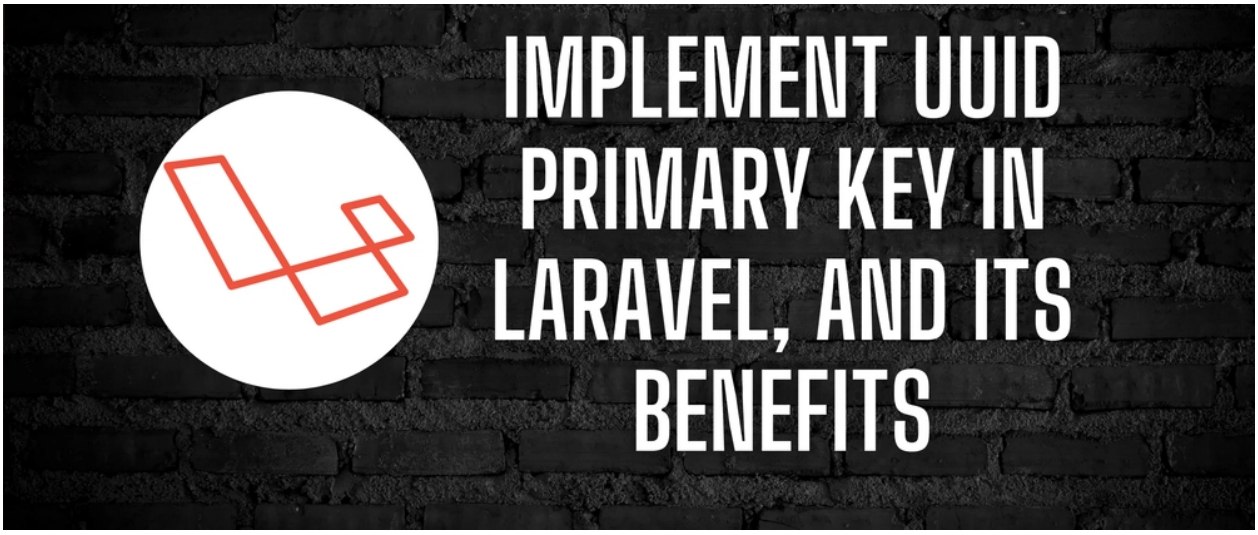
Implement UUID primary key in Laravel, and its benefits

Adnan Babakan (he/him) 21 lug



Hey, DEV.to community!

It's been a long time since I last wrote here.

During the time that I've been absent, I've been working on some large-scale projects in which I've seen how UUIDs can benefit your application in multiple ways.

I've been using UUIDs as my primary key structure for a fair amount of time now but the real advantages have shown themselves after finishing the projects and I am glad I've decided to do so.

I primarily use Laravel as my back-end and love using MySQL (I know there are lots of other RDBMS/DBMS out there but I love MySQL and feel comfortable with it XD). Although when using MySQL or many other databases your primary key is set as an AI (auto-incremental) and the data type is an integer, you may use UUID with a few tweaks and believe me it's worth the hassle.

Me when coding.

Benefits

Enumeration exploit fix

An enumeration exploit is when your data is predictable. Let's say for instance you upload a video on YouTube which is hidden from the public and can only be viewed by those whom you shared the link with. Well, Let's look at a YouTube link:

```
https://www.youtube.com/watch?v=3wVTmlD86a
```

My favorite song BTW

As you can see the address ends in `v=3wVTmlD86a` which indicates the (let's say) ID of the video. So imagine YouTube used a numerical index and incremented it one by one for each video uploaded. Then what you might ask. In that case, someone could start from number 1 and try every link using a custom-made program to extract all the available videos on YouTube and your link-only video would be

exposed as well.

This is called enumeration exploiting which can cause serious data leaks depending on what your program is supposed to do.

The algorithm YouTube uses to make IDs for their videos is custom, but the purpose of my description is the same for UUIDs.

Given a UUID such as `700234f5-0e45-452e-ae3a-70b4b3d024e1`, you wouldn't know what UUID is before or after this since there is no order. As you can see this can solve the enumeration exploit in your application.

Horizontally expandable databases

Let's imagine that you have two database instances of the same system that you want to expand or just make a new cluster for some reason and then merge both instances. Given that the database structure is the same and you have complex relations in your database, you might find it difficult to merge data without manually adjusting some values or using an automated task.

As you can see in the simple diagram above of an abstract database, you might have a table of orders, which is related to both a product and a user.

