Laraplans Documentation

Release 2.1.0

Gerardo J. Báez

May 27, 2020

Contents

This library intends to provide a quick starting point to any app that requires SaaS style plans and subscriptions.

CHAPTER 1

Installation

1.1 Composer

```
$ composer require gerardojbaez/laraplans
```

1.2 Service Provider

Add `Gerardojbaez\Laraplans\LaraplansServiceProvider::class` to your application service providers file: `config/app.php`.

```
'providers' => [
    /**
     * Third Party Service Providers...
     */
    Gerardojbaez\Laraplans\LaraplansServiceProvider::class,
]
```

1.3 Config File and Migrations

Publish package config file and migrations with the following command:

```
php artisan vendor:publish --provider="Gerardojbaez\Laraplans\LaraplansServiceProvider
↪"
```

Then run migrations:

```
php artisan migrate
```

1.4 Traits and Contracts

Add `Gerardojbaez\Laraplans\Traits\PlanSubscriber` trait and `Gerardojbaez\Laraplans\Contracts\Plan` contract to your `User` model.

See the following example:

```
namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Gerardojbaez\Laraplans\Contracts\PlanSubscriberInterface;
use Gerardojbaez\Laraplans\Traits\PlanSubscriber;

class User extends Authenticatable implements PlanSubscriberInterface
{
    use PlanSubscriber;
```

CHAPTER 2

Usage

2.1 Create a Plan

```
use Gerardojbaez\Laraplans\Models\Plan;
use Gerardojbaez\Laraplans\Models\PlanFeature;

$plan = Plan::create([
    'name' => 'Pro',
    'description' => 'Pro plan',
    'price' => 9.99,
    'interval' => 'month',
    'interval_count' => 1,
    'trial_period_days' => 15,
    'sort_order' => 1,
]);

$plan->features()->saveMany([
    new PlanFeature(['code' => 'listings', 'value' => 50, 'sort_order' => 1]),
    new PlanFeature(['code' => 'pictures_per_listing', 'value' => 10, 'sort_order' =>␣
↪5]),
    new PlanFeature(['code' => 'listing_duration_days', 'value' => 30, 'sort_order' =>
↪ 10]),
    new PlanFeature(['code' => 'listing_title_bold', 'value' => 'Y', 'sort_order' =>␣
↪15])
]);
```

2.2 Accessing Plan Features

In some cases you need to access a particular feature in a particular plan, you can accomplish this by using the `getFeatureByCode` method available in the `Plan` model.

Example:

```
$feature = $plan->getFeatureByCode('pictures_per_listing');
$feature->value // Get the feature's value
```

2.3 Create a Subscription

First, retrieve an instance of your subscriber model, which typically will be your user model and an instance of the plan your user is subscribing to. Once you have retrieved the model instance, you may use the `newSubscription` method (available in `PlanSubscriber` trait) to create the model's subscription.

```
use Auth;
use Gerardojbaez\Laraplans\Models\Plan;

$user = Auth::user();
$plan = Plan::find(1);

$user->newSubscription('main', $plan)->create();
```

The first argument passed to `newSubscription` method should be the name of the subscription. If your application offer a single subscription, you might call this `main` or `primary`. Subscription's name is not the Plan's name, it is an *unique* subscription identifier. The second argument is the plan instance your user is subscribing to.

2.4 Subscription resolving

When you use the `subscription()` method (i.e., `$user->subscription('main')`) in the subscribable model to retrieve a subscription, you will receive the latest subscription created of the subscribable and the subscription name. For example, if you subscribe *Jane Doe* to *Free plan*, and later to *Pro plan*, Laraplans will return the subscription with the *Pro plan* because it is the newest subscription available. If you have a different requirement you may use your own subscription resolver by binding an implementation of `Gerardojbaez\Laraplans\Contracts\SubscriptionResolverInterface` to the service container; like so:

```
/**
 * Register the application services.
 *
 * @return void
 */
public function register()
{
    $this->app->bind(SubscriptionResolverInterface::class,␣
↪CustomSubscriptionResolver::class);
}
```

2.5 Subscription's Ability

There are multiple ways to determine the usage and ability of a particular feature in the user's subscription, the most common one is `canUse`:

The `canUse` method returns `true` or `false` depending on multiple factors:

- Feature *is enabled*

- Feature value isn't `0`.
- Or feature has remaining uses available

```
$user->subscription('main')->ability()->canUse('listings');
```

There are other ways to determine the ability of a subscription:

- `enabled`: returns `true` when the value of the feature is a *positive word* listed in the config file.
- `consumed`: returns how many times the user has used a particular feature.
- `remainings`: returns available uses for a particular feature.
- `value`: returns the feature value.

