Desenvolvimento de Aplicações Web Utilizando Framework Laravel

Everton S. R. Dewes

Parte - 1

Nesta **Parte - 1** teremos a apresentação dos principais conceitos e também o desenvolvimento prático de uma aplicação Full Stack simples para um Chat online.

Na **Parte - 2** será abordado com mais profundidade cada um dos módulos, suas configurações, melhores práticas e seus comandos sendo criada uma aplicação mais complexa para um registro de Projetos.

1. Conceitos

1.1. Frameworks

Um framework em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica. Um framework pode atingir uma funcionalidade específica, por configuração, durante a programação de uma aplicação. Ao contrário das bibliotecas, é o framework quem dita o fluxo de controle da aplicação, chamado de Inversão de Controle.
https://pt.wikipedia.org/wiki/Framework

Um framework de desenvolvimento é uma “base” de onde se pode desenvolver algo maior ou mais específico. É uma coleção de códigos fonte, classes, funções, técnicas e metodologias que facilitam o desenvolvimento de novos softwares. (MINETTO, Elton. Frameworks para Desenvolvimento em PHP. 1. Ed. São Paulo: Novatec Editora, 2007. p. 17-19.)

Desta forma além de bibliotecas de funções e componentes prontos temos também uma estrutura de trabalho pronta que guia o desenvolvimento do sistema através de melhores práticas, mantendo um padrão de codificação e organização mesmo com equipes grandes. Desta forma evitamos que cada programador escreva o código com um estilo particular, facilitando o reuso e aumentando a manutenibilidade da aplicação.

1.2. Laravel

O Laravel é um framework criado por Taylor Otwell em 2011 tendo como base o Symfony, outro framework popular do PHP. Outros frameworks populares além do Laravel e Symfony são o ,e Zend Framework.

Laravel tem o foco no desenvolvimento de aplicações de forma rápida mantendo o código limpo, simples e de fácil legibilidade.

A página oficial: https://laravel.com/

A comunidade: https://laracasts.com

Laravel possui diversas extensões como por exemplo o Horizon se trata de uma painel de controle otimizado para o monitoramento de métricas do sistema e o Laravel Dusk que fornece uma interface expressiva e fácil de usar, para teste e automação de navegador. Além dessas ferramentas, o framework é evidenciado por suas facilidades fornecidas para trabalhos em equipe e autenticações simples. Outras funcionalidades que merecem destaque são suas bibliotecas orientadas a objetos; a utilização do algoritmo bcrypt, para garantir a segurança de senhas; e a migração fácil de bancos de dados.
https://configr.com/laravel/

Além do framework gratuito, são fornecidos uma série de produtos para facilitar a vida dos programadores. Três dos principais produtos oferecidos são o Envoyer, o Lumen e o Spark. Envoyer se trata de uma ferramenta de implantação totalmente voltada para PHP, permitindo implantações em servidores múltiplos, monitoramento e notificações, além de outros benefícios, tudo isso sem que aplicação ou site fiquem inativos. Lumen se trata de um micro-framework desenvolvido para a criação de micro-funções de forma rápida e poderosa. Por fim, o Spark é uma aplicação que busca diversas funcionalidades, como por exemplo, cobrança de inscrições, autenticação, renovação de senha, fotos de perfil etc. Assim, o desenvolvedor não precisa se preocupar em escrever linhas de código do zero para criar essas funcionalidades.

Apesar de ter sido lançado há apenas 6 anos, Laravel se tornou um dos mais importantes frameworks para PHP em toda a internet. Além de seu design intuitivo, funcionalidades essenciais para desenvolvimento web e ferramentas inovadoras, o framework se distingue por sua crescente e engajada comunidade. Para auxiliar ainda mais aqueles que utilizam a biblioteca, no site oficial do Laravel é possível ter acesso aos Laracasts, uma plataforma centenas de vídeos tutoriais que auxiliam os programadores nas mais diversas tarefas, seja criar um fórum online, configurar sistemas de pagamento ou vídeos explicitando o que há de novo nas últimas versões lançadas do framework.

Laravel já vem com um esquema forte de segurança com com controle de autenticação, autorização e prevenção contra injeção de SQL.

https://www.easylaravelbook.com/blog/how-laravel-5-prevents-sql-injection-cross-site-request-forgery-and-cross-site-scripting/

Obs: A versão utilizada neste manual é a 5.6

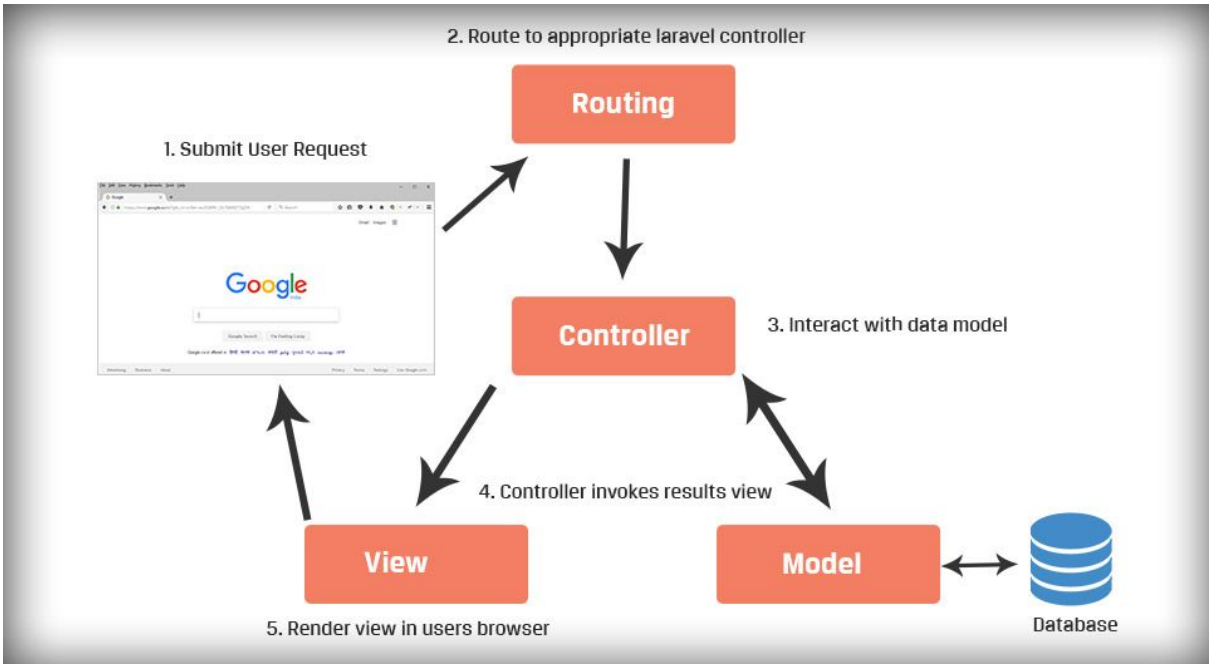
1.2.1. Model View Controller - MVC

MVC é um padrão de projeto muito utilizado para o desenvolvimento Web separando o sistema em uma parte de visão que irá ser apresentada para o usuário, uma parte de controlador que será responsável pelo fluxo dentro da aplicação e uma parte de modelo que cuidará do armazenamento dos dados.

Desta forma é possível, por exemplo, alterar o layout do site, alterar o fluxo de dados ou mesmo transformar o nosso modelo de forma simples e rápida comparado a um modelo em que regras de negócio e interface estão todos em um único bloco.

No Laravel teremos ainda o componente Router que será visto durante o curso.

Abaixo temos uma ilustração mecanismo:



http://www.savecontactform7.com/everything-you-need-to-know-about-laravel-framework

1.2.2. Eloquent

Eloquent é um ORM (Object Relational Mapper), que permite usar as tabelas do banco de dados em formato de objetos relacionais deixando a manipulação dos dados mais natural. Assim o programador não precisa utilizar o SQL para buscar, atualizar, criar e

deletar dados do banco. Outros ORM famosos são o JPA, Hibernate, Sequelize (do Node.js).

