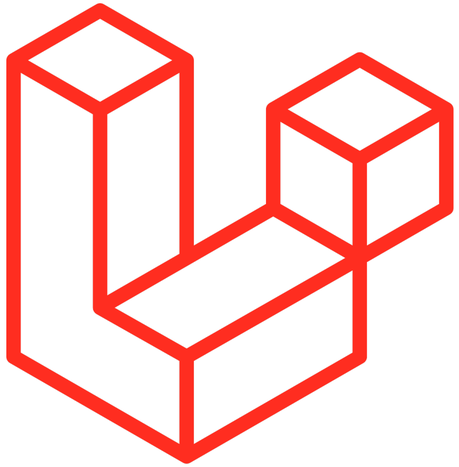
Programação para a WEB - servidor (server-side)

Sérgio da Silva Nogueira

PHP

Database: Migrations, Seeds & Factories



As migrations ou migrações definem a estrutura da base de dado. Funciona basicamente como uma documentação da linha histórica do crescimento da estrutura, criação das tabelas, ligações e etc.

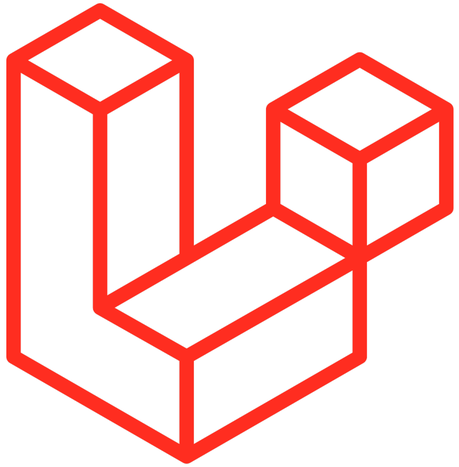
As migrations dentro do projeto podem ser encontradas na pasta database/migrations. Nesta pasta é possível encontrar alguns ficheiros de migração iniciais, como a migração para a tabela de user e a para a tabela de reset de senhas, e failed jobs.

Exemplo

20xx_xx_xx_000000_create_users_table.php:

