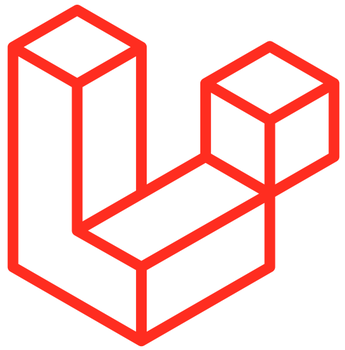
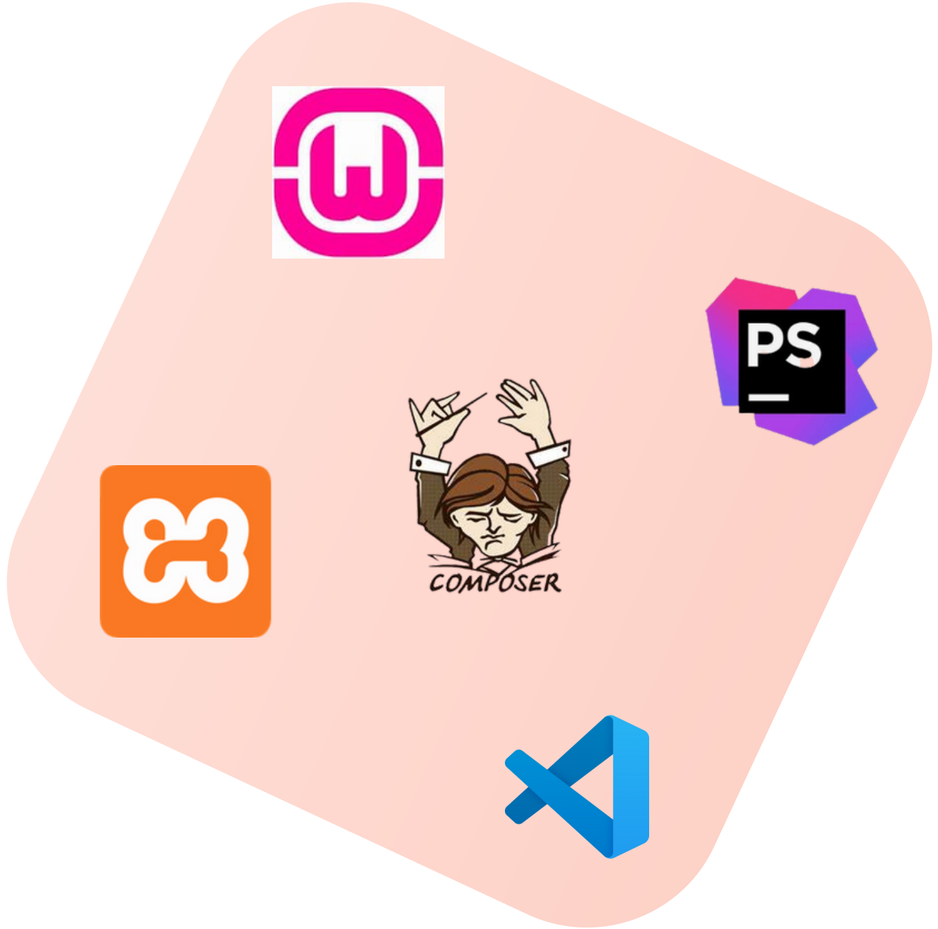


Laravel

Mohammed Najem Ali

Requirements

- Xampp/Wampserver/Mamp
- Install Composer
- Phpstorm/VSCode



Installation

Before creating your first Laravel project, you should ensure that your local machine has PHP and Composer installed.

```
composer create-project laravel/laravel example-app
```

```
cd example-app

php artisan serve
```

```
composer global require laravel/installer

laravel new example-app
```

MVC

The MVC (Model-View-Controller) pattern is a software architectural pattern that separates an application into three interconnected components: the Model, View, and Controller.

Model

Represents the data and business logic of the application. It interacts with the database, performs data manipulation, and enforces rules and validations.

View

Presents the user interface to the users. It receives data from the Model and displays it in a visually appealing and understandable format.

Controller

Handles user interactions and acts as the intermediary between the Model and View. It receives input from the user, updates the Model accordingly, and communicates with the View to display the appropriate information.

Routing

To define routes in Laravel, you can use the routes/web.php file. This file serves as the entry point for defining web routes.