O Eloquent foi desenvolvido utilizando-se o padrão de projeto Active Record, desta forma o nosso modelo terá as funções necessárias para manipular os dados.
https://pt.wikipedia.org/wiki/Active_record

1.2.3. Blade

Blade é um simples, porém poderoso motor de templates (template engine) provido pelo Framework Laravel. Com ele você pode criar muito mais do que simples views, mas templates complexos contando com componentes que podem ser herdados e usados em várias views, sem o uso de PHP plano. O Blade vai compilar suas views e salva-las em cache em PHP plano, o que significa que sua aplicação terá zero de overhead em relação às views.
http://clubedosgeeks.com.br/programacao/php/laravel-usando-blade-templates

2. Projeto com Laravel

2.1. Ferramentas

2.1.1. PHP

O PHP é a linguagem/plataforma que suporta o Laravel e pode ser baixado diretamente do site "https://php.net".

Para que todo o sistema funcione corretamente é importante habilitarmos dentro do "**`php.ini`**" :

```
extension=php_openssl.dll // ou openssl
...
extension=php_pdo_mysql.dll // ou mysql  (extension=pdo_mysql)
...
extension=php_mbstring.dll  // ou mbstring
```

(obs: lembre de verificar se o PATH do windows contém o diretório do php.exe.)

Para testar abra o "`cmd`" e digite:

```
> php -v
```

Se quisermos ver as opções:

```
> php -help
```

2.1.2. Composer

Composer é um gerenciador de pacotes para o PHP, que além de baixar automaticamente as dependências necessárias ainda mantém elas atualizadas. Ele pode ser encontrado em "https://getcomposer.org/"

Dentro do "**php.ini**" é necessário habilitar a opção para o composer:

allow_url_fopen=on

(obs: também é necessário verificar se o executável está no PATH)

Para testar a instalação execute no prompt de comando ("cmd"):

```
> composer -V
```

Obs: 'V' maiúsculo.

Para a ajuda:

```
> composer -h
```

Obs:

É possível realizar a instalação dos pacotes FrontEnd usando o npm install

https://www.npmjs.com/get-npm

```
> npm install
```

2.1.3. MySQL

É o banco de dados que iremos utilizar durante nossas aulas e será utilizado para persistir os dados.

Após termos o servidor em execução devemos criar uma base de dados para ser utilizada pelo nosso projeto:

```
1. create database chat;
```

2.2. Criando um Projeto

Criar um projeto com a ajuda do Composer é muito simples, bastando ir até a raiz do "`httpd`" do PHP e executar o comando:

```
> composer create-project laravel/laravel Chat --prefer-dist
```

ou para lermos a última versão de desenvolvimento:

```
> composer create-project laravel/laravel Chat --prefer-dist
dev-develop
```

Com este comando os pacotes selecionados serão baixados da internet eo projeto será preparado com a sua estrutura básica (este processo pode demorar um pouco)

É possível também especificar uma versão do Laravel a ser instalada:

```
> composer create-project laravel/laravel Chat "5.5.*"
--prefer-dist
```

A partir deste ponto a estrutura do projeto está criada e com nossa ferramenta de desenvolvimento já podemos criar um projeto com fontes existentes. É possível utilizar qualquer ferramenta de desenvolvimento PHP como por exemplo:

1. PhpStorm
2. NetBeans
3. Sublime Text
4. Nuclide.io

Outras IDEs podem ser encontradas em: https://imasters.com.br/desenvolvimento/comparacao-das-melhores-ides-de-php-de-2016-e-2017-para-windows-mac-e-linux/?trace=1519021197&source=single

OBS: caso o projeto tenha sido baixado do GitHub, os pacotes podem ser instalados utilizando-se os comandos:

```
> composer install
```

```
> composer update
```

2.2.1. Alias no apache.conf

```
<Directory "C:/www/Chat/public">
    DirectoryIndex index.php
    AcceptPathInfo on
    AllowOverride None
    Options None
    Order allow,deny
    Allow from all
    Require all granted
    Options +FollowSymLinks

    RewriteEngine on
    RewriteBase /Chat
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)$ index.php/$1 [L]

</Directory>

Alias "/Chat" "C:\www\Chat\public"
```

2.3. Estrutura

As principais pastas (diretórios) são:

app	Nesta pasta você encontrará a maioria dos códigos de controle do seu sistema, como controladores, Modelos e definições de rota. Se você não sabe o que é um Controlador, modelo ou definições de rota, não se preocupe, estudaremos sobre isto em um outro artigo.

bootstrap	Nesta pasta você encontrará os arquivos responsáveis pela inicialização do Laravel como o autoload, app e a pasta cache.
config	Nesta pasta você encontrará todos os arquivos responsáveis pela configuração do sistema.
database	Nesta pasta você encontrará as famosas migrations que são compostas pelas estruturas das tabelas do seu banco de dados e são como uma espécie de backup desta mesma estrutura.
node_modules	Módulos do Node
public	Este é o diretório público do projeto, quando seu sistema estiver online este será o diretório que o servidor irá apontar para startar os processos de inicialização da pasta bootstrap. Nesta pasta será o local onde existirá todos os arquivos públicos como o htaccess, robots.txt, folha de estilo, scripts, downloads ou imagens.
resources	Nesta pasta você encontrará os arquivos que são importantes para a entrega das views do seu sistema, aqui você encontra a pasta assets que contém os arquivos JS e SASS/LESS, a pasta Lang que armazena os arquivos de idioma e a tão famosa pasta VIEWS (Iremos falar sobre as Views em outro artigo).
routes	Nesta pasta você encontrará todas as definições de rota do sistema, sejam elas rotas HTTP, rotas de comandos do Artisan ou rotas de console.
storage	Nesta pasta você encontrará os arquivos de caches, arquivos compilados e logs do sistema.
tests	Nesta pasta você encontrará os testes de unidade e integração do sistema.
vendor	Nesta pasta você encontrará todas as dependências instaladas pelo Composer. Esta pasta é ignorada pelo Git (Se você não sabe o que é o Git, não se preocupe, iremos falar sobre ele no próximo artigo) pois o Composer sempre deve ser executado como parte do processo de implantação em outros servidores.

https://laravel.com/docs/5.5/structure

https://www.hmtec.com.br/artigo/estruturas-de-pastas-e-diretorios-do-laravel

2.3.1. app

Console	Comandos disponíveis no console “`php artisan`”, podendo ser personalizado para a aplicação.
Events	Esta pasta não é criada por padrão. Ela irá conter os arquivos de eventos criados com os comandos “`event:generate`” e “`make:event`”. Eventos podem ser utilizados por outras partes da aplicação avisando que uma determinada ação ocorreu, proporcionando uma grande flexibilidade e desacoplamento
Exceptions	Classes de exceções criadas para o sistema atual
Providers	Contém os provedores de serviços. O Laravel já vem com alguns sendo possível adicionar novos. Serviços provêm um ponto inicial da aplicação com containers, registradores de eventos ou qualquer outra tarefa para as requisições que viram.
Http	Esta pasta contém a lógica para o tratamento das requisições do sistema
Http\Controllers	Controladores são usualmente utilizados para manipular grupos de requisições relacionadas, como um CRUD em um determinado modelo.
Http\Middleware	Middleware fornecer mecanismos convenientes para filtrar as requisições HTTP que são recebidas pela aplicação. Por exemplo, verificar se um determinado usuário que enviou uma requisição está ou não logado.
Http\Requests	Permite receber uma instância de uma determinada requisição, e sua validação, de uma forma simples através de injeção de dependência

2.3.2. database

factories	
migrations	
seeds	

2.3.3. resources

assets	
lang	
views	

2.3.4. resources\views

auth	
layouts	
“message”	

2.3.5. storage

app	
framework	
logs	

2.4. Configurando a Conexão com o Banco de Dados

Dentro de “`.env`”:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=chat
DB_USERNAME=root
DB_PASSWORD=
```

Obs: Opcional:
Configurar o Nome e Versão do Projeto dentro do package.json

```
{
   "name":"Chat",
   "version":"1.0",
   "private": true,
   "scripts": {
…...
```

2.5. Console

O Laravel possui um “Console” de comandos que permite executar diversas operações dentro do projeto. Para ter acesso as opções basta digitar o comando:

```
> php artisan
```

A documentação oficial pode ser encontrada em “https://laravel.com/docs/5.5/artisan”

2.6. Configurações da aplicação

2.6.1. Gerando uma chave

Essa chave é gerada automaticamente (não é necessário executar o comando abaixo). Ela serve como parte da segurança do projeto encriptando as sessões dos usuários por exemplo.