PHP

Database: Migrations, Seeds & Factories

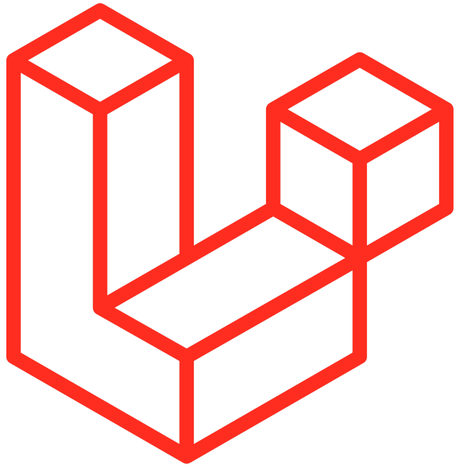


```php
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

PHP

Database: Migrations, Seeds & Factories



A classe de migração para a tabela users, inicialmente extende de uma classe base chamada Migration e traz a definição de dois métodos, o método up e o método down.

O método up é executado quando executarmos a migração. O método down contêm a definição do inverso do método up, no método down é definido a remoção do que foi aplicado no método up, isso nos permite voltarmos para estados anteriores pré-execução do último lote de migrações.

ELOQUENT
ORM (Object Relational Mapping) Conventions

Naming Controllers

Controllers should be in singular case, no spacing between words, and end with "Controller".

Also, each word should be capitalised (i.e. BlogController, not blogcontroller).

For example: `BlogController`, `AuthController`, `UserController`.

Bad examples: `UsersController` (because it is in plural), `Users` (because it is missing the Controller suffix).

ELOQUENT
ORM (Object Relational Mapping) Conventions

Naming database tables in Laravel

DB tables should be in lower case, with underscores to separate words (snake_case), and should be in plural form.

For example: `posts`, `project_tasks`, `uploaded_images`.

Bad examples: `all_posts`, `Posts`, `post`, `blogPosts`

ELOQUENT
ORM (Object Relational Mapping) Conventions

Pivot tables

Pivot tables should be all lower case, each model in alphabetical order, separated by an underscore (snake_case).

For example: `post_user`, `task_user` etc.

Bad examples: `users_posts`, `UsersPosts`.

PHP

ELOQUENT
ORM (Object Relational Mapping)
Conventions

Table columns names

Table column names should be in lower case, and snake_case (underscores between words). You shouldn't reference the table name.

For example: `post_body`, `id`, `created_at`.

Bad examples: `blog_post_created_at`, `forum_thread_title`, `threadTitle`.

ELOQUENT
ORM (Object Relational Mapping) Conventions

Primary Key

This should normally be `id`.

Foreign Keys

Foreign keys should be the model name (singular), with '_id' appended to it (assuming the PK in the other table is 'id').

For example: `comment_id`, `user_id`.

ELOQUENT ORM (Object Relational Mapping) Conventions

Variables

Normal variables should typically be in camelCase, with the first character lower case.

For example: `$users = ...`, `$bannedUsers = ...`.

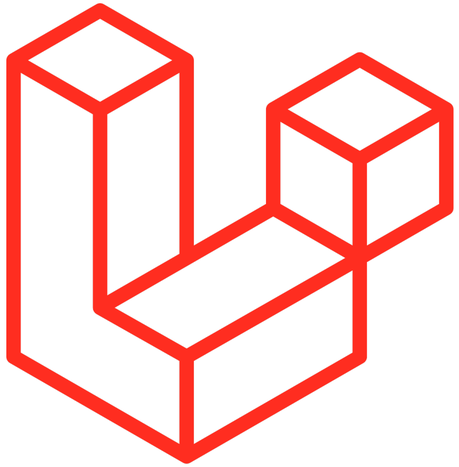
Bad examples: `$all_banned_users = ...`, `$Users=...`.

If the variable contains an array or collection of multiple items then the variable name should be in plural. Otherwise, it should be in singular form.

For example: `$users = User::all();` (as this will be a collection of multiple User objects), but `$user = User::first()` (as this is just one object)

PHP

Database: Migrations, Seeds & Factories



Documentação:

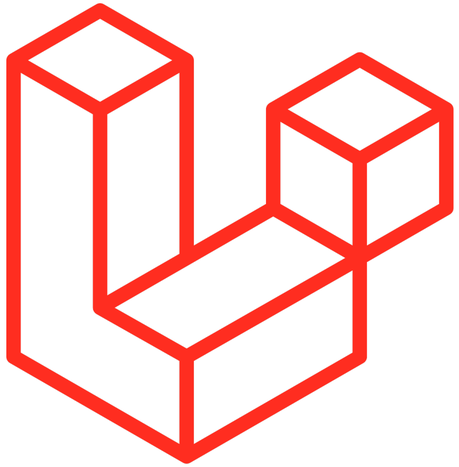
https://laravel.com/docs/7.x/migrations

Env:

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

P H P

Database: Migrations, Seeds & Factories

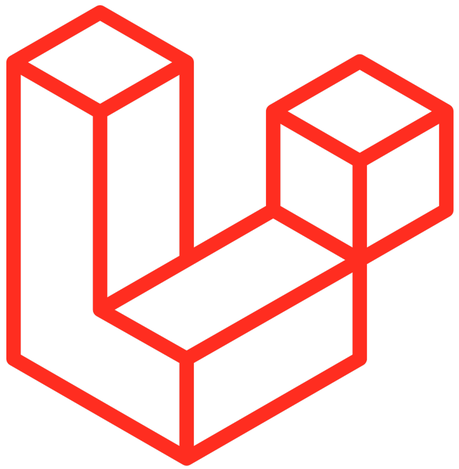


Executar migração na CMD:

php artisan migrate