The first route responds to a GET request to the root URL ("/") and returns the string "Hello, Laravel!" as the response.

The second route responds to a POST request to the "/submit" URL and calls the submit method of the FormController class.

```php
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return 'Hello, Laravel!';
});

Route::post('/submit', 'FormController@submit');
```

Additionally, you can define route parameters to capture dynamic values from the URL.

```php
Route::get('/users/{id}', function ($id) {
    return 'User ID: ' . $id;
});
```

Controllers

Controllers play a crucial role in Laravel by serving as the intermediary between routes, models, and views. They handle incoming requests, process data, and coordinate the flow of information within an application.
Controllers are responsible for processing requests made to your application and returning a response.

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

```
php artisan make:controller UserController
```

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

```
php artisan make:controller ProvisionServer --invokable
```

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

```
php artisan make:controller PhotoController --resource
```

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Views (Blade)

Views separate your controller / application logic from your presentation logic and are stored in the resources/views directory. When using Laravel, view templates are usually written using the Blade templating language.

- Blade templates are compiled into plain PHP code, which means that they are executed at runtime and can be cached for improved performance.
- Blade provides a number of directives that allow you to insert PHP code, variables, and expressions into your templates.
- Blade supports inheritance, which allows you to create reusable templates that can be extended by other templates.
- Blade components are a way to encapsulate reusable UI elements into their own files.

Models

Models provide a powerful and enjoyable interface for you to interact with the tables in your database.

You may use the following Artisan command to generate a new model:

```
php artisan make:model -mrc Note
```

Provides a convenient overview of all the model's attributes and relations:

```
php artisan model:show Note
```

Eloquent will assume the Flight model stores records in the flights table, while an AirTrafficController model would store records in an air_traffic_controllers table.

```
protected $table = 'my_flights';
```

The Eloquent all method will return all of the results in the model's table. However, since each Eloquent model serves as a query builder, you may add additional constraints to queries and then invoke the get method to retrieve the results:

```
foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

```
$flights = Flight::where('active', 1)
               ->orderBy('name')
               ->take(10)
               ->get();
```

Retrieving Single Models

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the find, first, or firstWhere methods. Instead of returning a collection of models, these methods return a single model instance:

```php
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);
```

Eloquent: Relationships

One To One

A one-to-one relationship is a very basic type of database relationship. For example, a User model might be associated with one Phone model.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

```php
$phone = User::find(1)->phone;
```

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

Eloquent: Relationships

One To Many

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments.

```php
class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

```php
class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

Eloquent: Relationships

Many To Many

An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application.

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

```php
class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

```php
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

```php
class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

Eloquent: Relationships

Polymorphic Relationships

One To One (Polymorphic)

A one-to-one polymorphic relation is similar to a typical one-to-one relation; however, the child model can belong to more than one type of model using a single association. For example, a blog Post and a User may share a polymorphic relation to an Image model. Using a one-to-one polymorphic relation allows you to have a single table of unique images that may be associated with posts and users.

```
posts
    id - integer
    name - string

users
    id - integer
    name - string

images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

```php
use Illuminate\Database\Eloquent\Relations\MorphTo;
class Image extends Model
{
    /**
     * Get the parent imageable model (user or post).
     */
    public function imageable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

```php
use Illuminate\Database\Eloquent\Relations\MorphOne;
class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

```php
use Illuminate\Database\Eloquent\Relations\MorphOne;
class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image(): MorphOne
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

```php
$post = Post::find(1);
$image = $post->image;
```

```php
$image = Image::find(1);
$imageable = $image->imageable;
```

Eloquent: Relationships

Polymorphic Relationships

One To Many (Polymorphic)

A one-to-many polymorphic relation is similar to a typical one-to-many relation; however, the child model can belong to more than one type of model using a single association. For example, imagine users of your application can "comment" on posts and videos. Using polymorphic relationships, you may use a single comments table to contain comments for both posts and videos.

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

```php
use Illuminate\Database\Eloquent\Relations\MorphTo;
class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).
     */
    public function commentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

```php
use Illuminate\Database\Eloquent\Relations\MorphMany;
class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

```php
use Illuminate\Database\Eloquent\Relations\MorphMany;
class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Eloquent: Relationships

Polymorphic Relationships

Many To Many (Polymorphic)

a Post model and Video model could share a polymorphic relation to a Tag model. Using a many-to-many polymorphic relation in this situation would allow your application to have a single table of unique tags that may be associated with posts or videos.

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

```php
class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags(): MorphToMany
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

```php
class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts(): MorphToMany
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos(): MorphToMany
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

```php
$tag = Tag::find(1);
foreach ($tag->posts as $post) {
    // ...
}
foreach ($tag->videos as $video) {
    // ...
}
```

Authentication

Application Starter Kits

Laravel Breeze: Simple Laravel authentication with Blade + Tailwind CSS.
Laravel Fortify: Headless Laravel authentication backend with features like two-factor auth and email verification.
Laravel Jetstream: Robust app starter kit with modern UI (Tailwind CSS, Livewire, Inertia) consuming Fortify's services. Optional features include 2FA, teams, session management, and API token auth with Sanctum.

Laravel's API Authentication Services

Laravel offers two packages for API token management: **Passport** and **Sanctum**. They work alongside Laravel's cookie-based authentication services, which focus on browser authentication. Applications may use both approaches simultaneously.

```php
$user = Auth::user();
// $id = Auth::id();
// $user = $request->user();
if (Auth::check()) {
    // The user is logged in...
}
```

Manually Authenticating Users

```php
class LoginController extends Controller
{
    public function authenticate(Request $request): RedirectResponse
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();
            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ])->onlyInput('email');
    }
}
```

Authentication

```php
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // Authentication was successful...
}
```

```php
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

```php
if (Auth::viaRemember()) {
    // to determine if the currently user was authenticated using the "remember me" cookie
}
```

```php
Auth::login($user);
Auth::login($user, $remember = true);
Auth::guard('admin')->login($user);
Auth::loginUsingId(1);
if (Auth::once($credentials)) {
    // No sessions or cookies will be utilized when calling this method
}
```

```php
public function logout(Request $request): RedirectResponse
{
    Auth::logout();
    $request->session()->invalidate();
    $request->session()->regenerateToken();
    return redirect('/');
}
```

Authorization

Laravel provides two primary ways of authorizing actions: gates and policies. Think of gates and policies like routes and controllers.

Gates are most applicable to actions that are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Gates

Gates are simply closures that determine if a user is authorized to perform a given action. Typically, gates are defined within the boot method of the App\Providers\AuthServiceProvider class using the Gate facade.

```
public function boot(): void
    {
        Gate::define('update-note', function (User $user, Note $note) {
            return $user->id === $note->user_id;
        });
    }
```

Like controllers, gates may also be defined using a class callback array

```
Gate::define('update-note', [NotePolicy::class, 'update']);
```

Authorization

Authorizing Actions

To authorize an action using gates, you should use the allows or denies methods provided by the Gate facade. Note that you are not required to pass the currently authenticated user to these methods. Laravel will automatically take care of .passing the user into the gate closure

```
public function update(Request $request, Note $note)
    {
        if (! Gate::allows('update-note', $note)) {
            abort(403);
        }

    }
```

If you would like to determine if a user other than the currently authenticated user is authorized to perform an action, you may use the forUser method on the Gate facade:

```
if (Gate::forUser($user)->denies('update-post', $post)) {
    // The user can't update the post...
}
```

You may authorize multiple actions at a time using the any or none methods:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user can't update or delete the post...
}
```

Authorization

Authorizing Actions

If you would like to attempt to authorize an action and automatically throw an Illuminate\Auth\Access\AuthorizationException if the user is not allowed to perform the given action, you may use the Gate facade's authorize method.

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

Gate Responses

Sometimes you may wish to return a more detailed response, including an error message.

```
Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::deny('You must be an administrator.');
});
```

Even when you return an authorization response from your gate, the Gate::allows method will still return a simple boolean value; however, you may use the Gate::inspect method to get the full authorization response returned by the gate:

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

Response::denyWithStatus(404) and Response::denyAsNotFound() can be used as response

Occasionally, you may wish to determine if the currently authenticated user is authorized to perform a given action without writing a dedicated gate that corresponds to the action.

```
Gate::allowIf(fn (User $user) => $user->isAdministrator());
Gate::denyIf(fn (User $user) => $user->banned());
```

Authorization

Creating Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, if your application is a blog, you may have a App\Models\Post model and a corresponding App\Policies\PostPolicy to authorize user actions such as creating or updating posts.

```
php artisan make:policy PostPolicy
```

If you would like to generate a class with example policy methods related to viewing, creating, updating, and deleting the resource, you may provide a --model option when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

Authorization

Registering Policies

The App\Providers\AuthServiceProvider included with fresh Laravel applications contains a policies property which maps your Eloquent models to their corresponding policies. Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given Eloquent model:

```
protected $policies = [
  Post::class => PostPolicy::class,
];
```

The update method will receive a User and a Post instance as its arguments, and should return true or false indicating whether the user is authorized to update the given Post.

```
public function update(User $user, Post $post): bool
{
  return $user->id === $post->user_id;
}
```

Policy Responses

```
public function update(User $user, Post $post): Response
{
  return $user->id === $post->user_id
      ? Response::allow()
      : Response::deny('You do not own this post.');
}
```

```
$response = Gate::inspect('update', $post);
if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

Authorization

Methods Without Models

Some policy methods only receive an instance of the currently authenticated user. This situation is most common when authorizing create actions.

```php
public function create(User $user): bool
{
    return $user->role == 'writer';
}
```

Policy Filters

For certain users, you may wish to authorize all actions within a given policy. This feature is most commonly used for authorizing application administrators to perform any action:

```php
public function before(User $user, string $ability): bool|null
{
    if ($user->isAdministrator()) {
        return true;
    }

    return null;
}
```

Authorizing Actions Using Policies

Via The User Model

The App\Models\User model that is included with your Laravel application includes two helpful methods for authorizing actions: can and cannot. The can and cannot methods receive the name of the action you wish to authorize and the relevant model.

```php
public function update(Request $request, Post $post): RedirectResponse
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }
        // Update the post...
        return redirect('/posts');
    }
```

Authorization

Authorizing Actions Using Policies

Via Controller Helpers

In addition to helpful methods provided to the App\Models\User model, Laravel provides a helpful authorize method to any of your controllers which extend the App\Http\Controllers\Controller base class.

```php
public function update(Request $request, Post $post): RedirectResponse
    {
        $this->authorize('update', $post);
        // $this->authorize('create', Post::class);
        // The current user can update the blog post...

        return redirect('/posts');
    }
```

Via Middleware

Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers. Let's explore an example of using the can middleware to authorize that a user can update a post:

For convenience, you may also attach the can middleware to your route using the can method:

Again, some policy methods like create do not require a model instance.

```php
Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

```php
Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->can('update', 'post');
```

```php
Route::post('/post', function () {
    // The current user may create posts...
})->can('create', Post::class);
```

Authorization

Authorizing Actions Using Policies

Via Blade Templates

When writing Blade templates, you may wish to display a portion of the page only if the user is authorized to perform a given action. For example, you may wish to show an update form for a blog post only if the user can actually update the post.
In this situation, you may use the @can and @cannot directives:

```
@can('update', $post)
    <!-- The current user can update the post... -->
@elsecan('create', App\Models\Post::class)
    <!-- The current user can create new posts... -->
@else
    <!-- ... -->
@endcan

@cannot('update', $post)
    <!-- The current user cannot update the post... -->
@elsecannot('create', App\Models\Post::class)
    <!-- The current user cannot create new posts... -->
@endcannot
```

You may also determine if a user is authorized to perform any action from a given array of actions. To accomplish this, use the @canany directive:

```
@canany(['update', 'view', 'delete'], $post)
    <!-- The current user can update, view, or delete the post... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- The current user can create a post... -->
@endcanany
```

You may pass a class name to the @can and @cannot directives if the action does not require a model instance:

```
@can('create', App\Models\Post::class)
    <!-- The current user can create posts... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- The current user can't create posts... -->
@endcannot
```

Laravel Localization

Laravel provides two ways to manage translation strings. First, language strings may be stored in files within the application's lang directory. Within this directory, there may be subdirectories for each language supported by the application. This is the approach Laravel uses to manage translation strings for built-in Laravel features such as validation error messages:

```
/database
/lang
    /en
        messages.php
    /es
        messages.php
/public
/resources
```

Or, translation strings may be defined within JSON files that are placed within the lang directory. When taking this approach, each language supported by your application would have a corresponding JSON file within this directory. This approach is recommended for applications that have a large number of translatable strings:

```
/resources
   /lang
        en.json
        es.json
        ar.json
```

By default, the Laravel application skeleton does not include the lang directory. If you would like to customize Laravel's language files or create your own, you should scaffold the lang directory via the lang:publish Artisan command.

```
php artisan lang:publish
```

Laravel Localization

You may use the currentLocale and isLocale methods on the App facade to determine the current locale or check if the locale is a given value:

```
use Illuminate\Support\Facades\App;
$locale = App::currentLocale();
if (App::isLocale('en')) {
    // ...
}
```

You may retrieve translation strings from your language files using the __ helper function. If you are using "short keys" to define your translation strings, you should pass the file that contains the key and the key itself to the __ function using "dot" syntax.

```
echo __('messages.welcome');
echo __('I love programming.');
```

If you wish, you may define placeholders in your translation strings. All placeholders are prefixed with a **:**

```
'welcome' => 'Welcome, :name',

echo __('messages.welcome', ['name' => 'Mohammed']);
```

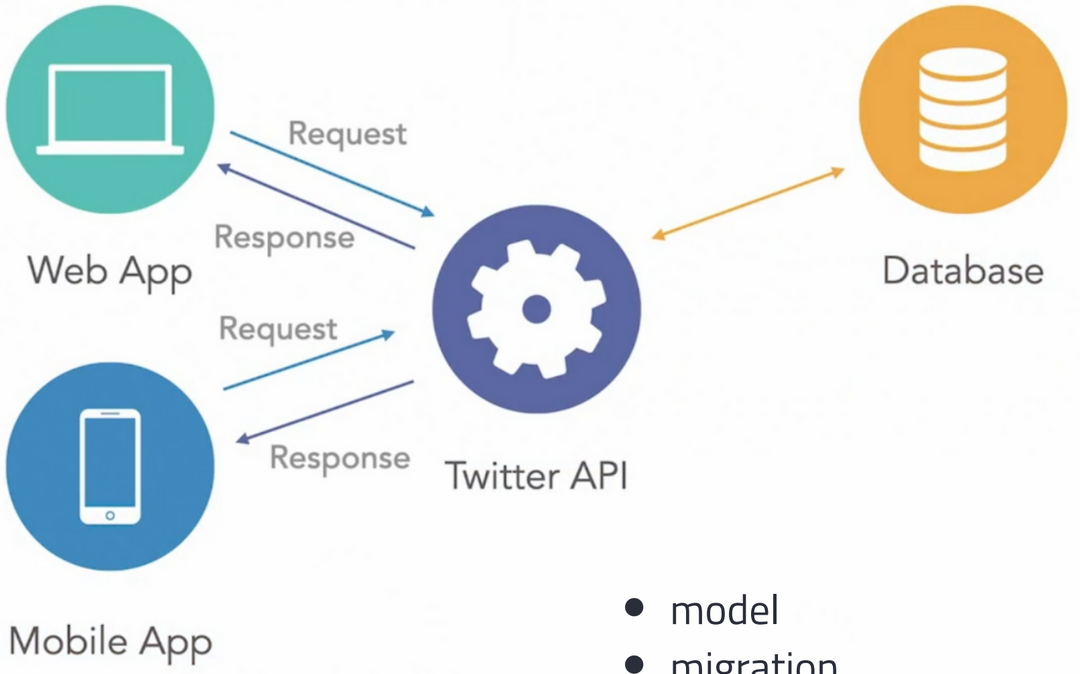
API (Application Programming Interface)

Generating a model, a migration file, a factory, and a seeder for a "Note" entity.

```
php artisan make:model Note -mfs
```

Generating a controller designed for API usage, and this controller will be connected to the "Note" model, allowing you to manage "Note" data through API endpoints.

```
php artisan make:controller NoteController --api --model=Note
```



- model
- migration
- controller
- API resource
- factory
- seeder

API (Application Programming Interface)

API resources are templates where you define how you want the JSON data to be returned back to the user when they send an API request.

```
php artisan make:resource NoteResource
```

When we want to return more than one resource, more than one note, we need to create a resource collection.

```
php artisan make:resource NoteCollection
```

```
// inside NoteResource.php file

public function toArray($request)
{
  return [
    'id' => $this->id,
    'title' => $this->title,
  ];
}
```

Returning the Resources from NoteController:

```
class NoteController
{
    public function index()
    {
        return NoteResource::collection(Note::all());
        // or adding the following
        return new NoteCollection(Note::all());
    }
}
```