```
> php artisan key:generate
```

2.6.2. Colocando o Nome dos Pacotes

Dentro de cada arquivo PHP tem um namespace:

```
namespace App;
```

Se executarmos o comando abaixo, alteramos o pacote padrão de todas as nossas classes/funções:

```
> php artisan app:name Chat
```

O resultado fica:

```
namespace Chat;
```

Isso se refletirá nas classes já geradas e nas novas que serão construídas.

2.7. Subindo o Servidor

Para o nosso servidor rodar é bem simples, bastando executar na raiz do nosso projeto o seguinte comando:

```
> php artisan serve --port=80
```

Aqui já é possível abrir o navegador e acessar o site em “http://localhost/”

É importante lembrar que para podermos realizar o login devemos ainda preparar a base de dados aplicado as “`Migrations`”, que veremos logo em seguida.

Utilizando a opção HOST com 0.0.0.0 permitimos então o acesso a outros computadores da rede.

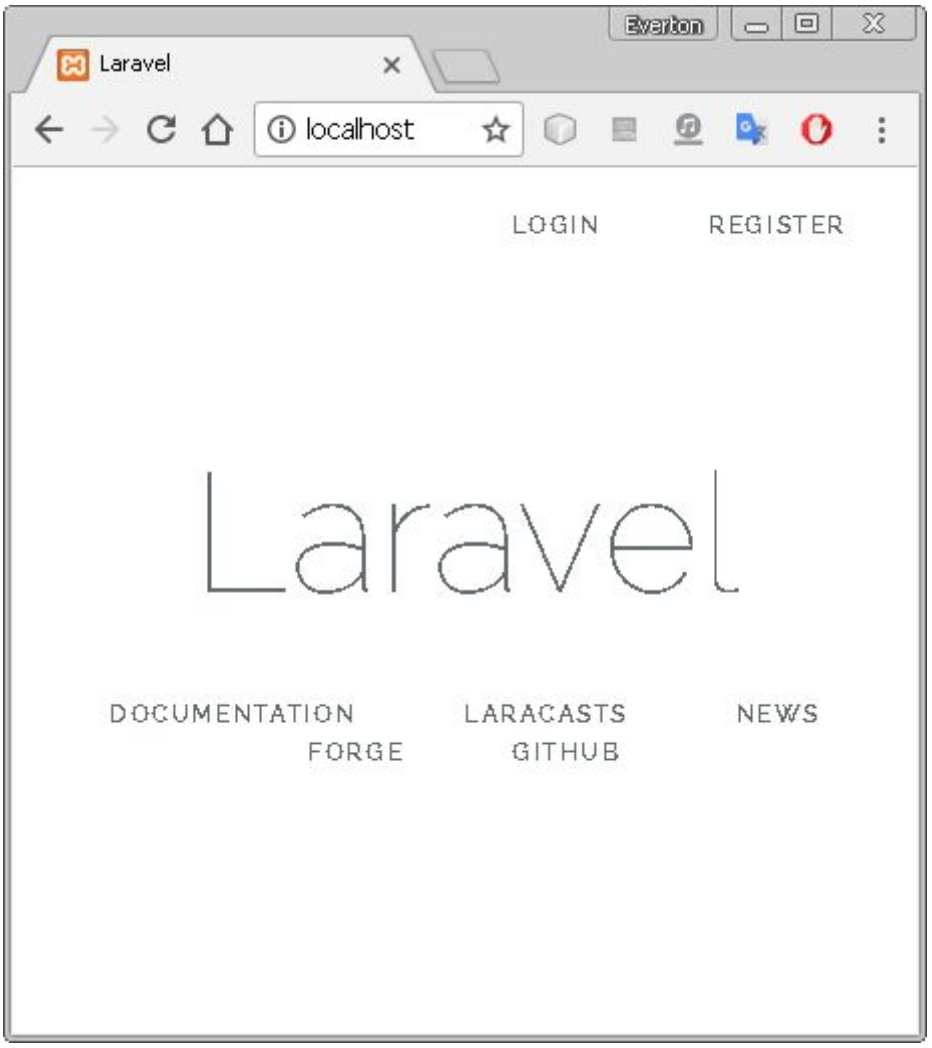
```
> php artisan serve --host=0.0.0.0 --port=80
```

2.8. Criando o sistema de autenticação

O Laravel possui um pacote básico de autenticação e controle de acesso bem fácil de ser adicionado a aplicação:

```
> php artisan make:auth
```

Automaticamente o todo o sistema de autenticação é criado, com rotas, views, controllers e tabelas.



Com esse comando “`Migration`”, “`Routes`”, “`Model`”, etc… são criados para gerenciar os usuários.

2.8.1. Arquivo de configuração

As configurações de nossa aplicação ficam em “`config\app.php`” ou no “.env”. É importante ressaltar que o arquivo “.env” não vai subir sua cópia para a produção. O que ocorre é que cada ambiente (Desenvolvimento, Teste, Homologação, Produção, etc..) terá o seu próprio arquivo.

2.8.1.1. Idioma

```
1. 'locale' => 'pt',
```

2.8.1.2. Nome da aplicação

```
'name' => 'Chat',
```

sempre que usarmos o:

```
{{ @config('app.name') }}
```

dentro de uma página teremos o nome do nosso projeto

3. Migrate: Criando o Banco de Dados

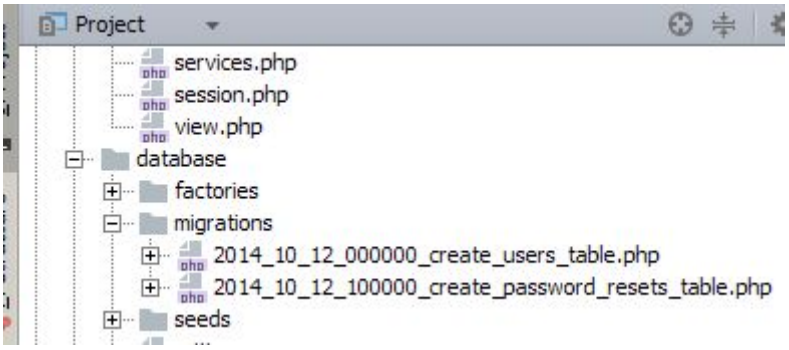
Os comando ”`Migrate`” permitem a manipulação do banco de dados criando, alterando e apagando tabelas.

migrate	Executa as migrações
migrate:fresh	Apaga todas as tabelas e executa novamente todas as migrações. Muito cuidado pois todos os dados do banco serão perdidos.
migrate:install	Cria o repositório de migrações
migrate:refresh	Reseta e executa novamente todas as migrações
migrate:reset	Desfaz todas as migrações
migrate:rollback	Desfaz a última migração
migrate:status	Mostra o status de cada migração

ATENÇÃO: É necessário realizar um ajuste dependendo da versão do banco que será utilizada no arquivo “`app\Providers\AppServiceProvider.php`”:

```
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;

class AppServiceProvider extends ServiceProvider
{
  public function boot()
  {
      Schema::defaultStringLength(191);
  }
```

As migrações ficam em uma pasta própria e são executadas em ordem alfabética. Quando criamos a autenticação o framework gerou automaticamente duas migrações que serão transformadas em tabelas



Então quando executamos o comando:

```
> php artisan migrate
```

O resultado deve ser a criação das tabelas no banco de dados:

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
```

Se checarmos no banco de dados deveremos ter uma estrutura semelhante a esta:

A tabela “`migrations`” é utilizada pelo nosso framework para controlar a sincronização das migrações com o banco.

Agora já é possível se cadastrar e realizar o login no site.

4. Migrations

---

154 down vote accepted

You can do it if you start from the model

```
php artisan make:model Todo -mcr
```

if you run `php artisan make:model --help` you can see all the available options

-m, --migration Create a new migration file for the model.

-c, --controller Create a new controller for the model.

