サーバーサイドスクリプトⅡ

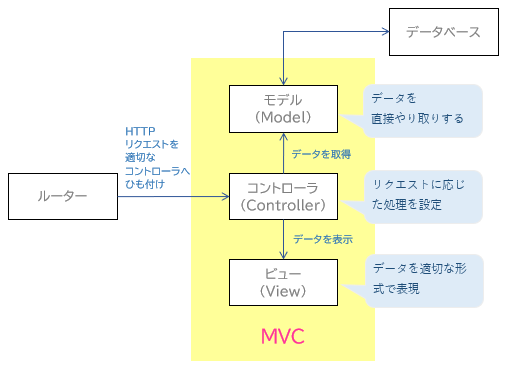
５．マイグレーション、　モデル、　シーダー

データベースの内容を自動生成するために用意されているのが「マイグレーション」と「シーディング」です。本章ではそれらの使い方を習得します。

今回の範囲はMVCすべてとなり、今まで実施していなかった下記①②の手順を行っていきます。（③以降のコントローラなどは次回作成）

1. **マイグレーション**
2. **モデルの作成**
3. **コントローラの作成**
4. **ルーティングの設定**
5. **ビューの作成**

■図：　MVC



５－１． マイグレーション

Laravelではデータベース（以下DBと略）のテーブル作成や定義変更を管理する機能のことを指します。PHPのソースコードで、DBに対してCREATE TABLEやALTER TABLE文などを発行する仕組みです。定義の反映はartisanコマンドを使用します。DBに直接接続して操作する必要はありません。

５－１－１． 前提条件

本授業ではDBはMySQLを使用します。DBのテーブル作成や定義変更を管理するためには、MySQLのautocommit機能をオンにする必要があります。

ターミナルもしくはコマンドプロンプトからＭｙSQLにログインし、autocommitの状態を確認します。

■自動コミットモードの確認コマンド

＊MySQLにログインしてから実行

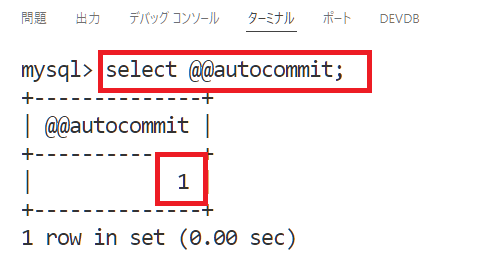


■実行結果

　＊下図のように結果に「１」と出ていれば　autocommitがオンになっています。

　＊「０」であれば設定が必要。MySQLのmy.iniファイルの設定で「autocommit=1」とすること。

　　（MySQLのmy.iniファイル詳細についてはDBの授業資料参照）



５－１－２． マイグレーション処理の流れ

マイグレーション全体の流れを記載します。一つ一つの手順については後述します。

1. マイグレーションファイル作成コマンドを実行
2. 定義を適用するコードと定義を削除するコードを記述
3. マイグレーション実行コマンドを実行（DBに定義を反映）

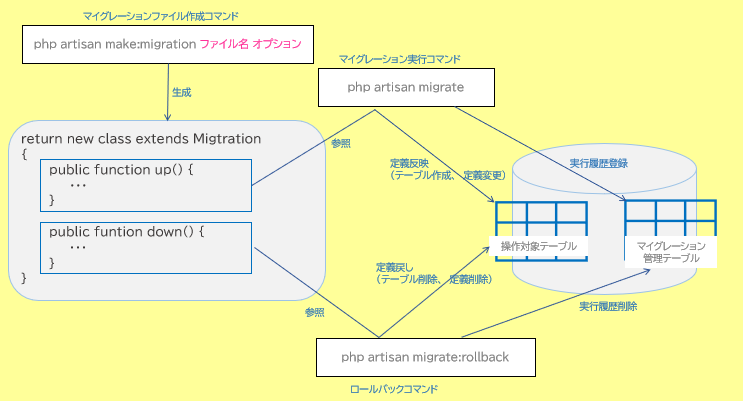
もしくは

ロールバックコマンドを実行（DBの定義を戻す）

＊③の実行履歴はマイグレーション管理テーブルで管理される。

　　　　（このとき、MySQLのautocommitがオフだと管理がうまくいかない。）

■マイグレーションの流れ



５－１－３． マイグレーションファイルの作成

マイグレーションファイルはartisanコマンドで作成します。マイグレーションファイルの作成される場所は、「プロジェクトフォルダ\**database\migrations**」フォルダ内です。

■マイグレーションファイル作成コマンド



　＊ファイル名は「**create\_テーブル名\_table**」とします。

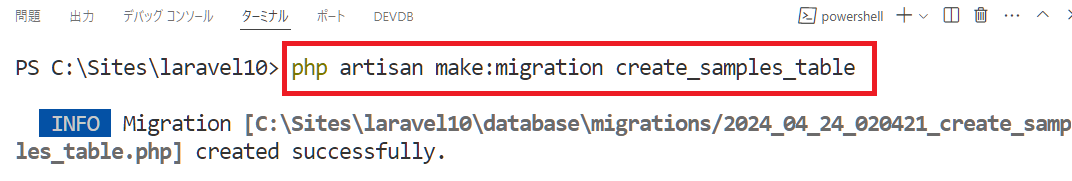
**マイグレーションファイルにテーブルを新規追加するための記述が追加されます**。

