na outra model por uma fk, que no caso é produto\_id e então retorna o objeto de produtoDetalhe que pode ser acessado depois no objeto produto, como se fosse mais um atributo, ficando então $produto->produtoDetalhe->comprimento, por exemplo, reduzindo uma dezena de linhas para 3.

**Aula 178**

Mostrado o método belongsTo, que faz o oposto de hasOne. Com ela, é possível recuperar os valores de um objeto de produto, dentro do objeto produtoDetalhe.

**Aula 179**

Feito a mesma operação das aulas anteriores, porém dessa vez, foram criadas duas novas models para que realizem a mesma operação, porém para exemplificar como seria caso as models fossem de nome diferentes da padronização do eloquent orm. As principais diferencças estão no fato de que nas novas models item e itemDetalhe, é necessário informar a tabela protected $table, passando o nome correto da tabela, do mesmo jeito que ela está no banco, ‘produtos’ para a model item e ‘produto\_detalhes’ para a model itemDetalhe. Em seguida, nos métodos de hasOne e belongsTo é necessário definir mais dois parâmetros para as funções. No caso de hasOne o segundo parâmetro se refere ao nome da coluna fk que está presente na tabela que possui a fk, no caso ‘produto\_id’ e o terceiro parâmetro se refere a qual coluna na própria tabela está sendo referenciada na fk, no caso sendo ‘id’. No belongsTo é realizado o processo contrário, onde o segundo parâmetro se refere a qual coluna da própria tabela está presente a fk, no caso ‘prdouto\_id’ e como terceiro parâmetro, qual é a coluna na tabela principal que está sendo referênciada, no caso ‘id’. Em ambos os métodos, os parâmetros acabam sendo os mesmos mas é importante ter a distinção do que está ocorrendo especificamente. Após isso é mudado a forma em que é dado o edit de produtoDetalheController, que antes recebia um produto, mas agora recebe um id, que então é buscado com um find na model item\_detalhe, que busca na tabela produto\_detalhe ($produtoDetalhe = ItemDetalhe::find($id)), fazendo com que a execução seja a mesma por parte das views, porém sendo realizada de maneira diferente no controller. Também é alterado o nome dos métodos que possuem o hasOne (itemDetalhe()) e belongsTo(item), fazendo com que seja necessário a mudança na chamada dos objetos, onde está sendo imprimidos os valores dos objetos da relação para o mesmo nome do método para que em seguida seja possível chamar o atributo desejado normalmente.

**Aula 180**

Mostrado as diferenças entre o Lazy Loading, que é o modo de carregamento padrão para os objetos e o Eager Loading. O lazy loading ocorre quando é realizada uma busca no banco pelos registros de uma tabela que são guardadas em uma variável. O que acontece no lazy loading é que os registros que possuem um relacionamento, não serão carregados até que o relacionamento seja realizado, como é no caso do relacionamento produtoDetalhe, presente na model produto. Caso os registros sejam passados para um json por exemplo antes da chamada do relacionamento, presente na tabela html criada para exibir os registros, o json não conterá os valores de produtoDetalhe, porém caso o json for criado a partir da variável que contém os registros após a tabela, onde é realizada a chamada do relacionamento, os produtos serão carregados e estarão presentes no json. Para realizar o Eager Loading, que consiste no chamada dos relacionamentos a partir do momento em que os registros são buscados, basta definir o método estático with([‘\*relacionamento1\*’, ‘\*relacionamento2\*’]) antes da método all() ou find(), por exemplo. Isso fará com que os relacionamentos presentes nessas models sejam carregados no momento da busca, de acordo é claro com quais relacionamentos forem definidos no método with.

**Aula 181**

Realizada a mesma operação da aula anterior, porém dessa vez com o produtoDetalheController, onde é colocado o método with na variável $produto\_detalhe, que realiza um find no id passado como parâmetro no método edit do controller.

**Aula 182**

Feita a criação de um relacionamento entre tabelas de um para muitos, no sentido de que um fornecedor pode ter N produtos. Para isso foi criada uma migration que irá realizar a adição da foreign key na tabela produto. Logo, foi criada a coluna forncecdor\_id após a coluna id da mesma tabela. Logo em seguida é criada a fk apontando para a tabela fornecedores, na coluna id. Existe porém um problema, pois o quando essa coluna e fk forém criadas, haverá um problema pois o tipo da coluna unsignedBigInt não pode ser nula e inicialmente seria o caso para os registros já presentes na tabela então para resolver isso, é criado então utilizando a classe DB, para criar um novo registro na tabela de fornecedores com o método insertGetId, que é armazenada na variável $fornecedor\_id, que então é utilizada na criação da coluna fk como valor default, fazendo assim com que os registros já criados na tabela produtos recebam como valor para a fk, o novo registro criado na tabela fornecedores, resolvendo assim o problema. Após realizado o migrate, o relacionamento está implementado.

**Aula 183**

Feita a exibição do fornecedor utilizando o método belongsTo na model de produto, apontando para a tabela de fornecedores. Também é possível definir o relacionamento no controller de produto, para que seja utilizado o Eager Loading.