-r, --resource Indicates if the generated controller should be a resource controller

Update

As mentioned in the comments by @arun in newer versions of laravel > 5.6 it is possible to run following command:

```
php artisan make:model Todo -a
```

-a, --all Generate a migration, factory, and resource controller for the model

---

As migrações são como controle de versão para o seu banco de dados, permitindo que sua equipe facilmente modifique e compartilhe o esquema do banco de dados do aplicativo. As migrações geralmente são combinadas com o construtor de esquema do Laravel para criar facilmente o esquema de banco de dados do seu aplicativo. Se você já teve que dizer a um colega de equipe para adicionar manualmente uma coluna ao seu esquema de banco de dados local, você enfrentou o problema que as migrações de banco de dados resolvem.

https://laravel.com/docs/5.5/migrations

São o começo da nossa modelagem e onde diremos quais tabelas serão criadas, a ordem da criação e a sua estrutura. O comando a baixo criar uma estrutura para a criação da tabela "`messages`".

```
> php artisan make:migration create_messages_table
```

Por padrão no Laravel utilizamos o nome do tabela no plural "`Messages`".

Um arquivo será gerado, com a data e mais um número garantindo que ele fique na ordem de criação. Se precisar alterar a ordem basta alterar o nome do arquivo.

O arquivo gerado com o comando:

```php
class CreateMessagesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('messages', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('messages');
    }
}
```

Existem duas funções:

1. **`Up`**: é chamada quando a migração é realizada e contém a estrutura da tabela
2. **`Down`**: indica o que deve ser feito quando um rollback é realizado

Dentro da função Up temos o "`Schema::create`" que é a criação da tabela efetivamente e onde definiremos a estrutura (colunas) da nossa tabela. Automaticamente são criados dois atributos:

1. **`id`**: é a chave primária da nossa tabela do tipo inteiro e auto incrementável
2. **`timestamps`**: cria dois campos na tabela:
    a. **`created_at`**: campo que armazena a data/hora de criação do registro.
    b. **`updated_at`**: campo que armazena a data/hora da última atualização.

Existem outras opções avançadas para o "`make:migration`" que serão aprofundadas no módulo avançado.

4.1. Adicionado Campos

Dentro da função "Up" a tabela será modificada para conter os campos e seus relacionamentos.

```
1. Schema::create('messages', function (Blueprint $table) {
2.     $table->increments('id');
3.     $table->integer('user_id')->unsigned()->index();
4.     $table->string('text');
5.     $table->timestamps();
6.     $table->foreign('user_id')->references('id')->on('users');
7. });
```

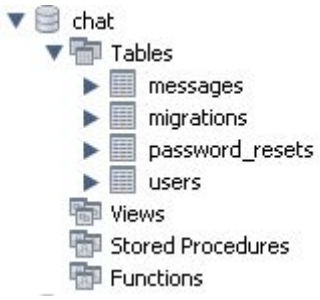
No exemplo acima foram realizadas três modificações:

1. **`$table->integer('user_id')->unsigned()->index()`**: adiciona um campo do tipo inteiro sem sinal, com o nome de "`user_id`" e cria um índice para ele
2. **`$table->string('text')`**: adiciona um campo do tipo string com o nome de "`text`"
3. **`$table->foreign('user_id')->references('id')->on('users')`**: cria uma chave estrangeira (Foreign Key) para o campo "`user_id`" referenciando o campo "`id`" da tabela "`users`"

```
> php artisan migrate
```

Obs: Verifique se o banco de dados está vazio

Depois do comando gerado podemos verificar a nossa tabela criada no banco:



Obs: Podemos inserir alguns dados manualmente em nossa tabela para podermos realizar a consulta dos dados:

```
1.  INSERT INTO messages(user_id, text, created_at, updated_at)
2.  VALUES(1, 'Mensagem do Chat', '2018-01-05 22:21:22', '2018-01-05 22:21:22');
```

Lembre-se de adicionar um usuário antes.

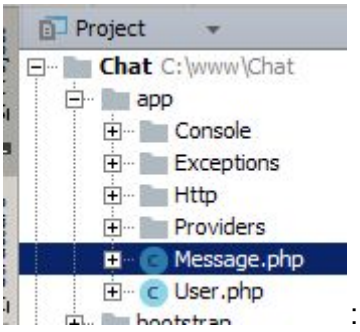
5. CRUD Simples

5.1. Model - As Mensagens

O Modelo define como iremos manipular nossos dados. Isto é feito de forma automática para as principais funções do CRUD não necessitando que o desenvolvedor escreva o código aumentando a produtividade.

```
> php artisan make:model Message
```

Com esse comando um novo modelo será criado vazio:



O código:

```
1.  class Message extends Model
2.  {
3.      //
4.  }
```

Por padrão no Laravel utilizamos o nome do modelo no singular.

Existem opções para a criação do modelo já criando e vinculando uma Migration e um Controller como será visto no capítulo avançado.

Dentro da nossa classe devemos colocar um atributo indicando quais os campos podem ser preenchidos. No nosso caso será a indicação do "`user_id`" e o "`text`": Este atributo é o "`fillable`" que indica quais os dados serão acessíveis:

```
1.     protected $fillable = ['user_id', 'text'];
```

É possível também utilizar o inverso com:

```
1. protected $guarded = [];
```

Neste caso, com um array vazio, significa que todos os campos podem ser preenchidos.

Também adicionaremos uma ligação com o modelo "`User`":

```
1. public function user(){
2.     return $this->belongsTo('\Chat\User');
3. }
```

Obs: "belongs to" significa "pertence a"

Dentro do modelo "`User`" também irá existir uma ligação para as mensagens:

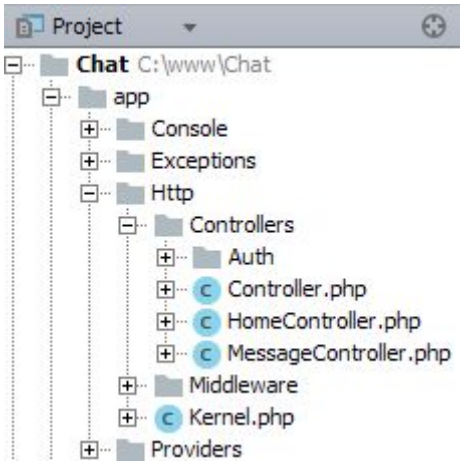
```
1. public function messages(){
2.     return $this->hasMany('\Chat\Message');
3. }
```

Obs: "has many" significa "tem muitos"

5.2. Controller - O Início

```
> php artisan make:controller MessageController --model=Message
```

Quando indicamos o modelo as funções do CRUD são automaticamente geradas.

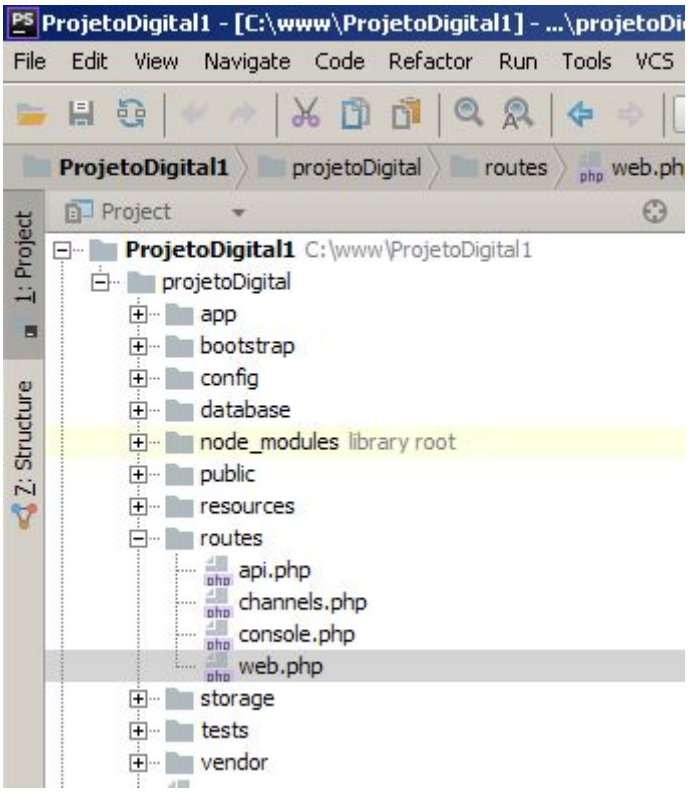

Project
Chat C:\www\Chat
app
Console
Exceptions
Http
Controllers
Auth
Controller.php
HomeController.php
MessageController.php
Middleware
Kernel.php
Providers

```php
class MessageController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param  \Chat\Message  $message
     * @return \Illuminate\Http\Response
     */
    public function show(Message $message)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  \Chat\Message  $message
     * @return \Illuminate\Http\Response
     */
    public function edit(Message $message)
    {
        //
```

```
54.    }
55.
56.    /**
57.     * Update the specified resource in storage.
58.     *
59.     * @param  \Illuminate\Http\Request  $request
60.     * @param  \Chat\Message  $message
61.     * @return \Illuminate\Http\Response
62.     */
63.    public function update(Request $request, Message $message)
64.    {
65.        //
66.    }
67.
68.    /**
69.     * Remove the specified resource from storage.
70.     *
71.     * @param  \Chat\Message  $message
72.     * @return \Illuminate\Http\Response
73.     */
74.    public function destroy(Message $message)
75.    {
76.        //
77.    }
78. }
```