　＊オプションについては、サンプルでは省略します。

■samplesテーブルのマイグレーションファイルを作成

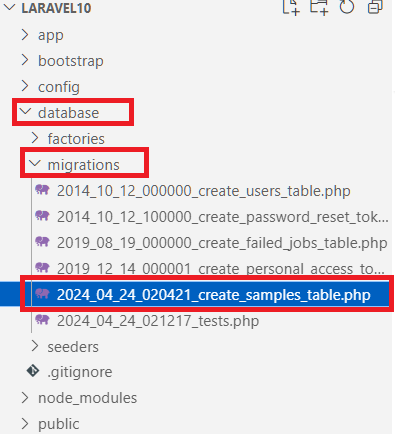
　「php artisan make:migration create\_samples\_table」を実行。

　＊「～～successfully」と表示されればOK



■作成されたマイグレーションファイル

　マイグレーションファイル名には、指定した**ファイル名の先頭に「作成日付と番号」が付加**されます。



５－１－３． マイグレーションの設定

作成したマイグレーションファイルに記述していきます。

まずは作成したマイグレーションファイルを確認してください。

■yyyy\_mm\_dd\_xxxxx\_create\_samples\_table.php

　（yyyy\_mm\_dd\_xxxxx\_部分は作成日と番号が入る。以降create\_samples\_table.phpで省略）

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

return new class extends Migration

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

        Schema::create('samples', function (Blueprint $table) {

            $table->id();

            $table->timestamps();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

        Schema::dropIfExists('samples');

    }

};

upメソッドには、DB定義の追加（あるいは変更）を行うための処理を記述します。

downメソッドには、upメソッドの内容を戻す処理を記述します。

サンプルでは、テーブルを新規作成するため、upメソッドにはSchena::createメソッド、downメソッドにはSchema::dropIfExistsメソッドが記述されています。

今回は、downメソッドはデフォルトのまま使用します。

upメソッドに今回作成するテーブル定義を記載します。

テーブル構造は、**Schemaクラスの「create」メソッドで定義**します。

createメソッドの第一引数はテーブル名を指定し、第２引数は、新しいテーブルを定義するために使用できるBlueprintオブジェクトを受け取るクロージャとなり、Blueprintオブジェクトのクロージャー内で、テーブルのカラム指定をおこないます。

デフォルトでは、idと登録日時・更新日時の列が記述されています。

サンプルでは、下記の仕様で記述していきます。（サンプルのコードは後述。）

■samplesテーブル仕様

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 型 | 制約 | 備考 |
| id | AUTO\_INCREMENT | UNSIGNED |  |
| title | VARCHAR(255) | ----- | タイトル |
| body | TEXT | NULL許可 | 本文 |
| created\_at | TIMESTAMP | CURRENT\_TIMESTAMPを  デフォルト値として使用 | 作成日時 |
| updated\_at | TIMESTAMP | ●CURRENT\_TIMESTAMPを  デフォルト値として使用  ●レコードが更新されたときに  CURRENT\_TIMESTAMPを使用 | 更新日時 |
| deleted\_at | ソフトデリート用  TIMESTAMPカラム | ------ | 論理削除用  （＊） |

|  |  |
| --- | --- |
| 警告 単色塗りつぶし | **（＊）論理削除とは**  論理削除とは、DBのレコードを削除する際、現在有効か否かを示すフィールドの値を変更することで削除したのと同じ扱いにする方式。 実際のデータは削除しない。 |

■create\_samples\_table.php

　＊upメソッド部分のみ記載。

public function up(): void

{

    Schema::create('samples', function (Blueprint $table) {

        $table->id()->unsigned();

        $table->string('title', 255);

        $table->text('body')->nullable();

        $table->timestamp('created\_at')->useCurrent();

        $table->timestamp('updated\_at')->useCurrent()->useCurrentOnUpdate();

        $table->softDeletes();

    });

}

生成できる主なカラムタイプを下記に記載します。

■生成できるカラムタイプとスキーマビルダの構文（次ページへつづく）

|  |  |  |
| --- | --- | --- |
| タイプ | スキーマビルダ | 備考 |
| BOOLEAN型 | $table->boolean（’カラム名’）; | ----- |
| CHAR型 | $table->char(‘カラム名’,サイズ); | ----- |
| DATE型 | $table->date(‘カラム名’); | ----- |
| DATETIME型 | $table->datetime(‘カラム名’); | ----- |
| DOUBLE型 | $table->double(‘カラム名’,最大桁数,小数点以下の桁数); | ----- |
| FLOAT型 | $table->float(‘カラム名’,最大桁数,小数点以下の桁数); | ----- |
| ID（主キー）カラム | $table->id(); | 自動インクリメントするBIGINT型カラムを生成 |
| INTEGER型 | $table->integer(‘カラム名’); | ----- |
| ソフトデリート用  TIMESTAMPカラム | $table->softDeletes(); | ソフトデリートのための  「deleted\_at」カラムを追加 |
| VARCHAR型 | $table->string(‘カラム名’,サイズ); | ----- |
| TEXT型 | $table->text(‘カラム名’); | ----- |
| TIMESTAMP型 | $table->timestamp(‘カラム名’); | ----- |
| 登録/更新日時用  TIMESTAMPカラム | $table-> timestamp (); | データ登録日時と更新日時のための「created\_at」と  「updated\_at」カラムを追加 |