Now let's think about this: In each of your database instances, you only have 10 users and 10 products, and 10 orders which are numerically indexed from 1 to 10. Let's say in our first database instance, order 2 is related to user ,3 whose name is `Adnan Babakan`. In the other instance, order 2 is related to use 3 as well but their name is `Arian Amini`. Even if we import the orders properly and their IDs aren't referenced from other tables and their indexes are reassigned from 1 to 20, since the users are reindexed as well now our orders reference wrong users!

I suppose you see the problem and solution now. If we use UUIDs, since they are globally unique there will be no collision between our data and we can merge as many instances as we want with no problem.

Apart from merging two databases, UUIDs make it easier to manage

a cluster-based database architecture as well with multiple database instances running at the same time.

Downsides

There are two main downsides to using UUIDs in databases. First is the size it takes in your database and second is the insert problem.

Since UUIDs aren't ordered performing an insert operation becomes costly because your record will be inserted somewhere random rather than at the end of your table when using a numerical index system.

Even though these disadvantages are present, it is forgivable as space isn't that expensive and the insert problem isn't such a big deal for most cases.

There is a way to avoid the insert problem which is using the same numerical index system as before but assigning a separate UUID for each record as well so you benefit from the points aforementioned.

Using UUIDs in Laravel

In this section of the article, I will discuss how you can implement UUIDs in your Laravel application and the problems I faced and how I solved them. Remember that UUIDs are NOT for Laravel/MySQL only and you may use them in any other programming language, framework and database you like.

Keep in mind this is my preferred way of implementing such an approach and you might find a more efficient way, and let me know in the comments if you did so.

Migrations

The first thing you need to do is know how to define your migrations so your tables use UUID.

Fortunately, Laravel comes packed with many good methods and stuff to help you with your goal.

First of all, let's change how we define our PK (primary key) in a migration. This is how you usually define your PK in a Laravel migration:

The code above is equivalent of:

```
$table->integer('id')->primary();
```

Unfortunately, there is no shorthand method for creating a UUID-based PK in migrations. But it isn't that much of a work either:

```
$table->uuid('id')->primary();
```

TADA! Now your table will have a column named `id` which holds UUIDs.

References and relations

Now let's talk about relations. A relation is when you want to reference a different record from another table or the same table. Referencing is done using the target record's id since it is the unique key that you are sure will never be duplicated.

Imagine you want to reference a user in your table, this is how you define a normal relation in your migrations:

```
$table->foreignId('user_id')->constrained();
```

Keep in mind the `constrained()` method is a shorthand for:

```
$table->foreignId('user_id')->references('id')->on('users');
```

Laravel gives you the power of the shorthand at the cost of following naming conventions. So when your foreign id column is named `user_id` and `constrained()` is called afterwards, Laravel knows you mean that this `user_id` will reference the `id` column on the `users` table. If it is not the case you must define it using the `references()` and the `on()` methods to specifically determine what you mean.

To reference a foreign id which is a UUID, you may use the `foreignUuid()` method instead of `foreignId()` method. The rest is the same and you can apply `constrained()` to a foreign UUID reference as well. This is how you reference a user record given that the `users` table utilizes UUID as its PK structure.

```
$table->foreignUuid('user_id')->constrained();
```

Or more elaborately:

```
$table->foreignUuid('user_id')->references('id')->on('users');
```

Models

Now that you've successfully defined your migrations, there is a small tweak you need to make on your models.

A model's defaults tell it to utilize integer-based ids and we should

tell the model that it is not the case.

First, add a private property named `$keyType` to your model and set its value as `'string'`:

```
protected $keyType = 'string';
```

This tells the model that your key is a type of string and not an integer (UUIDs are strings).

The second thing to do is tell the model not to use the incrementing system for this type of key which is done by setting the `$incrementing` property as `false`:

```
public $incrementing = false;
```

After doing these your model know how your PK works perfectly.

But wait! Is there a problem you face when creating a new record telling you that `id` cannot be null and doesn't have a default value?

To solve this issue either you have to define a UUID each time you create a new record such as below:

```
$new_user = new User();
$new_user->id = Str::uuid();
$new_user->username = 'Adnan';
$new_user->password = Hash::make('helloWorld');
$new_user->save();
```

The `Str` class is imported from `Illuminate\Support\Str`. The `Str::uuid()` helper creates a new UUID for you.

Or you can use model events to tell Laravel how to create an ID for your model when creating the record on the database.