5.3. Route - Rotas de Mapeamento Padrão

Rotas definem uma forma simples de indicarmos o caminho que nossa aplicação irá tomar de acordo com um recurso que será acessado. Ele funciona como REST, implementando todas as funções do CRUD (Create, Read, Update e Delete), porém não fica limitado apenas isso, disponibilizando uma infinidade de combinações entre ações e perfis de usuários, entre outros.



Uma rota simples:

```
1.  Route::get('/teste', function () {
2.      return '<html><body>Teste Simples</body></html>';
3.  });
```

Essa rota serve para o método `'GET'` acessando o recurso `'teste'`. O acesso é realizado diretamente na URL:

http://localhost/teste

Para o CRUD:

```
1.  Route::resource('message', 'MessageController');
```

Tabela com as operações/URI e as ações que serão automaticamente chamadas:

Verb	URI	Action	Route Name
GET	/message	index	message.index

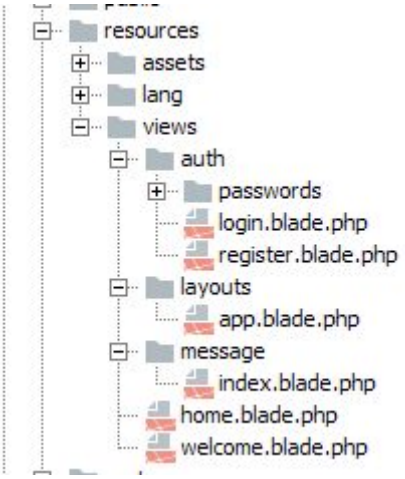
GET	/message/create	create	message.create
POST	/message	store	message.store
GET	/message/{message}	show	message.show
GET	/message/{message}/edit	edit	message.edit
PUT/PATCH	/message/{message}	update	message.update
DELETE	/message/{message}	destroy	message.destroy

5.4. View - Listando as Mensagens

O Laravel utiliza por padrão o BootStrap (http://getbootstrap.com.br/) como framework Frontend além de um esquema de templates chamado Blade, como já foi mencionado
https://laravel.com/docs/5.5/views
.

A combinação destes elementos nos permite construir interfaces muito poderosas, responsivas se adaptando a qualquer dispositivo, muito leves além de herança componentes reusáveis, permitindo escrever menos código mantendo uma alta manutenibilidade.

Para uma exibição simples das telas que operam o CRUD sobre um modelo é possível organizar criando uma sub-pasta dentro de "`views`"



Dentro da pasta pode ser criado os arquivos que serão a interface desse modelo. É interessante manter um padrão de nomes que vem do "`Controller`", apesar de não ser obrigatório acaba simplificando.

Para a listagem pode ser criado o "`index.blade.php`":

```
1.  @extends('layouts.app')
2.
3.  @section('content')
```

```
4.     <div class="container">
5.         <h1 class="page-header">Mensagens</h1>
6.         <div class="table-responsive">
7.             <table class="table table-striped table-bordered table-hover">
8.                 <thead>
9.                 <tr>
10.                    <th>#</th>
11.                    <th>Usuário</th>
12.                    <th>Mensagem</th>
13.                </tr>
14.                </thead>
15.                <tbody>
16.                @foreach($messages as $message)
17.                    <tr>
18.                        <th>{{ $message->id }}</th>
19.                        <td>{{ $message->user_id }}</td>
20.                        <td>{{ $message->text }}</td>
21.                    </tr>
22.                @endforeach
23.                </tbody>
24.            </table>
25.        </div>
26.    </div>
27.
28. @endsection
```

5.4.1. Controller: Dados do Model para a View

Um framework guia o desenvolvimento por boas práticas, mas não necessariamente obriga o desenvolvedor a utilizá-las.

Uma classe controladora organiza o código ajudando na manutenção, realizando a conexão entre modelo e a visão:

```
1. public function index()
2. {
3.     return view('message.index', ['messages' => Message::all()]);
4. }
```

No comando acima é criada uma "`view`" que recebe dois parâmetros:

1. O primeiro faz referência a página "`index`" dentro da pasta "`message`". Podemos perceber que o "`blade.php`" do nome do arquivo foi ignorado, pois ele é colocado por padrão pelo nosso framework.
2. O segundo é um array associativo, contendo uma chave e as informações que ela contém e que serão enviadas para a página. A chave vira uma variável de mesmo nome na nossa "`view`".

A partir deste momento já teremos nossos dados exibidos “http://localhost/message”.

5.4.2. Menu

```
<ul class="dropdown-menu">
    <li>
        <a href="{{ route('message.index') }}">
            Mensagens
        </a>

        <a href="{{ route('logout') }}"
            onclick="event.preventDefault();
                     document.getElementById('logout-form').submit();">
...
```

5.4.3. Ordem de Exibição

Existem diversos comando que podemos aplicar ao nosso modelo, e vermos eles no decorrer do curso. Para o momento utilizaremos a ordenação dos dados pelo “`id`”:

```
public function index()
{
    $messages = Message::all()->sortByDesc("id");

    return view('message.index', ['messages' => $messages]);
}
```

É possível perceber como se pode manipular os dados aplicando diversas regras antes de enviar para o formulário.

5.5. A Tela para Adicionar Mensagem

Na tela de “`index`” podemos colocar um botão para que o navegador entre na tela de envio de mensagens:

```
<p>
    <a class="btn btn-info" href="{{route('message.create')}}">Enviar
Mensagem</a>
</p>
```

Pode se observar a ligação do “`route('message.create')`” com o código das rotas na sessão anterior.

Quando o botão é acionado somos redirecionados para "http://localhost/message/create" pois dentro de nosso controlador nem uma ação foi definida:

Instanciando a View de criação de mensagens:

```
1.  public function create()
2.  {
3.      return view('message.create');
4.  }
```

Desta forma a página "create.blade.php" dentro da pasta "message" será exibida:

```
1.  @extends('layouts.app')
2.
3.  @section('content')
4.      <div class="container">
5.          <h1 class="page-header">Mensagens</h1>
6.          <div class="table-responsive">
7.              <form method="post" action="{{ route('message.store') }}">
8.                  {{ csrf_field() }}
9.                  <div class="form-group">
10.                     <label for="text">Mensagem</label>
11.                     <input type="text" name="text" class="form-control"
    placeholder="Texto">
12.                 </div>
13.                 <button class="btn btn-info">Enviar</button>
14.             </form>
15.         </div>
16.     </div>
17. @endsection
```

obs: Dentro do formulário Token e Campo de Segurança

```
1.      {{ csrf_token() }} // apenas o token
2.      {{ csrf_field() }} // o campo inteiro
```

Ainda dentro do formulário é possível adicionar um campo com o Id do usuário ou ou este valor pode ser lido diretamente de dentro da sessão. Se for

```
1.  <input type="hidden" name="user_id" value="{{  Auth::id() }}"
    readonly="readonly">
```

Recebendo e armazenando os dados:

```
1.  public function store(Request $request)
2.  {
3.      \Chat\Message::create($request->all());
4.
5.      return redirect()->route('message.index');
6.  }
```

Ou lendo o Id do usuário da sessão armazenada o servidor, e ignorando o dado que vem da Web:

```
1.  public function store(Request $request)
2.  {
3.      $message = new \Chat\Message($request->all());
4.
5.      $message->user_id = \Auth::user()->id;
6.
7.      $message->save();
8.
9.      return redirect()->route('message.index');
10. }
```

5.5.1. Segurança

|Cross-site request forgery. O CSRF (do inglês Cross-site request forgery- Falsificação de solicitações entre sites) é um tipo de ataque informático malicioso a um website no qual comandos não autorizados são transmitidos através de um utilizador em quem o website confia.

5.6. Excluindo uma Mensagem

A exclusão de dados é bastante simples, bastando colocar uma referência para o método "`delete`" do controlador, e lá dentro então realizar a operação.

Na nossa listagem, podemos adicionar

```
1.  @foreach($messages as $message)
2.      <tr>
3.          <th>{{ $message->id }}</th>
4.          <td>{{ $message->user_id }}</td>
5.          <td>{{ $message->text }}</td>
6.          <td>
7.              <form action="{{ route('message.destroy', $message->id) }}"
    method="POST">
8.                  {{ csrf_field() }}
9.                  <input type="hidden" name="_method" value="DELETE">
10.                 <button class="btn btn-danger" onclick="return
    confirm('Deseja mesmo excluir?')">
11.                     Apagar
12.                 </button>
13.             </form>
14.         </td>
```

```
    </tr>
@endforeach
```

Note que é adicionado um campo escondido "`_method`" do tipo "**`DELETE`**". Este campo que fará a ligação dentro da nossa rota conforme a tabela já apresentada.

No Controller basta a chamada do método "`destroy`" e depois realizar um redirecionamento para a página desejada:

```
    public function destroy(Message $message)
    {
        $message->delete();

        return $this->index(); // difer. do formato do create mas com = resulta
    }
```

Podemos também impedir que um usuário apague as mensagens do outro. Para escondermos o botão na interface podemos utilizar um "`@if/@endif`" testando se o usuário logado é o mesmo da mensagem

```
<td>
    @if($message->user_id == \Auth::user()->id)
    <form action="{{ route('message.destroy', $message->id) }}" method="POST">
        {{ csrf_field() }}
        <input type="hidden" name="_method" value="DELETE">
        <button class="btn btn-danger" onclick="return confirm('Deseja mesmo excluir?')">
            Apagar
        </button>
    </form>
    @endif
</td>
```

Por medida de segurança, podemos colocar a validação também dentro do "`Controller`":

```
public function destroy(Message $message)
{
    if($message->user_id == \Auth::user()->id) {
        $message->delete();
    } else {
        throw new Exception('Usuário não tem permissões para apagar essa mensagem!');
    }

    return $this->index();
```

```
}
```

Está é uma segurança bastante simples, nos próximos tópicos será apresentada uma solução de segurança mais completa.

Obs: Para que não seja apresentado o código da aplicação em produção devemos desabilitar o debug dentro o "`.env`": "`APP_DEBUG=true`".

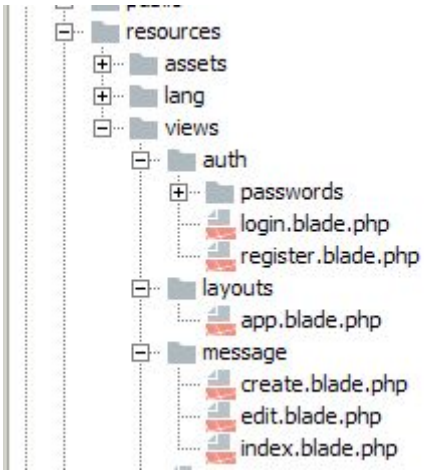
5.7. Alterando as Mensagens

Claro que alterar uma mensagem que já está no meio de uma conversa não tem muito sentido podendo transformar o texto sem sentido, porém é um exercício isso será ignorado.

Para realizar a alteração deve ser adicionado um botão na listagem, aproveitando o "`@if`" do botão "`delete`":

```
<td>
    @if($message->user_id == \Auth::user()->id)
        <form action="{{ route('message.destroy', $message->id) }}" method="POST">
            {{ csrf_field() }}
            <input type="hidden" name="_method" value="DELETE">
            <button class="btn btn-danger" onclick="return confirm('Deseja mesmo excluir?')">
                Apagar
            </button>
        </form>
        <form action="{{ route('message.edit', $message->id) }}" method="POST">

            {{ csrf_field() }}
            <input type="hidden" name="_method" value="GET">
            <button class="btn btn-info">
                Editar
            </button>
        </form>
    @endif
</td>
```

Criação da página "`edit.blade.php`":



A página de edição:

```
@extends('layouts.app')

@section('content')
    <div class="container">
        <h1 class="page-header">Mensagens</h1>
        <div class="table-responsive">
            <form method="post" action="{{ route('message.update', $message) }}">
                {{ csrf_field() }}

                {{ method_field('PUT') }}
                <div class="form-group">
                    <label for="text">Mensagem</label>
                    <input type="text" name="text" class="form-control" placeholder="Texto"   value="{{ $message->text }}">
                </div>
                <button class="btn btn-info">Enviar</button>
            </form>
        </div>
    </div>
@endsection
```

No método edit do Controller:

```
    public function edit(Message $message)
    {
        if( \Auth::user()->id == $message->user_id) {
            return view('message.edit', array('message' => $message));
        } else {
            throw new \Exception('Você não tem acesso!');
        }
    }

```

No método update do Controller:

```
1.  public function update(Request $request, Message $message)
2.  {
3.      if($message->user_id == \Auth::user()->id) {
4.          $message->update($request->all());
5.          return $this->index();
6.      } else {
7.          throw new \Exception('Usuário não tem permissões para editar!');
8.      }
9.  }
```

5.8. Apresentando dados da tabela relacionada

O relacionamento é criado através de uma função:

```
1.  class Message extends Model
2.  {
3.      protected $fillable = ['user_id', 'text'];
4.
5.      public function user(){
6.          return $this->belongsTo('\Chat\User');
7.      }
8.  }
```

Ela é utilizada diretamente como uma propriedade do objeto.

```
1.  @foreach($messages as $message)
2.      <tr>
3.          <th>{{ $message->id }}</th>
4.          <td>{{ $message->user->name }}</td>
5.          <td>{{ $message->text }}</td>
6.          <td>
```

5.9. Detalhes da mensagem

O método “show” dentro do controlador serve para mostrar detalhes de um registro. O link na listagem:

```
1.  <tr>
```

```
    <th><a href="{{ route('message.show', $message->id  ) }}">{{ $message->id }}</a></th>
    <td>{{ $message->user->name }}</td>
    <td>{{ $message->text }}</td>
```

No Controller é realizado o carregamento da view:

```
    public function show(Message $message)
    {
        return view('message.show', ['message' => $message]);
    }
```

A página de exibição “`show.blade.php`”:

```
@extends('layouts.app')

@section('content')
    <div class="container">
        <h1 class="page-header">Mensagens</h1>
        <div class="panel-body">
            <p><b>Id: </b>{{ $message->id }}</p>
            <p><b>Usuário: </b>{{ $message->user->name }}</p>
            <p><b>Texto: </b>{{ $message->text }}</p>
            <p><b>Criada: </b>{{ $message->created_at }}</p>
            <p><b>Alterada: </b>{{ $message->updated_at  }}</p>
        </div>
    </div>
@endsection
```

6. Seeders: Preenchendo com dados aleatórios

Seeders permitem “semear” nossa base de dados com registros aleatórios formando uma massa de testes para a nossa aplicação. Existem vários comandos que podem ser executados para a criação de nomes, emails, endereços, telefones, etc…

A documentação completa pode ser encontrada em:

1. https://laravel.com/docs/5.5/seeding
2. https://laravel.com/docs/5.5/database-testing#writing-factories

6.1. DatabaseSeeder

Podemos gerar os dados dentro do “`DatabaseSeeder`” ou criar uma “`Seeder`” para específica para cada modelo. Utilizando-se de forma genérica:

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

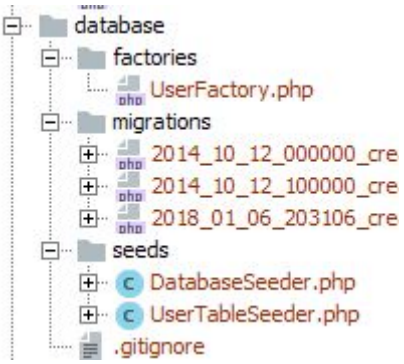
```
> php artisan make:seeder UserTableSeeder
```

```
> php artisan make:seeder MessageTableSeeder
```

Também dentro do “`DatabaseSeeder`” é possível chamar as outras classes para semear a base:

```php
    public function run()
    {
        $this->call(UserTableSeeder::class);
        $this->call(MessageTableSeeder::class);
    }
```

6.2. Factorys



Quando o projeto é criado automaticamente uma fábrica de "`User`" já é criada:

```
1.  $factory->define(Chat\User::class, function (Faker $faker) {
2.      return [
3.          'name' => $faker->name,
4.          'email' => $faker->unique()->safeEmail,
5.          'password' =>
    '$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
6.          'remember_token' => str_random(10),
7.      ];
8.  });
```

Para criar uma segunda fábrica:

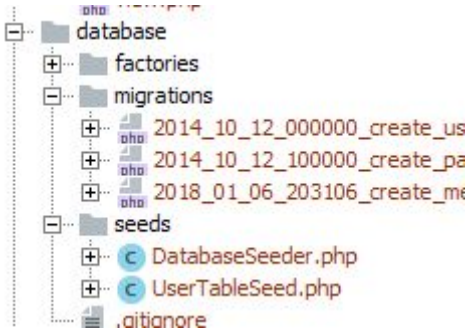
```
> php artisan make:factory MessageFactory
```

```
1.  <?php
2.
3.  use Faker\Generator as Faker;
4.
5.  $factory->define('Chat\Message', function (Faker $faker) {
6.      return [
7.          'user_id' => \Chat\User::all()->random()->id,
8.          'text' => $faker->sentence(10),
9.      ];
10. });
```

6.3. Table Seeder

Para utilizar a Factory, primeiramente o arquivo é gerado com o comando abaixo:

```
> php artisan make:seed UserTableSeeder
```




```php
class UserTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory('Chat\User', 50)->create();
    }
}
```

```
> php artisan make:seed MessageTableSeeder
```

```php
class MessageTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory('Chat\Message', 500)->create();
    }
}
```

6.4. Populando as tabelas

Para semear a base utiliza-se o comando abaixo que por padrão irá chamar o "`DatabaseSeeder`":

```
> php artisan db:seed
```

É possível também executar apenas para um modelo específico:

```
> php artisan db:seed --class=UserTableSeeder
```

7. Melhorias na interface

7.1. Paginação

Na seleção dos dados é indicado a quantidade de registros por página:

```php
public function index()
{
    $messages = Message::orderBy('id', 'desc')->paginate(15);

    return view('message.index', ['messages' => $messages]);
}
```

Na página de listagem "`index.blade.php`" é adicionado o código:

```php
<div align="center">
    {!! $messages->links() !!}
</div>
```

Obs: A marcação é diferente "**{!! !!}**" pois se trata de um código HTML gerado pelo framework.

7.2. BreadCrumbs

"Bread crumbs" significa literalmente "migalhas de pão", uma trilha por onde o usuário entrou. No comércio eletrônico é bastante comum o uso para indicar categorias e subcategorias de produtos.

O Laravel possui alguns plugins para realizar está tarefa, ou pode ser criado também manualmente.

Ná página de envio (criação) de mensagens, o resultado ficaria:

```
1.  <ol class="breadcrumb">
2.      <li class="breadcrumb-item"><a href="{{ route('message.index')
    }}">Mensagens</a></li>
3.      <li class="breadcrumb-item active">Enviar</li>
4.  </ol>
```

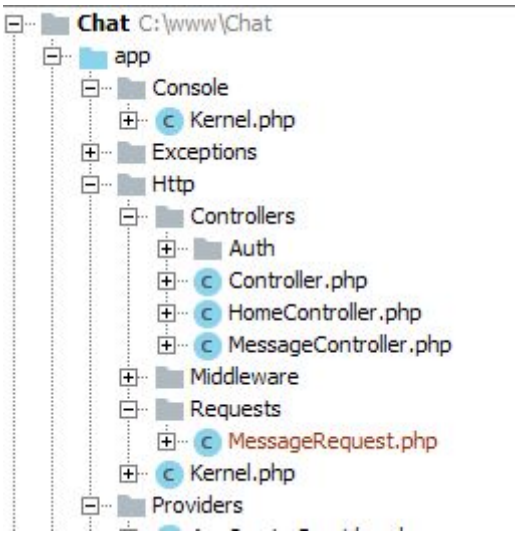
Exercício:

1. Crie as trilhas de navegação (Breadcrumb) para as demais páginas

8. Validação de Formulários - Requisições

A validação pode ser realizada diretamente dentro do controlador ou pode ser implementada dentro de uma classe específica de requisição:

```
> php artisan make:request MessageRequest
```




```
1.  <?php
2.
3.  namespace Chat\Http\Requests;
4.
5.  use Illuminate\Foundation\Http\FormRequest;
6.
7.  class MessageRequest extends FormRequest
8.  {
9.      /**
10.      * Determine if the user is authorized to make this request.
11.      *
```

```
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

Tem dois métodos principais:

1. authorize: determina quais os papéis tem acesso, por exemplo: Gerente, Cliente, Diretor,, etc.
2. rules: determina as regras que serão aplicadas

Primeiramente devemos autorizar o acesso, então alteramos para:

```
public function authorize()
{
    return true;
}
```

Nas regras:

```
public function rules()
{
    return [
        'text' => 'required|max:150'
    ];
}
```

Também podemos colocar um método com mensagens personalizadas, sobrescrevendo o método herdado:

```
public function messages()
{
    return [
        'text.required' => 'Obrigatório!',
```

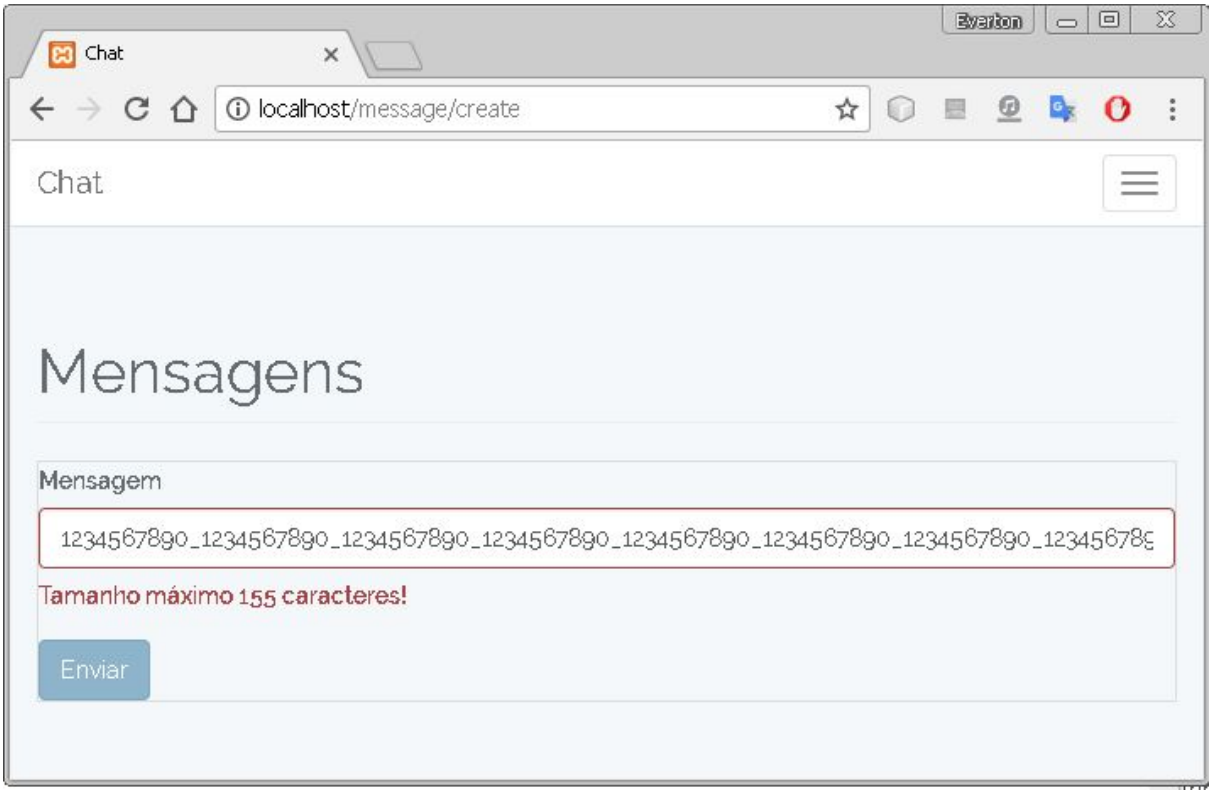
```
        'text.max' => 'Tamanho máximo 155 caracteres!'
    ];
}
```

Dentro do "`Controller`", apenas modificamos a Classe do "`Request`":