**Aula 184**

Feito justamente o que havia escrito na última aula, tratamento para que o relacionamento presente em produto seja realizado por eager loading.

**Aula 185**

Realizado o relacionamento inverso, em que um forncedor pode ter vários produtos. Para isso, no controller de forncedor, foi utilizado o método hasMany, que indica o relacionamento um para N. Após isso, foi criada uma tabela na view de fornecedores dentro de cada td informando quais produtos foram criados por cada fornecedor, recebendo os dados do relacionamento.

**Aula 186**

Alteradas as operações de crição e edição de produtos, para que possa ser possivél associar o fornecedor ao cadastro.

**Aula 187**

Feita a criação de três novas models e tabelas, para que seja implementado o relacionamento de N para N. As models/tabelas criadas foram as de clientes, pedidos, pedido\_produtos. Pedidos e produtos terão o relacionamento N para N, que é realizado com o auxilio da tabela pedido\_produtos. Além disso, os pedidos realizados precisarão informar qual foi o cliente que fez o pedido, em um relacionamento 1 para N. Para isso foi criado uma migration que realiza a criação das tabelas necessárias, além das foreign keys utilizadas pelas mesmas.

**Aula 188**

Implementado os controllers e as rotas das tabelas criadas na aula anterior, nada de novo.

**Aula 189**

Feito a implementação da index de cliente, pegando como base a index de produto, modificando a tabela html para wxibir somente as colunas da tabela de clientes.

**Aula 190**

Implementados o create e o store do controller de cliente, com base no controller de produto.

**Aula 191**

Feita a implementação da index de pedido controller, exibindo o id do pedido e do cliente.

**Aula 192**

Criada a tela de cadastros de pedidos, com base na que foi criada para clientes, utilizando as colunas presentes na tabela de pedidos, sem a ligação com pedido\_produtos ainda.

**Aula 193**

Feita a implementação de pedido\_produto, que será acionada ao clicar no botão de adicionar produtos na tabela html de pedidos, onde levará para um formulário que permitirá a escolha dos produtos a serem adicionados. Também foram criadas as rotas de create e store de maneira manual, visto que o create e o store necessitam do parâmetro do pedido, pois não são cadastros “independentes”, na medida que pertencem a um pedido.

**Aula 194**

Implementado o relacionamento belongsToMany e também o método store de pedidoProduto, além de exibir quais produtos foram associados a cada pedido. Inicialmente, feita a validação da request para que sejam armazenados os produtos na tabela pedido\_produtos com o validate. Após isso, foi realizada a inserção na tabela de pedido\_produtos com base no id do produto e do pedido recuperado no formulário store. Após a criação do método de relacionamento produtos na model de produtos, é possível realizar o eager loading no create de pedidoProduto utilizando $pedido->produtos. Essa chamada irá carregar os elementos presentes em produtos para o pedido, para que possam ser exibidas informações dos produtos na tela de cadastro de produtos para o pedido.

**Aula 195**

Feito a implementação de belongsToMany para a model de produto, para que seja exibido na index de produto, quais pedidos cada produto faz parte (não muito eficiente na minha opinião). Após a recuperação dos dados do pedido, é exibido na tabela de produtos, quais pedidos tal produto faz parte.

**Aula 196**

Na recuperação de dados por meio das funções belongsToMany(One) e hasMany(One), nem todos os dados são recuperados, como por exemplos as colunas created\_at e updated\_at. Para especifircar quais valores se deseja recuperar, basta utilizar o método estático withPivot(‘coluna1’, ‘coluna2’...). Utilizando este método é possível recuperar dados que normalmente não seriam enviados, com base nessa operação, foram incluidas as datas de inclusão dos itens no cadastro de pedidoProdutos. Para acessar esses dados na view, basta chamar por pivot e a coluna desejada, exemplo: $produto->pivot->created\_at->format(‘d/m/Y’).

**Aula 197**

Adicionada a coluna de quantidade na tabela de pedido\_produtos. Também utilizada a inserção de registros pelo método attach, que realiza a inserção de registros relacionados, facilitando a inserção das relações, além de ter a funcionalidade de registrar quantos novos registros forem necessários de uma vez, no caso sendo possível adicionar vários produtos a um pedido de uma vez só.

**Aula 198**

Fieto o outro lado da operação da aula passada, sendo a exclusão de registros qque foram criados por attach. Existem duas maneiras, a convencional que basicamente faz a busca no banco pelo id do pedido e do produto relacionado ao pedido, então o exclui e também existe o comando dettach, que realiza a mesma função porém com base no relacionamento já existente, reduzindo assim o tanto de código necessário.

**Aula 199**

Feita a utilização do pivot, para recuperar o id da tabela pedido\_produtos, para que na exclusão de um produto, seja realizado pelo id da tabela pedido\_produtos diretamente, não sendo necessário especificar qqual é o produto e em qual pedido está para realizar a exclusão. Porém ainda acaba sendo necessário passar o id do pedido para que seja possível realizar o redirecionamento da tela após a exclusão.