All methods share the same signature: `$user->subscription('main')->ability()->consumed('listings');` .

2.6 Record Feature Usage

In order to efectively use the ability methods you will need to keep track of every usage of usage based features. You may use the `record` method available through the user `subscriptionUsage()` method:

The `record` method accepts 3 parameters: the first one is the feature's code, the second one is the quantity of uses to add (default is `1`), and the third one indicates if the usage should be incremented (`true`: default behavior) or overriden (`false`).

See the following example:

```
// Increment by 2
$user->subscriptionUsage('main')->record('listings', 2);

// Override with 9
$user->subscriptionUsage('main')->record('listings', 9, false);
```

2.7 Reduce Feature Usage

Reducing the feature usage is *almost* the same as incrementing it. In this case we only *substract* a given quantity (default is `1`) to the actual usage:

```
// Reduce by 1
$user->subscriptionUsage('main')->reduce('listings');

// Reduce by 2
$user->subscriptionUsage('main')->reduce('listings', 2);
```

2.8 Clear The Subscription Usage Data

In some cases you will need to clear all usages in a particular user subscription, you can accomplish this by using the `clear` method:

```
$user->subscriptionUsage('main')->clear();
```

2.9 Check Subscription Status

For a subscription to be considered **active** the subscription must have an active trial or subscription's `ends_at` is in the future.

```
$user->subscribed('main');
$user->subscribed('main', $planId); // Check if subscription is active AND using a␣
↪particular plan
```

Alternatively, you can use the following methods available in the subscription model:

```
$user->subscription('main')->isActive();
$user->subscription('main')->isCanceled();
$user->subscription('main')->isCanceledImmediately();
$user->subscription('main')->isEnded();
$user->subscription('main')->onTrial();
```

Caution: **Canceled** subscriptions **with** an active trial or `ends_at` in the future are considered active.

2.10 Renew a Subscription

To renew a subscription you may use the `renew` method available in the subscription model. This will set a new `ends_at` date based on the selected plan and **will clear the usage data** of the subscription.

```
$user->subscription('main')->renew();
```

Caution: Canceled subscriptions with an ended period can't be renewed.

`Gerardojbaez\Laraplans\Events\SubscriptionRenewed` event is fired when a subscription is renewed using the `renew` method.

2.11 Cancel a Subscription

To cancel a subscription, simply use the `cancel` method on the user's subscription:

```
$user->subscription('main')->cancel();
```

By default, the subscription will remain active until the period ends. Pass `true` to *immediately* cancel a subscription.

```
$user->subscription('main')->cancel(true);
```

CHAPTER 3

Events

The following are the events fired by the package:

- `Gerardojbaez\Laraplans\Events\SubscriptionCreated`: Fired when a subscription is created.
- `Gerardojbaez\Laraplans\Events\SubscriptionRenewed`: Fired when a subscription is renewed using the `renew()` method.
- `Gerardojbaez\Laraplans\Events\SubscriptionCanceled`: Fired when a subscription is canceled using the `cancel()` method.
- `Gerardojbaez\Laraplans\Events\SubscriptionPlanChanged`: Fired when a subscription's plan is changed; it will be fired once the `PlanSubscription` model is saved. Plan change is determined by comparing the original and current value of `plan_id`.

CHAPTER 4

Eloquent Scopes

```php
use Gerardojbaez\Laraplans\Models\PlanSubscription;

// Get subscriptions by plan:
$subscriptions = PlanSubscription::byPlan($plan_id)->get();

// Get subscription by user:
$subscription = PlanSubscription::byUser($user_id)->first();

// Get subscriptions with trial ending in 3 days:
$subscriptions = PlanSubscription::findEndingTrial(3)->get();

// Get subscriptions with ended trial:
$subscriptions = PlanSubscription::findEndedTrial()->get();

// Get subscriptions with period ending in 3 days:
$subscriptions = PlanSubscription::findEndingPeriod(3)->get();

// Get subscriptions with ended period:
$subscriptions = PlanSubscription::findEndedPeriod()->get();

// Exclude subscriptions which are canceled:
$subscriptions = PlanSubscription::excludeCanceled()->get();

// Exclude subscriptions which are immediately canceled:
$subscriptions = PlanSubscription::scopeExcludeImmediatelyCanceled()->get();
```