```
public function store(\Chat\Http\Requests\MessageRequest $request)
```

Alterando View para mostrar as mensagens de erro:

```
<div class="form-group {{ $errors->has('text') ? 'has-error' : '' }}">
    <label for="text">Mensagem</label>
    <input type="text" name="text" class="form-control" placeholder="Texto">
    @if($errors->has('text'))
        <span class="help-block">
            <strong>{{ $errors->first('text') }}</strong>
        </span>
    @endif
</div>
```



Na documentação do Laravel é possível encontrar as várias formas de realizar a validação do lado do servidor e também os tipos de validações que já estão implementados dentro do framework:

Available Validation Rules

Below is a list of all available validation rules and their function:

Accepted	Digits Between	Not In
Active URL	Dimensions (Image Files)	Numeric
After (Date)	Distinct	Present
After Or Equal (Date)	E-Mail	Regular Expression
Alpha	Exists (Database)	Required
Alpha Dash	File	Required If
Alpha Numeric	Filled	Required Unless
Array	Image (File)	Required With
Before (Date)	In	Required With All
Before Or Equal (Date)	In Array	Required Without
Between	Integer	Required Without All
Boolean	IP Address	Same
Confirmed	JSON	Size
Date	Max	String
Date Equals	MIME Types	Timezone
Date Format	MIME Type By File Extension	Unique (Database)
Different	Min	URL
Digits	Nullable	

https://laravel.com/docs/5.5/validation

8.1. Avisos

No template principal pode ser criado um local para que se possa apresentar mensagens temporárias para o usuário:

```
1.  </nav>
2.
3.      @if(Session::has('flash_message'))
4.          <div class="container">
5.              <div class="row">
6.                  <div class="col-md-8 col-md-offset-1">
7.                      <div align="center" class="alert {{
    Session::get('flash_message')['class'] }}">
8.                          {{ Session::get('flash_message')['message'] }}
```

```
                    </div>
                </div>
            </div>
        </div>
    @endif

    @yield('content')
</div>
```

No Código, por exemplo no controlador envio de uma nova mensagem:

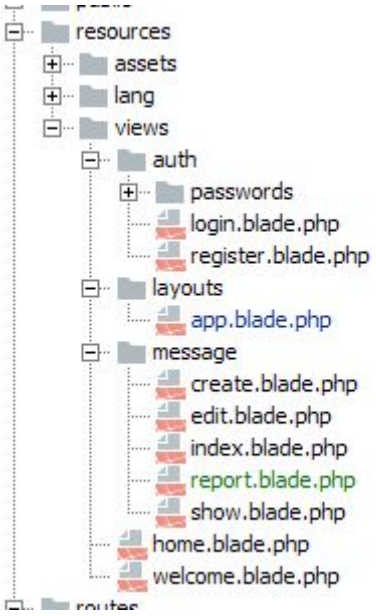
```
public function store(\Chat\Http\Requests\MessageRequest $request)
{
    $message = new \Chat\Message($request->all());
    $message->user_id = \Auth::user()->id;
    $message->save();

    \Session::flash('flash_message', [
        'message'=>'Mensagem enviada com sucesso!',
        'class'=>'alert-success'
    ]);

    return redirect()->route('message.index', ['messages' =>
Message::all()]);
}
```

Exercício:

1. Crie avisos para a edição e exclusão de mensagens.

9. Consultas Avançadas

Cria-se uma tela para a exibição dos dados:




```
1.  @extends('layouts.app')
2.
3.  @section('content')
4.      <div class="container">
5.          <h1 class="page-header">Mensagens</h1>
6.          <div class="table-responsive">
7.              <ol class="breadcrumb">
8.                  <li class="breadcrumb-item"><a href="{{
    route('message.index') }}">Mensagens</a></li>
9.                  <li class="breadcrumb-item active">Relatório</li>
10.             </ol>
11.             <table class="table table-striped table-bordered table-hover">
12.                 <thead>
13.                 <tr>
14.                     <th>Usuario</th>
15.                     <th>Quantidade de Mensagens</th>
16.                 </tr>
17.                 </thead>
18.                 <tbody>
19.                 @foreach($rows as $row)
20.                     <tr>
21.                         <td>{{ $row->name }}</td>
22.                         <td>{{ $row->count }}</td>
23.                     </tr>
24.                 @endforeach
25.                 </tbody>
26.                 <div align="center">
27.                     {!! $rows->links() !!}
28.                 </div>
29.             </table>
30.         </div>
31.     </div>
32.
33. @endsection
```

Criamos uma entrada no menu:

```
<a class="dropdown-item" href="{{ route('message.report') }}">
    Relatório
</a>
```

Dentro do controlador cria-se um método para a busca dos dados e exibição para o usuário (é necessário também adicionar ao arquivo do controlador o ‘`use DB;`’ no início):

```
public function report()
{
   $rows = DB::table('messages')
        ->join('users', 'users.id', '=', 'messages.user_id')
        ->select('users.name', DB::raw('count(*) as count'))
        ->groupBy('users.name')
        ->paginate(15);

   return view('message.report', ['rows' => $rows]);
}
```

Raw significa “cru”, quer dizer que o Laravel não deve tentar transformar o comando passado pois este já está no formato correto.

O último passo é a definição da rota:

```
Route::get('message.report',[
    'as' => 'message.report',
    'uses' => 'MessageController@report'
]);
```

10. URLs e Rotas Avançadas

A ligação entre páginas dentro do Laravel pode ser realizada utilizado URL ou Rotas.

1. A URL é engessada, utiliza-se a unção”`url`” passando o caminho que será chamado. Esse caminho é o primeiro parâmetro passado no método a ser invocado na classe “Route” e é igual ao que será apresentado no navegador do usuário.

```
<li><a class="nav-link" href="{{ url('notasDoSemestreURL') }}">Notas do Semestre URL</a></li>
```

2. A rota “`route`” utiliza-se de um nome. Isso significa que se a URL por alguma razão for alterada no futuro, a função “`route`” irá refletir em toda a aplicação.

```
<li><a class="nav-link" href="{{ route('notasDoSemestreROUTE') }}">Notas do Semestre Route</a></li>
```

Intuitivamente o nome da rota pode ser definido quando no segundo argumento, dentro do array, envia-se a chave “`as`” ou chamando o método “`name('nome_da_rota')`” no rota criada.

```
function apresentarNotas() {
    return '<html><body>Página das Notas</body></html>';
};

$umaRota= Route::get('notasDoSemestreURL', function(){
    return apresentarNotas();
});

$umaRota->name('notasDoSemestreROUTE');

//dd($umaRota);
```

A alternativa então, seguindo o padrão utilizado no início da apostila, é arquivo controlador com o conteúdo:

```
namespace Chat\Http\Controllers;

class Notas {
  function apresentarNotas() {
      return '<html><body>Página das Notas</body></html>';
  }
}
```

E a rota sendo definida como:

```
$umaRota= Route::get('notasDoSemestreURL', [
  'as' => 'notasDoSemestreROUTE',
  'uses' => 'Notas@apresentarNotas'
]);
```

Obs: também é possível colocar a função dentro do array de parâmetros do segundo atributo:

```
$umaRota= Route::get('notasDoSemestreURL', [
```

```
    'as' => 'notasDoSemestreROUTE',
    function () {
        return '<html><body>Página das Notas</body></html>';
    }
]);
```

10.1. Listando as Rotas e URLs

Para obter uma listagem de todas as rotas definidas com suas configurações basta dar o comando:

```
> php artisan route:list
```