```
blog: php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (0.14 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (0.08 second
s)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (0.07 seconds)
blog:
```

Database: Migrations



Executar migração na CMD:

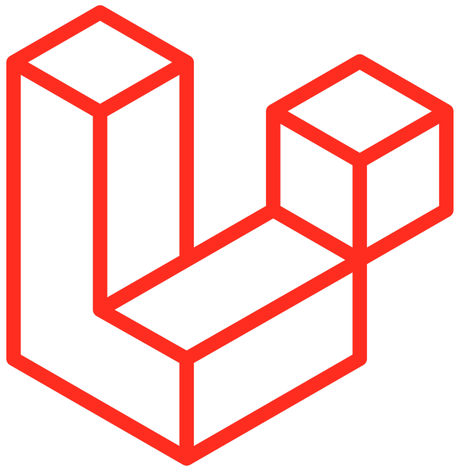
php artisan make:migration create_table_posts --create=posts

```
blog: php artisan make:migration create_table_post --create=posts
Created Migration: 2019_09_23_095103_create_table_post
blog:
```

O comando acima criará o primeiro ficheiro de migração dentro da pasta de migrações, chamado de 20xx_xx_xx_xxxxxx_create_table_post, o nome do ficheiro de migração respeita a data de criação mais o timestamp e o nome. Esta definição da data e timestamp permite o Laravel organizar a ordem das migrações.

O parâmetro --create=posts adicionará o código da classe Schema e o método create como podemos ver no conteúdo do ficheiro gerado:

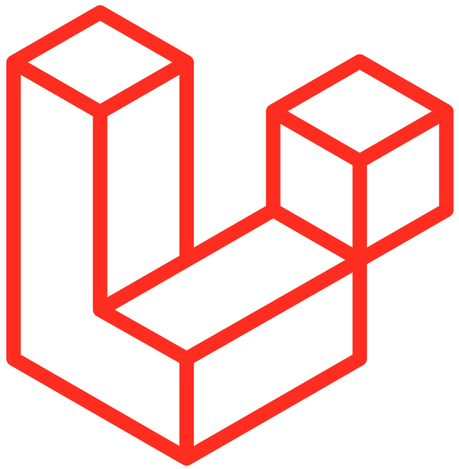
Database: Migrations



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

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

Database: Migrations

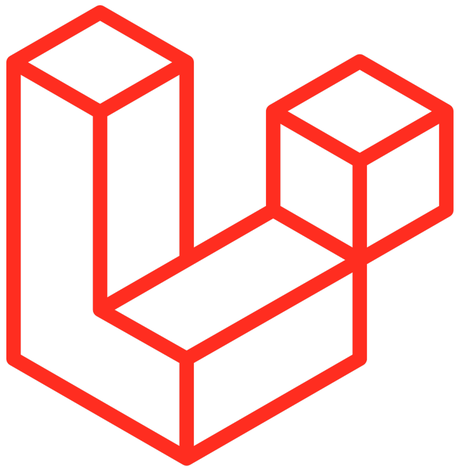


```php
class CreateTablePosts extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->string('description');
            $table->text('content');
            $table->string('slug');
            $table->boolean('is_active');
            $table->timestamps();
        });
    }

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

PHP

Database: Criar Relações em Migrations



Após a tabela posts criada, vamos definir constraints entre posts e users caracterizando assim a relação de posts e autor. A relação neste use case será de 1:N (Um para Muitos) onde 1 autor(user) poderá ter N(vários) posts e 1 post poderá ter ou pertencer a apenas um 1 autor/user.

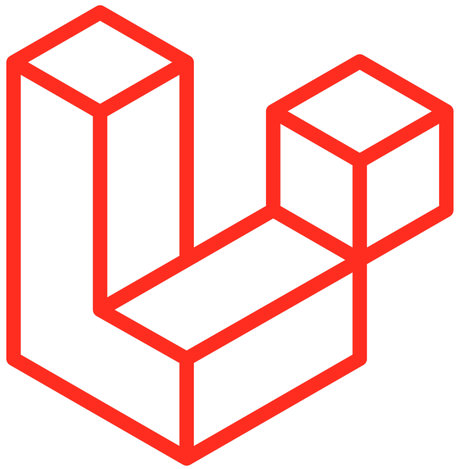
Como estamos a definir a nossa base de dados via migrations vamos definir esta relação alterando uma tabela já existente por meio de migrations, neste caso iremos alterar os posts para adicionarmos a referência para o user e ainda criar a chave estrangeira.

Para isso executamos o comando abaixo:

```
php artisan make:migration alter_table_posts_add_column_user_id --table=posts
```

PHP

Database: Criar Relações em Migrations

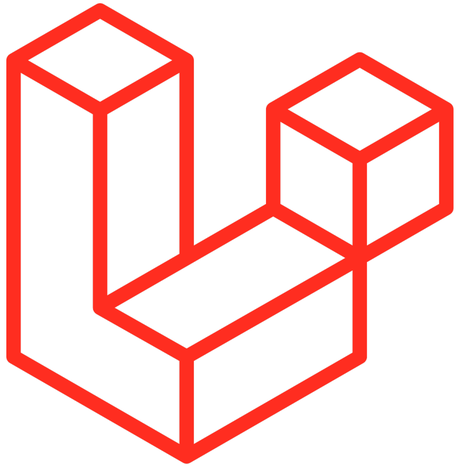


class AlterTablePostsAddColumnUserId extends Migration

```
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->unsignedBigInteger('user_id')->after('id');
            $table->foreign('user_id')->references('id')->on('users');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->dropForeign('posts_user_id_foreign');
            $table->dropColumn('user_id');
        });
    }
}
```

Database: Criar Relações em Migrations



CMD

php artisan migrate

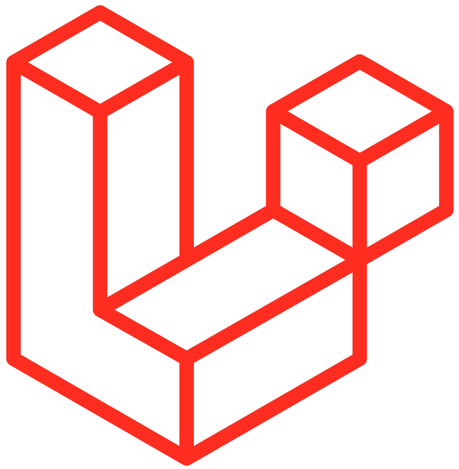
Executar a última migração criada, porque ainda não existe registo dela, na tabela de migrations.

```
blog: php artisan migrate
Migrating: 2019_09_23_135618_alter_table_posts_add_column_user_id
Migrated:  2019_09_23_135618_alter_table_posts_add_column_user_id (0.13 seconds)
blog:
```

Reset das migrations CMD:

php artisan migrate:fresh

Database: Seeds



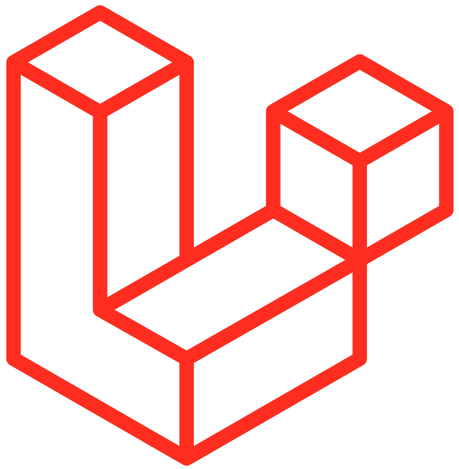
Quando estamos num contexto de desenvolvimento precisamos inserir registos na base de dados para testar a aplicação, os seeds permitem este processo. Podemos por meio das seeds alimentar a base de dados com informações para testar um CRUD.

Os seeds na framework estão localizados dentro de database/seeds, dentro desta pasta é possível encontrar o ficheiro DatabaseSeeder.php que é o responsável por executar todas as outras seeds que tivermos nesta pasta.

Documentação:

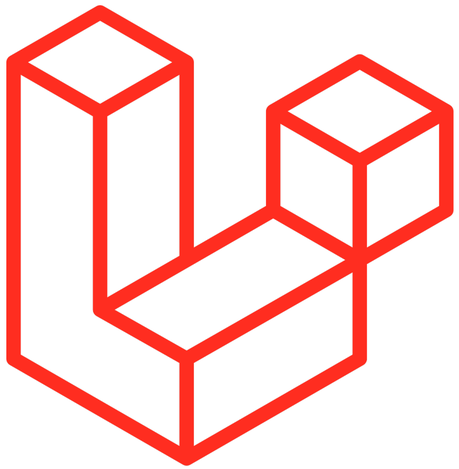
https://laravel.com/docs/7.x/seeding

Database: Seeds



```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UserSeeder::class);
    }
}
```

Database: Seeds



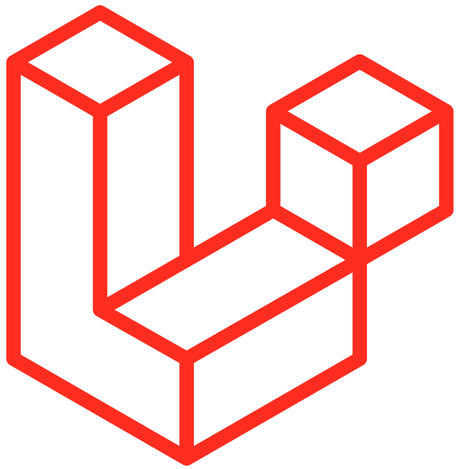
CMD:

php artisan make:seeder PostsTableSeeder

php artisan make:seeder UsersTableSeeder

```
blog: php artisan make:seeder UsersTableSeeder
Seeder created successfully.
blog: php artisan make:seeder PostsTableSeeder
Seeder created successfully.
blog:
```

Database: Seeds

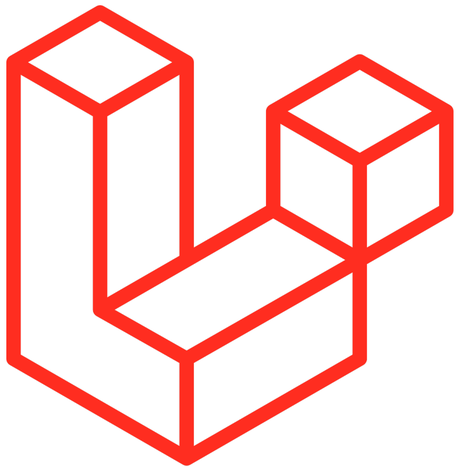


Exemplo:

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

    }
}
```

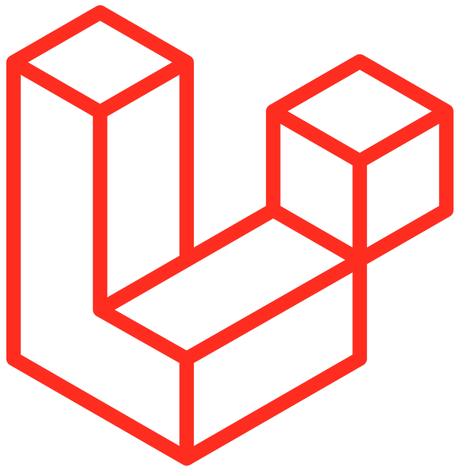
Database: Seeds



Exemplo:

```php
class PostsTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        \DB::table('posts')->insert([
            'title'       => 'First Title',
            'description' => 'First Post Description',
            'content'     => 'First Post Content',
            'is_active'   => 1,
            'slug'        => 'first-post',
            'user_id'     => 1,
        ]);
    }
}
```

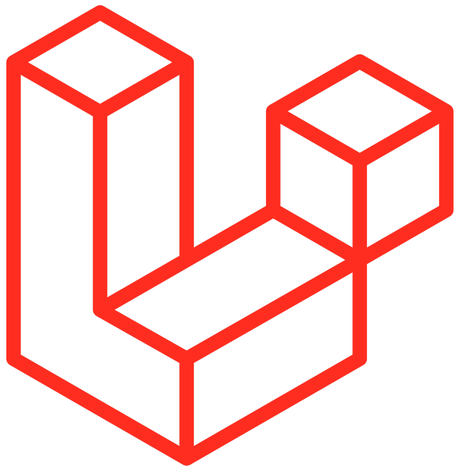
Database: Seeds



Executar:

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(UsersTableSeeder::class);
        $this->call(PostsTableSeeder::class);
    }
}
```

Database: Seeds

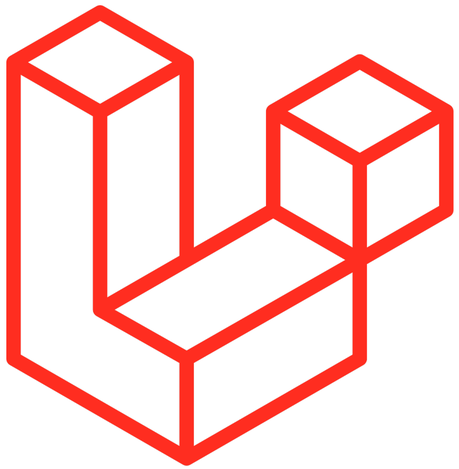


Run seed CMD:

php artisan db:seed

```
blog: php artisan db:seed
Seeding: UsersTableSeeder
Seeding: PostsTableSeeder
Database seeding completed successfully.
blog:
```

Database: Factories



Quando precisamos realizar a gerar muitos dados de forma mais automatizada podemos utilizar as Factories ou Model Factories.

As factories definem as regras, com base no model escolhido e conforme a quantidade especificada, para criação de dados fictícios nas tabelas. São chamados de factories ou fábricas por serem um esboço para criação de dados e junto com os seeds realizamos as gerações esperadas. Os ficheiros de factories encontram-se na pasta database/factories.

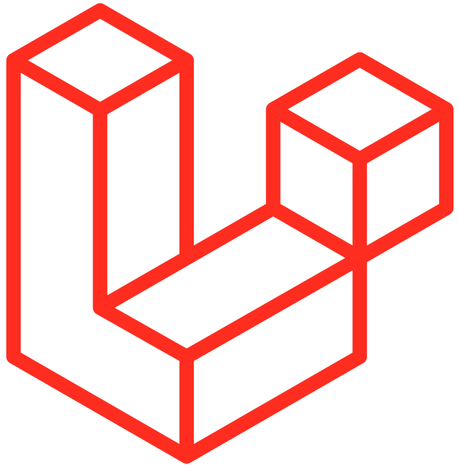
Database: Factories

Nesta pasta já temos um ficheiro de factory, UserFactory.php.

```
$factory->define(User::class, function (Faker $faker) {
    return [
        'name'              => $faker->name,
        'email'             => $faker->unique()->safeEmail,
        'email_verified_at' => now(),
        'password'          => '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi',
        'remember_token'    => Str::random(10),
    ];
});
```

Documentação do Faker:
https://github.com/fzaninotto/Faker#formatters

Database: Factories



Vamos executar a nossa inserção de dados utilizando o modelo definido pela factory. Para utilizar a factory combinamos a geração dos dados, utilizando a factory, junto com a execução via seeds. Para isso alteramos o nosso **UsersTableSeeder** para utilizar as factories.

Comente o conteúdo do método run do UsersTableSeeder e adicione a chamada abaixo:

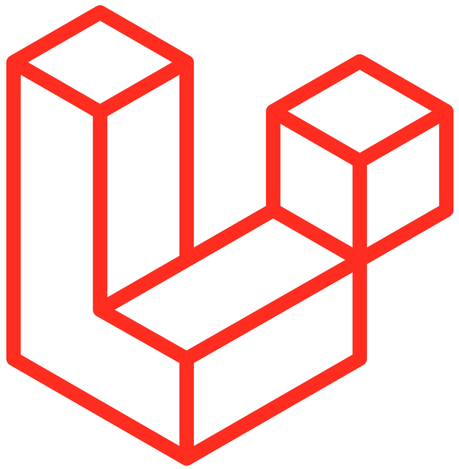
```
factory(\App\User::class,50)->create();
```

E execute a seed pela CMD:

php artisan db:seed --class=UsersTableSeeder

php artisan db:seed

Database: Factories



Para criar uma factory utilizamos o seguinte comando na CMD:

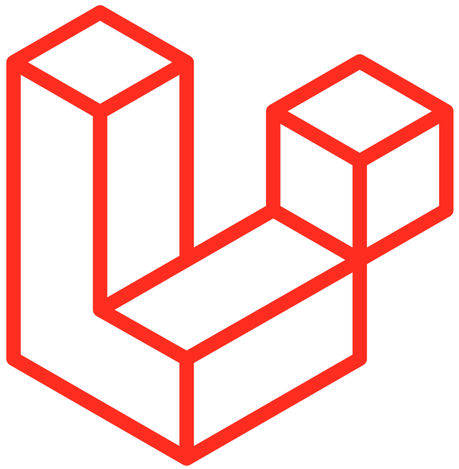
php artisan make:factory PostFactory

```
$factory->define(Model::class, function (Faker $faker) {
    return [
        //
    ];
});
```

Primeira coisa que precisamos é definir o model Post, para o criar vamos ao terminal e utilizamos o seguinte comando na CMD:

php artisan make:model Post

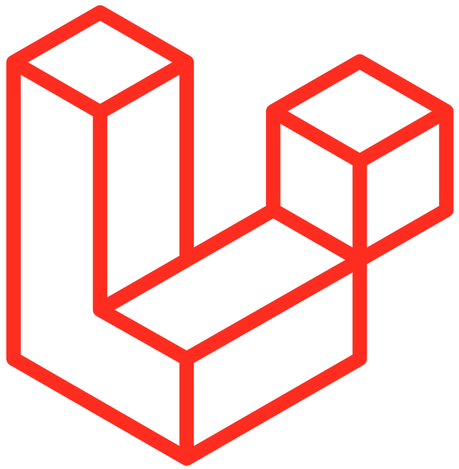
Database: Factories



E alteramos a factory

```php
$factory->define(\App\Post::class, function (Faker $faker) {
    return [
        'title'       => $faker->words(4, true),
        'description' => $faker->sentence,
        'content'     => $faker->paragraphs(9, true),
        'slug'        => $faker->slug,
        'is_active'   => $faker->boolean,
        'user_id'     => rand(1, 10)
    ];
});
```

Database: Factories



Após isso só temos que alterar a chamada da seed:

```
/*
\DB::table('posts')->insert([
    'title'       => 'First Title',
    'description' => 'First Post Description',
    'content'     => 'First Post Content',
    'is_active'   => 1,
    'slug'        => 'first-post',
    'user_id'     => 1,
]);
*/
factory(\App\Post::class,30)->create();
```

E executar a seed pela CMD:

php artisan db:seed --class=PostsTableSeeder

php artisan db:seed

P H P

Database: CMD

Todos os comandos de migrações estão disponíveis na documentação:

https://laravel.com/docs/7.x/migrations

Database: Migrations, Seeds & Factories

Criar migration, seeder, factory, e resource controller para o modelo:

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

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

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

Database: Migrations, Seeds & Factories

Criar migration, seeder, factory, e resource controller para o modelo:

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

-f, --factory Create a new factory for the model

--force Create the class even if the model already exists

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

-s, --seed Create a new seeder file for the model

-p, --pivot Indicates if the generated model should be a custom intermediate table model

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