また、カラムに属性を与えるためのメソッド一覧も記載します。

■カラムに属性を与えるメソッド

|  |  |
| --- | --- |
| メソッド | 内容 |
| ->after(‘カラム名’) | 引数で指定したカラムの直後に配置する（MySQLのみ有効） |
| ->nullable() | カラムにNULL値の挿入を許可する |
| ->default(デフォルト値) | カラムのデフォルト値を指定する |
| ->unsigned() | 数値型のカラムをUNSIGNED（符号なし）にする |

５－１－４． マイグレーションの実行

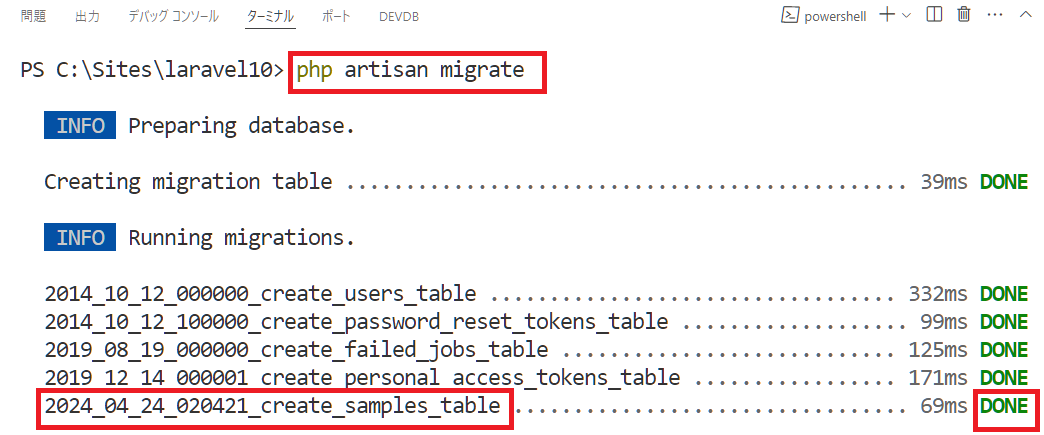
作成したマイグレーションファイルはmigrateコマンドを使ってDBへ反映させます。

migrateコマンドはdatabase/migrationsフォルダにあるマイグレーションファイルで、DBにまだ反映されていないものを一括で反映するコマンドです。

■マイグレーション実行コマンド



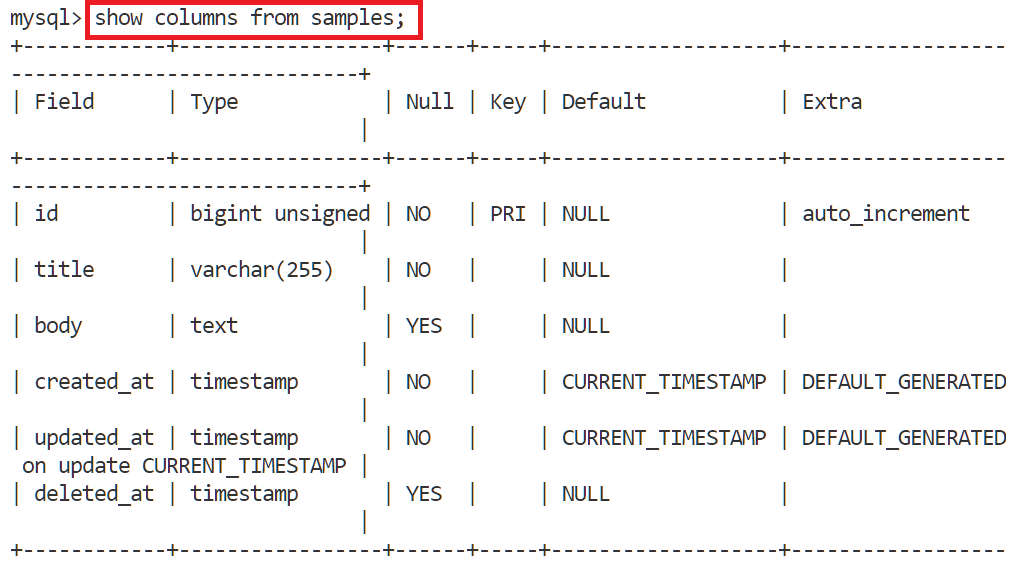
■サンプル環境で実行



実行が完了したことを確認します。今回サンプルで作成したsamplesテーブル以外は、Laravelに用意されている機能で利用するテーブルです。今回はsamplesテーブルについて、見ていきます。

MySQLにログインして、samplesテーブルが作成されていることを確認します。

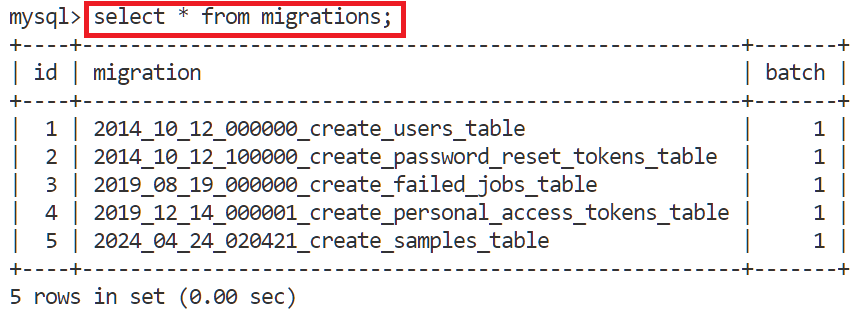
* samplesテーブル確認　＊ターミナルもしくはコマンドプロンプトで実行



また、マイグレーションを実行すると、実行状態を管理する「migrations」テーブルも作成されています。

続いてmigrationsテーブルの内容を確認してみます。

■migrationsテーブル確認



マイグレーションファイル名と実行順が管理されています。migrateコマンドはこのテーブルに登録されていないマイグレーションファイルを実行し、migrate:rollbackコマンドは「batch」がもっとも大きい番号のマイグレーションファイルが対象となります。

サンプルではロールバックなどのコマンドは実行しませんが、主なマイグレーションコマンドには下記のようなものがあります。

■マイグレーション主要コマンド

＊php artisanは省略して記載。

|  |  |
| --- | --- |
| コマンド | 内容 |
| migrate | マイグレーションを実行する |
| migrate:rollback | 最後に実行したマイグレーションの実行をロールバックする |
| migrate:reset | 全てのマイグレーションの実行をロールバックする |
| migrate:refresh | 全てのマイグレーションの実行をロールバックしてから、マイグレーションを実行する |
| migrate:fresh | 全てのテーブルを削除してから、マイグレーションを実行する |
| migrate:status | マイグレーション状況を表示する |

５－２． モデル

次にモデルを作成します。モデルはテーブルの情報を操作することができるクラスです。

Laravelには、データベースとのやり取りをサポートするオブジェクトリレーショナルマッピング（ORM）（＊）であるEloquentというライブラリが含まれています。

|  |  |
| --- | --- |
| 警告 単色塗りつぶし | **（＊）オブジェクトリレーショナルマッピング（ORM）とは**  DBとモデルを関連付ける仕組みです。モデルを利用してテーブルからレコードを取得すると、1件のレコードが1つのモデルオブジェクトに格納されます。また、モデルオブジェクトに格納された値をテーブルに登録することができます。 |

５－３－１． Eloquentのクラスファイル作成

Eloquentのクラスファイル（以降、モデルと記載）を作成するには、artisanコマンドを使用します。

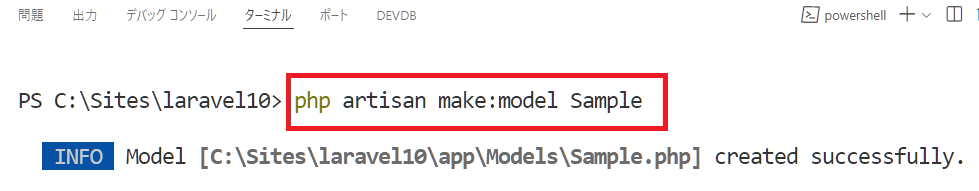
**モデル名はテーブル名の単数形にすることで、自動的にテーブルとモデルのマッピングが行われます。**

■モデルの作成



■サンプル環境での実行

　＊samplesテーブルのモデルを作成するので、モデル名は単数形の「sample」とします。



作成したモデルクラスは、「プロジェクトフォルダ\app\Models」の配下にあります。

５－３－２．モデルの設定

作成したモデルクラスを確認します。

■Sample.php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;

class Sample extends Model

{

    use HasFactory;

}

初期状態では、use HasFactory;のみが記述されています。具体的な処理は親クラスで実装されています。そのため、基本的な機能はプログラミングしなくても利用できるようになっています。

今回は論理削除を使用しますので、「use SoftDeletes;」を追記します。また、モデル名をテーブル名で作成した場合は省略可能ですが、テーブル名の紐付けの方法も併せて確認します。

＊追記コードは次ページに記載。

■Sample.php

＊赤枠部分を追記。

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\SoftDeletes;

class Sample extends Model

{

    use HasFactory;

    use SoftDeletes;                // 論理削除の場合

    // モデル名をテーブル名の単数形とした場合は省略可↓

    protected $table = 'samples';  // テーブル名

}

今回はプライマリキーにidを用いているので必要ありませんが、任意のカラムをプライマリキーにしたい場合は、このクラスでの指定が必要です。また、編集可能なカラムの設定や編集不可なカラムの設定なども、このクラスで行います。

＊今回は基本的な使用方法について学びますので、詳しくは割愛します。

５－３． シーダー

マイグレーションの実行でテーブルの作成が完了し、モデルの作成でデータを操作するクラスの準備も完了しましたが、まだデータはない状態です。テーブルでは作成に合わせて初期データ投入が必要となるケースがあります。Laravelではデータ投入をコードで実行する仕組みが用意されています。

シーダー（Seeder）クラスを用いて、データ投入を行います。

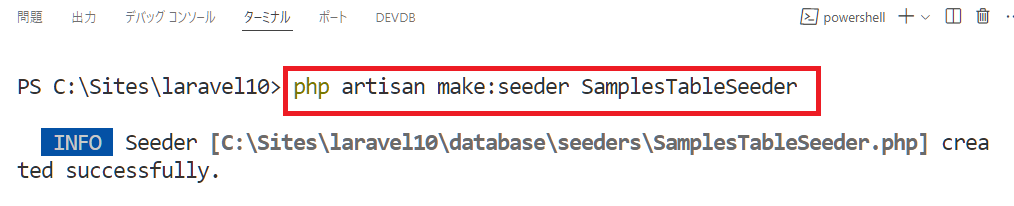
５－３－１． シーダーの作成

LaravelではデフォルトでDatabaseSeederクラスが定義されています。シードファイルの作成は、artisanコマンドを使用します。

■シードファイルの作成



■サンプル環境で実行



実行が完了したことを確認します。作成したファイルは「プロジェクトフォルダ\database\seeders\」の配下にあります。

■SamplesTableSeeder.php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;

use Illuminate\Database\Seeder;

class SamplesTableSeeder extends Seeder

{

    public function run(): void

    {

        //

    }

}

シーダークラスには、デフォルトで、**runメソッド**のみ存在します。

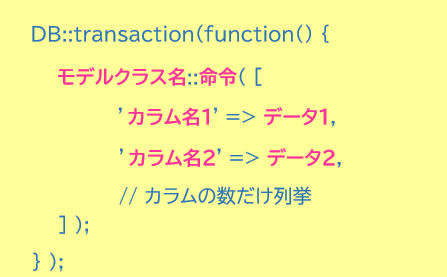
このメソッドは、 **artisanコマンドの「 db:seed 」が実行されるときに呼び出される**ため、データベースに挿入するデータを定義できます。

５－３－２． 挿入するデータの作成

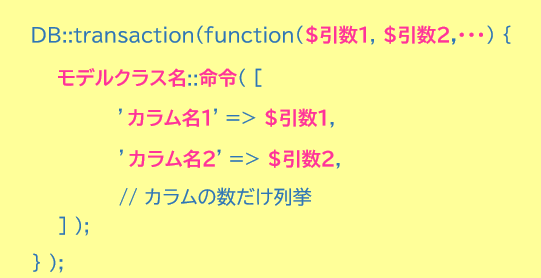
SampleTableSeeder.phpに挿入するデータの定義を行います。

自動でコミットとロールバックを行うコードを記載していきます。

■DB::transactionメソッド（クロージャ内　引数なし）



■DB::transactionメソッド（クロージャ内　引数あり）



サンプル環境では、samplesテーブルに3件のデータを挿入するコードを記述します。

■SamplesTableSeeder.php

　＊runメソッド部分のみ抜粋。

public function run(): void

{

    //

    DB::transaction(function () {

        Sample::insert([

            [

                'title' => 'Global Education Awards',

                'body'  => '本文１'

            ],

            [

                'title' => '地球祭が今年も開催！！',

                'body'  => '本文２'

            ],

            [

                'title' => ' 「ハロウィンDay」でした。',

                'body'  => '本文３'

            ],

        ]);

    });

}

５－３－３． シーダーの登録

データの定義が完了したので、続いてDatabaseSeederクラス内で、callメソッドを使用して追加のシードクラスを実行します。

callメソッドを使用することで、データベースのシードを複数のファイルに分割して、単一のシーダークラスが大きくなりすぎないようにできます。

「プロジェクトフォルダ\database\seeders\」の配下にあるDatabaseSeeder.phpファイルを開きます。

■DatabaseSeeder.php

　＊デフォルトで存在するクラスファイル。今回作成したSampleTabldeSeederではないので注意。

　＊下記はデフォルト状態のコード

namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder

{

    /\*\*

     \* Seed the application's database.

     \*/

    public function run(): void

    {

        // \App\Models\User::factory(10)->create();

        // \App\Models\User::factory()->create([

        //     'name' => 'Test User',

        //     'email' => 'test@example.com',

        // ]);

    }

}

今回のサンプルではSampleTableSeederクラスを呼び出すように記述します。呼び出しにはcallメソッドを使用します。

■callメソッド

ロゴ が含まれている画像

自動的に生成された説明

runメソッドの中身を下記のように変更してください。

■DatabaseSeeder.php

　＊runメソッド部分のみ抜粋。赤枠部分を追加。

public function run(): void

{

    $this->call([

        SamplesTableSeeder::class,

    ]);

    // \App\Models\User::factory(10)->create();

    // \App\Models\User::factory()->create([

    //     'name' => 'Test User',

    //     'email' => 'test@example.com',

    // ]);

}

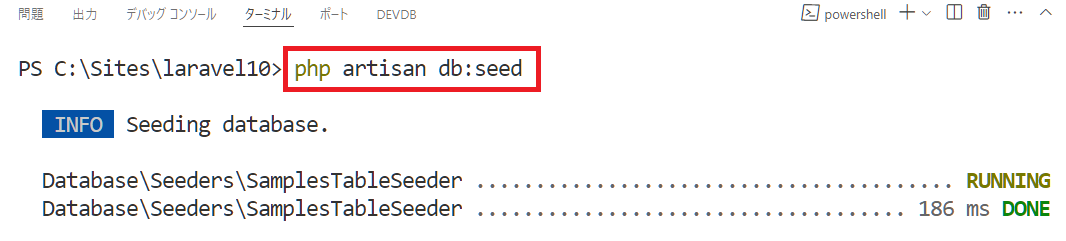
５－３－４． シーダーの実行

作成・登録したシーダークラスを実行します。実行にはartisanコマンドを使用します。

■シーダーの実行



■サンプル環境での実行

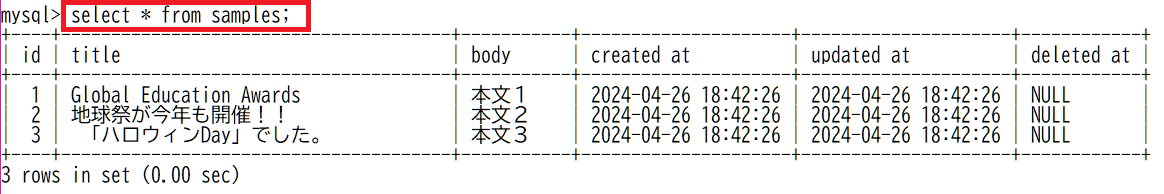


実行が完了したら、MySQLでsamplesテーブルを確認します。

コマンドプロンプトでDB名laravel10のsamplesテーブルのデータをselectします。

■samplesテーブルデータ

　＊SampleTableSeederクラスで定義したデータが挿入されていることを確認。



５－４． デモデータの投入

SampleTableSeederへのデータの投入が完了しましたが、本番環境を想定したテストを行う場合、「本文１」「本文２」などではなく、より実データに近いテストデータが必要なケースがあります。また、大量のテストデータの投入を行いたい場合、前述の方法では効率的ではありません。

Laravelでは、テストデータを簡単に作成できるFakerライブラリが用意されています。このFakerを利用してデータの投入を行います。また、大量のデータ投入にはFactoryクラスを使用すると便利です。

５－４－１． フェイカーライブラリの活用

Fakerはテスト用のダミーデータを生成するPHPライブラリになります。

Laravelには、テストデータの生成用にフェイカー（Faker）ライブラリが最初から組み込まれています。

「プロジェクトフォルダ\ｃonfig\app.php」 内の下記オプションの設定を変更します。

■faker\_localeオプション

アプリケーションのFakerロケール。

＊デフォルトでは、「 en\_US 」になっているため「ja\_JP」へ修正。ダミーデータを日本語で生成。

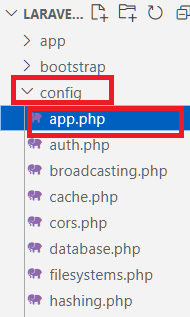
■timezoneオプション

Webアプリケーションで使用するタイムゾーン。

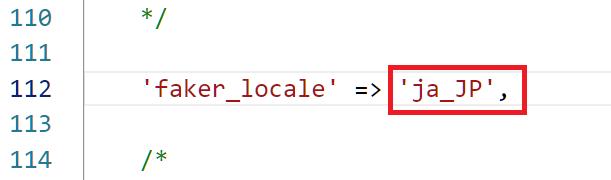
＊デフォルトでは「 UTC 」になっているため、「 Asia/Tokyo 」へ修正。

＊「 php.ini 」や「.htaccess」で指定せずに変更することが可能。

■configフォルダ内のapp.php



■app.php（faker\_locale）　＊赤枠部分を変更



■app.php（timezone）　＊赤枠部分を変更



修正したら、上書き保存をしてください。

５－４－２． ファクトリーの作成と設定

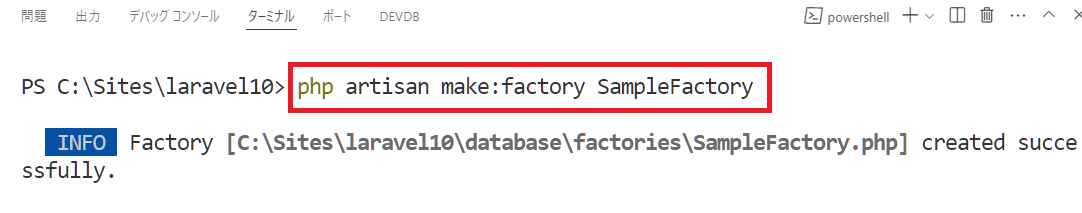
テストデータを作成するときに各カラムの値を自分でいちいち指定する代わりに、Laravelではファクトリー（Factory）を使用して、デフォルト属性を定義できます。

ファクトリーの作成にはartisanコマンドを使用します。

■Factoryの作成



■サンプル環境で実行



作成したファイルは「プロジェクトフォルダ\database\factories」配下にあります。

SampleFactory.phpファイルを確認します。デフォルトで用意されているdefinitionメソッドで投入するデータを定義します。

■SampleFactory.php　＊赤枠部分を追記。

**＊フェイカーのダミーデータを使用する。**

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/\*\*

 \* @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Sample>

 \*/

class SampleFactory extends Factory

{

    /\*\*

     \* Define the model's default state.

     \*

     \* @return array<string, mixed>

     \*/

    public function definition(): array

    {

        　return [

            //

            "title" => $this->faker->realText(20),

            "body"  => $this->faker->realText(200),

            "created\_at" => $this->faker->dateTime("now"),

        ];

    }

}

definitionメソッドは、ファクトリーを使用してモデルを作成するときに適用する必要がある属性値のデフォルトセットを返します。サンプルではフェイカーのダミーデータを利用しています。

フェイカーを利用せずに、下記のように記述することも可能です。（下記例では1件分のデータ）

■記述例（今回は使用せず）

public function definition()

{

    return [

        "title" => "サンプルのタイトル",

        "body"  => "サンプルの投稿内容。",

    ];

}

５－４－３． シーダーでファクトリーの利用

作成したファクトリーをシーダーで利用することで、シーディング時にファクトリーで定義したテストデータを使用することができます。

SampleTableSeederクラスを修正します。

＊コードが長いため、次ページへ記載。

■SampleＴａｂｌｅＳｅｅｄｅｒ.php

　＊runメソッドのみ抜粋。

**追加**

**Sample::insertメソッド部分**

**コメントアウト**

public function run(): void

{

    //

    DB::transaction(function () {

        // Sample::insert([

        //     [

        //         'title' => 'Global Education Awards',

        //         'body'  => '本文１'

        //     ],

        //     [

        //         'title' => '地球祭が今年も開催！！',

        //         'body'  => '本文２'

        //     ],

        //     [

        //         'title' => ' 「ハロウィンDay」でした。',

        //         'body'  => '本文３'

        //     ],

        // ]);

        // モデルファクトリーを利用したテストデータの挿入（100件分）

        Sample::factory()->count(100)->create();

    });

}

修正が完了したら、**「php artisan db:seed」コマンドでシーダーを実行**してください。samplesテーブルに100件のテストデータが追加で投入されていれば完了です。

最初に投入した３件＋ファクトリーで投入した100件で、合わせて103件のデータが揃っています。サンプル環境では、見易さのためツールを用いてsamplesテーブルのデータを確認しています。

■samplesテーブルのデータ



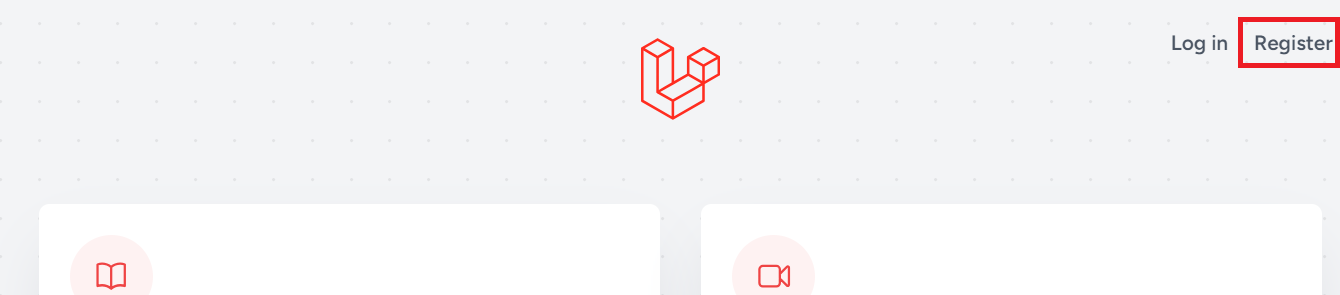
今回は以上です。データの使用は次回以降に行います。

５－５． おまけ（Laravelのデフォルトログイン機能）

マイグレーションを実行した際に、デフォルトのUsersテーブルも作成されています。これを利用してLaravelで用意されているログイン機能を使ってみましょう。

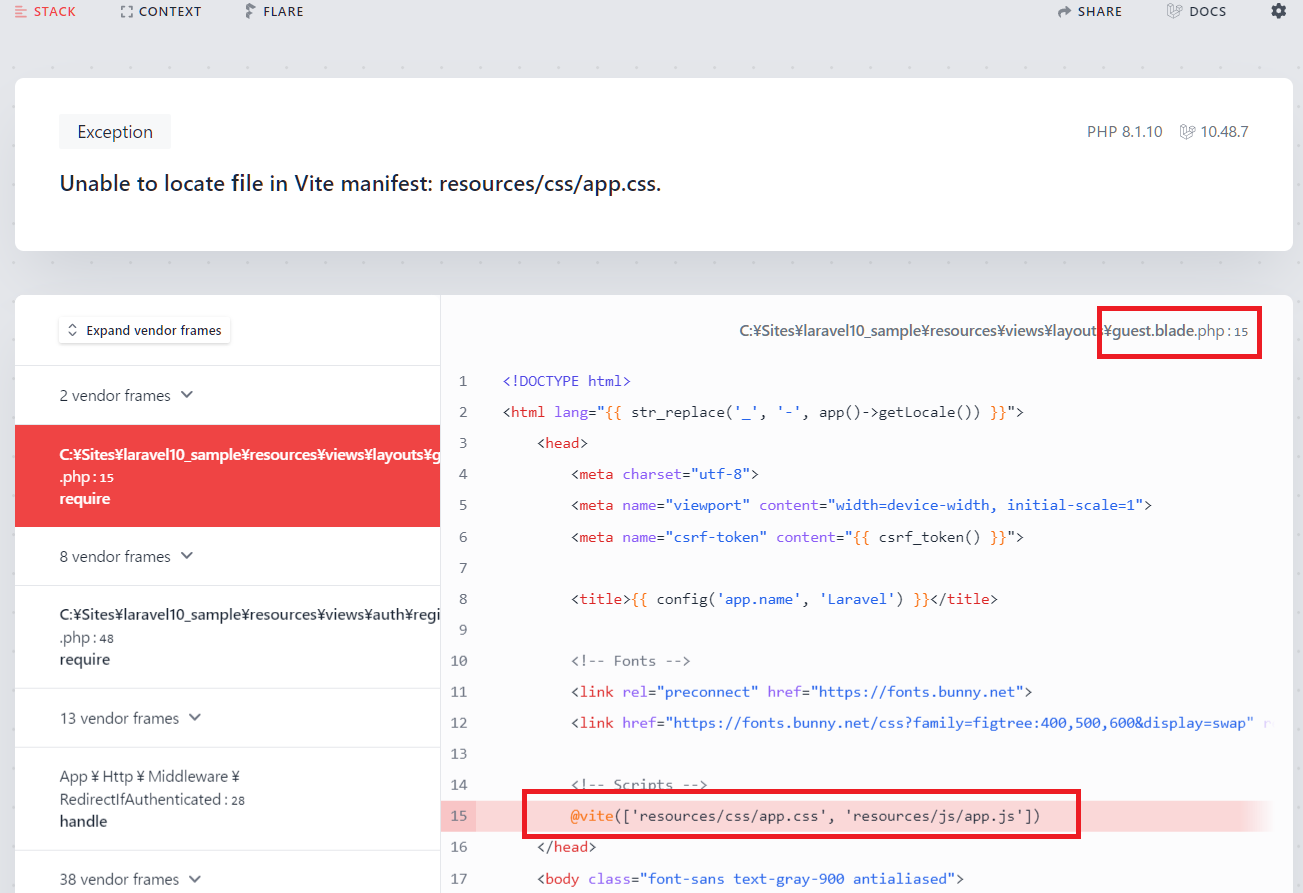
1. 「<http://localhost:8000/>」へアクセス。

最初はユーザーが存在しないので、「Register」から新規登録を行う。



1. エラーが発生します。

＊「プロジェクトフォルダ\\resources\views\layouts\guest.blade.php」内のCSSファイルの読み込みエラーが起きます。



1. サンプル環境ではsassを利用するようにしたので、そちらに合わせてguest.blade.phpファイルの記述を修正します。

■guest.blade.php

<!DOCTYPE html>

<html lang="{{ str\_replace('\_', '-', app()->getLocale()) }}">

<head>

～　中略　～

    <!-- Scripts -->

    @vite(['resources/sass/app.scss', 'resources/js/app.js'])

</head>

1. 同じく、app.blade.phpファイルも修正しておきます。

■app.blade.php

<!DOCTYPE html>

<html lang="{{ str\_replace('\_', '-', app()->getLocale()) }}">

    <head>

        <meta charset="utf-8">

        <meta name="viewport" content="width=device-width, initial-

～　中略　～

    <!-- Scripts -->

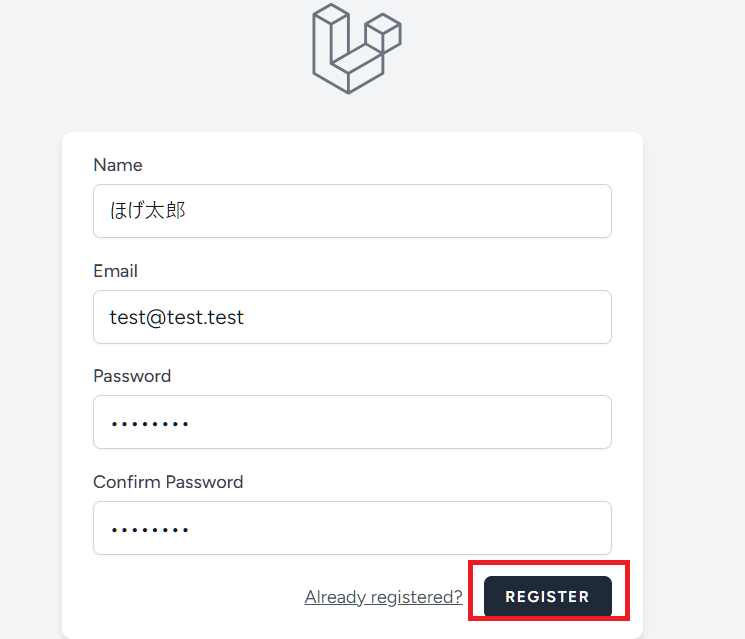
    @vite(['resources/sass/app.scss', 'resources/js/app.js'])

</head>

1. 再度、「<http://localhost:8000/>」へアクセス。

最初はユーザーが存在しないので、「Register」から新規登録を行う。

1. 必要事項を入力して「Register」ボタン押下。



1. ログインが完了するとダッシュボード画面へ遷移する。

グラフィカル ユーザー インターフェイス, アプリケーション

自動的に生成された説明

標準のログイン機能はテンプレートを利用しているので、ナビゲーション部分のみ他のページで利用することも可能です。

また、先ほど作成したユーザーがテーブルに追加されていることも確認してみましょう。

以上です。