This can be done simply by using closure events in your `booted` method. Add a static method called `booted` to your model:

```
public static function booted() {

}
```

Then you can use an event called `creating` inside of it and tell it to make a UUID and assign it to your model's id:

```
public static function booted() {
    static::creating(function ($model) {
        $model->id = Str::uuid();
    });
}
```

You can read more about model events at https://laravel.com/docs/10.x/eloquent#events-using-closures

If you are using the old `boot` method, remember to call the parent's `boot` method so it is not overwritten:

```
public static function boot() {
    parent::boot();

    static::creating(function ($model) {
        $model->id = Str::uuid();
    });
}
```

If you've followed so far, your `User` model or any other model should look something like this:

```
class User extends Model {
    use HasFactory;

    ...

    protected $keyType = 'string';

    public $incrementing = false;

    ...

    public static function boot() {
        parent::boot();

        static::creating(function ($model) {
            $model->id = Str::uuid();
        });
    }
}
```

Solving the Sanctum problem

If you try to use Sanctum to issue your tokens you might realize that you face an error telling you that your constraints fail. This is because your `personal_access_tokens` table is designed to reference integer-based foreign ids (this table uses morphs as it should be able to reference multiple types of models).

In case you already have a project running you should change the `tokenable_id` column's type in your `personal_access_tokens` table using a new migration.

First, create a new migration:

```
php artisan make:migration change_tokenable_id_type_in_personal_access_t
```

Then in your migration use the `change()` method on the `tokenable_id` column:

```
$table->foreignUuid('tokentable_id')->change();
```

This changes the data type of `tokenable_id` to reference UUIDs instead of integers.

To make our database structure unified I'd like to make my token records use UUID as their ids as well so add the instruction below as well:

```
$table->uuid('id')->primary()->change();
```

If you receive an error running this migration install the `doctrine/dbal` package using composer.

```
composer require doctrine/dbal
```

Although the method described above is the proper way to change the data type of the `tokenable_id` column, there is a simpler way of doing so if you are just starting a new project or you don't care about your current data and are willing to refresh your migration (which will delete your data).

Just open the `2019_12_14_000001_create_personal_access_tokens_table.php` file

(the date at the beginning might differ) and change:

```
$table->morphs('tokenable');
```

to:

```
$table->uuidMorphs('tokenable');
```

and change:

to:

```
$table->uuid('id')->primary();
```

and then run:

or:

```
php artisan migrate:fresh
```

if you've already run your migrations before.

Now that our `personal_access_tokens` can reference UUIDs and its id is also a UUID we need to make Laravel know how to treat our tokens as Sanctum uses a default model to handle these records.

First, let's create a new model that will be the model Sanctum uses since we've changed the structure:

```
php artisan make:model PersonalAccessToken
```

Now open the created model's file and instead of extending the `Model` class, extend the class by `Laravel\Sanctum\PersonalAccessToken`:

```
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;

class PersonalAccessToken extends SanctumPersonalAccessToken
{

}
```

Since our class has the same name as the class used by Sanctum's default PersonalAccessToken class, you should either define an alias when importing it or use absolute naming after the `extend` keyword like: `\Laravel\Sanctum\PersonalAccessToken`

We have to make sure Laravel knows how this model works just as before. So we should set `$keyType`, `$incrementing` and define an event to create a UUID for the id of each record.

This is how your final `PersonalAccessToken` model should look like:

```
<?php

namespace App\Models;

...
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Support\Str;
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;
...

class PersonalAccessToken extends SanctumPersonalAccessToken
{
```

```
    use HasFactory;

    public $keyType = 'string';

    public $incrementing = false;

    ....

    public static function boot()
    {
        parent::boot();

        static::creating(function ($model) {
            $model->id = (string) Str::uuid();
        });
    }

    ...
}
```

The last step is to define this model as the model used by Sanctum. This step can be done by using the `Sanctum::usePersonalAccessTokenModel()` method. This method should be called in a provider file. I prefer `AppServiceProvider` which is located at `app/Providers/AppServiceProvider.php`.

Open the provider file and add the code below inside of the `boot` method:

```
Sanctum::usePersonalAccessTokenModel(PersonalAccessToken::class);
```

Remember to import `Sanctum` and your custom `PersonalAccessToken` classes properly. The `Sanctum` class is imported from

`Laravel\Sanctum\Sanctum` and your custom `PersonalAccessToken` is imported from `App\Models\PersonalAccessToken`.

And now you are done implementing UUIDs in your Laravel application!

I hope you enjoyed this article and it helped you do what you wanted to do. Kindly let me know if there are any mistakes in this article or if you know a better way to do something.

BTW! Check out my free Node.js Essentials E-book here: