WEB 306 – PHP: Databases and Framework

Day 12 – User Authentication

1. User Accounts

Most web applications would require users to sign up for authenticated, password protected user accounts so that their data can be associated with them and so that it can be kept private. Laravel makes registering user accounts a breeze (in comparison to doing so from scratch at least).

Today we will be adding user accounts to our Artists app so that artists, artworks and galleries can be associated with specific users.

Laravel Resources

1. Laravel Website
2. Laravel Documentation
3. Laracasts
4. Laravel News
5. Scotch.io
6. Stack Overflow

2. Models

2.1 User Model

When you install Laravel it already provides you with a `User` model. A lot of the required code for the User model is already provided but there are some things which we will need to add in order for the `User` model to work with our specific application's needs.

```
namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
```

In order to associate data such as artists, artworks and galleries with specific users we need to use the namespaces of those models in the `User` model.

```
use App\Models\Artist;
use App\Models\Artwork;
use App\Models\Gallery;

class User extends Authenticatable {
    use HasFactory, Notifiable;
```

By default, Laravel uses a person's email address as their username as well. If we want to add the capability for the user to have a distinct username which they can use to log in instead of or in

addition to their email address then it is necessary to add "username" as a fillable field of the `User` model.

```
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['username', 'name', 'email', 'password'];
```

Laravel models use a property called `$hidden` which is an array of strings which specify fields which should not be included when models are exported in formats such as JSON. By default, the `User` class which Laravel provides us with sets the `password` and `remember_token` fields to be hidden. It is not necessary for us to add anything to this array for our purposes, but if you were storing any particularly sensitive data then it would make sense to keep it hidden from exported data.

```
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password', 'remember_token'];
```

Laravel models use a property called `$casts` to which is an array that lists attributes which should be cast to common data types. Supported types are `integer`, `real`, `float`, `double`, `decimal:<digits>`, `string`, `boolean`, `object`, `array`, `collection`, `date`, `datetime`, `timestamp`, `encrypted`, `encrypted:object`, `encrypted:array`, and `encrypted:collection`. By default, the User class which Laravel provides us with sets the `email_verified_at` attribute to cast to the `datetime` type. It is not necessary for us to add anything to this array but depending on the scenarios you want to be able to use an attribute in, it would make sense to cast certain attributes to different data types.

```
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = ['email_verified_at' => 'datetime'];
```

In order to associate our application's data with users, we do need to add some methods to the `User` model. We would want to create has-many relationships between our user and our other data models so it is necessary to include corresponding methods which return the `hasMany()` method with the related model's class.

```
    /**
     * Get the artists for the user.
     */
    public function artists() {
        return $this->hasMany(Artist::class);
    }

    /**
     * Get the artworks for the user.
```

```
     */
    public function artworks() {
        return $this->hasMany(Artwork::class);
    }

    /**
     * Get the galleries for the user.
     */
    public function galleries() {
        return $this->hasMany(Gallery::class);
    }
}
```

We do not need to include a method for the `Bio` model, because the only context it is created in is in relation to an Artist model due to their one-to-one relationship.

2.2 Artist, Artwork and Gallery Models

It is also necessary to create inverse relationships in those models by using the `User` namespace and creating corresponding methods called `user()` which return the `belongsTo()` method which will take the `User` class as a parameter.

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use App\Models\Bio;
use App\Models\Artwork;
use App\Models\User;

class Artist extends Model {
    // Existing Artist class code...

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

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use App\Models\Artist;
use App\Models\Gallery;
use App\Models\User;

class Artwork extends Model {
```

```
    // Existing Artwork class code...

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

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use App\Models\Artwork;
use App\Models\User;

class Gallery extends Model {
    // Existing Gallery class code...

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

3. Migrations

3.1 User Migration

Laravel also provides two pre-written migrations files, one of which is for migrating a users table and the other is for migrating a password resets table. We wouldn't need to alter the password resets migration but we would want to make some small adjustments to the users migration.

The primary change we would need to make would be to include a field for "username" which we added to the `User` model. We would also want to increase the maximum character limit to `255` for strings which do not use the `unique()` method.

The `unique()` method ensures that the value of a given field is kept unique among the all rows in the database. This should be used for fields such as usernames and email addresses. Unique string fields such as the email field can not have a length longer than `254`.

PHP Resources

1. PHP Website
2. XAMPP
3. Documentation
4. W3Schools
5. CSS Tricks
6. Scotch.io
7. Stack Overflow

The `remember_token()` method creates a nullable VARCHAR column which is used to store a token which remembers the user if they choose the "remember me" option when logging into the web app.

```
class CreateUsersTable extends Migration {
    public function up() {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('username')->unique();
            $table->string('name', 255);
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password', 255);
            $table->rememberToken();
            $table->timestamps();
        });
    }

    public function down() {
        Schema::dropIfExists('users');
    }
}
```

3.2 Artist, Artwork and Gallery Migrations

Because we are creating has-many relationships between our `User` model and our other models, we need to create new migrations for each model to add a "user_id" column to their corresponding tables.

Use the following artisan command to generate a migration which is set up to edit the artists table.

```
php artisan make:migration add_user_id_to_artists_table
```

The code in the `up()` method will add the "user_id" column to the "artists" table. The code in the `down()` method will drop the column from the table if you use an artisan rollback command.

```
class AddUserIdToArtistsTable extends Migration {
    public function up() {
        Schema::create('artists', function (Blueprint $table) {
            $table->bigInteger('user_id')->unsigned()->nullable();
        });
    }

    public function down() {
        Schema::create('artists', function (Blueprint $table) {
            $table->dropColumn('user_id');
        });
    }
}
```

We would also want to create migrations to add "user_id" columns to the other tables as well.

```
php artisan make:migration add_user_id_to_artworks_table
```

```
class AddUserIdToArtworksTable extends Migration {
    public function up() {
        Schema::create('artworks', function (Blueprint $table) {
            $table->bigInteger('user_id')->unsigned()->nullable();
        });
    }

    public function down() {
        Schema::create('artworks', function (Blueprint $table) {
            $table->dropColumn('user_id');
        });
    }
}
```

```
php artisan make:migration add_user_id_to_galleries_table
```

```
class AddUserIdToGalleriesTable extends Migration {
    public function up() {
        Schema::create('galleries', function (Blueprint $table) {
            $table->bigInteger('user_id')->unsigned()->nullable();
        });
    }

    public function down() {
        Schema::create('galleries', function (Blueprint $table) {
            $table->dropColumn('user_id');
        });
    }
}
```

3.3 App Service Provider

Before migrating our users table migrations, it is necessary to add two lines of code to a file called `AppServiceProvider.php` which is in the `/app/Providers` directory. First we need to use the namespace `Illuminate\Support\Facades\Schema`.

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

// This is not provided by default and MUST be added for
// Laravel to migrate user migrations
use Illuminate\Support\Facades\Schema;

class AppServiceProvider extends ServiceProvider {
```

Next we need to specify a default string length for table columns of `191` using the static `defaultStringLength()` method of the `Schema` class. This is to set the default string length for unique fields. If we do not add this line we will receive an error message when we are migrating.

```
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register() {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot() {
        // This is not provided by default and MUST be added for
        // Laravel to migrate user migrations
        Schema::defaultStringLength(191);
    }
}
```

After adding these two lines, we should be able to safely run our migrations using the following artisan command. Remove any migrations which you have already migrated or which you don't want to migrate from the migrations folder before running the command.

```
php artisan migrate
```

4. Authentication Routes and Blade Templates

4.1 The Laravel UI Package

Laravel can automatically generate routes, Blade files and controllers which are associated with all of the user authorization processes. Using these automatically generated routes and Blade files is optional but they do provide everything you need to set up user authorization easily. They can also be customized fairly easily.

Laravel 8 introduced a new automated authentication system called Jetstream. This new authentication system automates the creation of additional features such as two-factor authentication as well, but it is much more complex to customize than the previous automated authentication system. It is almost a whole framework on its own, makes use of different templating systems than normal Blade templates and has its own documentation.

To keep things simpler, we are going to install a Composer package called Laravel UI which will allow us to use the more basic but easier to customize automated authentication system which was used in Larravel 7 with Laravel 8. To install this composer package we need to enter the following command:

```
composer require laravel/ui
```

To automatically generate the authorization routes and blade files using the front-end framework, Bootstrap, use the following artisan command.

```
php artisan ui bootstrap --auth
```

If you would prefer to use Vue or React then you can use the following commands instead.

```
php artisan ui vue --auth
```

```
php artisan ui react --auth
```

Then it is necessary to enter the following NPM command to download and compile front-end assets such as Bootstrap, Vue or React.

```
npm install && npm run dev
```

4.2 Authorization Routes

When one of the above artisan commands is run it will add two new route methods to our `routes/web.php` file.

```
Route::resources([
    'artists' => 'App\Http\Controllers\ArtistController',
    'bios' => 'App\Http\Controllers\BioController',
    'artworks' => 'App\Http\Controllers\ArtworkController',
    'galleries' => 'App\Http\Controllers\GalleryController',
]);

Auth::routes();

Route::get('/home', [App\Http\Controllers\HomeController::class,
    'index'])->name('home');
```

The `Auth::routes()` method automatically generates routes which are related to the necessary authorization proccesses. The "home" route points to a newly created `HomeController` which directs the user to a home page which notifies them of whether they are logged in or not.

4.3 Authorization Blade Templates

Among the Blade templates which artisan generated is a new base template which includes the necessary code to allow users to register, login, and logout. We won't use this template as it is a bit overcomplicated and it doesn't work with our site's layout and CSS. Instead we will add register, login and logout buttons to our base template.

The `@auth` keyword is used as a conditional which checks to see if the user is logged in. We want to hide our application's main pages from users who are not logged in so we will put the navigation items which link to them inside of an `@auth` statement.

```
<nav>
@auth
<ul>
    <li><a href="{{ url('/artists') }}">Artists</a></li>
    <li><a href="{{ url('/artists/create') }}">Add Artist</a></li>
    <li><a href="{{ url('/artworks') }}">Artworks</a></li>
    <li><a href="{{ url('/artworks/create') }}">Add Artwork</a></li>
```

```
    <li><a href="{{ url('/galleries') }}">Galleries</a></li>
    <li><a href="{{ url('/galleries/create') }}">Add Gallery</a></li>
  </ul>
  @endauth
```

The `@guest` keyword is used as a keyword which acts as the inverse of the `@auth` keyword and checks to see if the user is ***not*** logged in. If the user is not logged in then we will display login and register buttons.

```
  <ul class="right-nav">
    @guest
    <li><a href="{{ route('login') }}">Login</a></li>
    <li><a href="{{ route('register') }}">Register</a></li>
```

We can also use the `@else` keyword in conjunction with the `@auth` and `@guest` keywords to provide a fallback. If the user is logged in then we will display the user's name using `Auth::user()->name` and a logout button. Loggin out is a bit more complicated. It is necessary to use a form to log the user out. The form only needs a CSRF token and a submit button though.

```
    @else
    <li><a href="#">Logged in as {{ Auth::user()->name }}</a>

        <form action="{{ route('logout') }}" method="post">
            @csrf

            <button type="submit">Logout</button>
        </form>
    </li>
    @endguest
  </ul>
  </nav>
```

Aside from the base template, we will use the other Blade templates which artisan generated. However, because we are not using the default base ttemplate we will need to change the `@extends` statement to use our base template instead.

```
  @extends('layouts.base')
```

The register and login templates include froms which allow the user to register for an account or log into their account. Because Laravel uses the email field as the username as well by default, it is necessary for us to add a label and input for our username field to these templates.

```
  <form action="{{ route('register') }}" method="post">
    @csrf

    <div class="form-group row">
        <label for="username" class="col-sm-4 col-form-label
            text-md-right">{{ __('Username') }}</label>

        <div class="col-md-6">
            <input id="username" type="text" class="form-control"
                name="username" value="{{ old('username') }}">
```

```
            @if ($errors->has('username'))
            <span class="invalid-feedback">
                <strong>{{ $errors->first('username') }}</strong>
            </span>
            @endif
        </div>
    </div>
```

5. Email Settings

When a user asks to reset their password they would be sent an email with a unique URL which would bring them to a page where they can reset their password. The process of setting up an email server would be complex and would be different depending on which hosting providor you are using.

To keep things simpler, we will simply save the emails as plain text LOG files in the **/storage/logs** directory. To do so, it is necessary to change the **MAIL_DRIVER** environment variable in the **.env** file.

```
MAIL_MAILER=log
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

It is also necessary to change the second parameter of the **env()** function of the **'default'** key to **'log'** in the **mail.php** file which is in the **/config** folder.

```
'default' => env('MAIL_MAILER', 'log'),
```

When the password reset emails are sent they will be stored in the **/storage/logs** directory and will look roughly like this. The email text will be stored in a log file with a lot of other logs which are unrelated to the email so you may have to scroll very far down before you find the email.

```
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable

[Artists](http://localhost)

# Hello!

You are receiving this email because we received a password reset
request for your account.

Reset Password: http://localhost/password/
reset/8e0e8e87b4206826b75974da2c9f57bc2e25a6d80aace89f03af603e517ace48

If you did not request a password reset, no further action is
required.

Regards,Artists
```

```
If you're having trouble clicking the "Reset Password" button, copy
and paste the URL below into your web browser:
[http://localhost/password/reset/

© 2020 Artists. All rights reserved.
```

6. Authentication Controllers

6.1 Registration Controller

Laravel also provides us with several pre-written controller files which are located in `/app/Http/Controllers/Auth` which already have the required code to manage creating and authenticating user accounts. However, we would normally want to customize these controllers to work for our needs. The first controller we will want to change will be the `RegisterController` which is used to manage the process fo registering new users.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use App\Models\User;
use Illuminate\Foundation\Auth\RegistersUsers;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;

class RegisterController extends Controller {
   use RegistersUsers;
```

We can change the `$redirectTo` property to alter where the controller redirects to after it has finished registering a new user. We will leave the default value for the `RegistrationController`.

```
    /**
     * Where to redirect users after registration.
     *
     * @var string
     */
    protected $redirectTo = RouteServiceProvider::HOME;

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct() {
         $this->middleware('guest');
    }
```

We need to add our "username" field to the list of fields which are proccessed by the controller's validator method. We can also adjust the validation requirements of the controller here.

```
    /**
     * Get a validator for an incoming registration request.
     *
     * @param array $data
     * @return \Illuminate\Contracts\Validation\Validator
     */
    public function validator(array $data) {
        return Validator::make($data, [
            'username' => ['required', 'string', 'max:255',
                'unique:users'],
            'name' => ['required', 'string', 'max:255'],
            'email' => ['required', 'string', 'email', 'max:255',
                'unique:users'],
            'password' => ['required', 'string', 'min:8',
                'confirmed'],
        ]);
    }
```

We also need to define the username property when a new instance of the `User` class is created in the below method.

```
    /**
     * Create a new user instance after a valid registration.
     *
     * @param array $data
     * @return \App\User
     */
    public function create(array $data) {
        return User::create([
            'username' => $data['username'],
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => $data['password'],
        ]);
    }
```

Laravel's authorization controllers would also return a view to display. In this case, the view will display a registration form. Laravel's authorization controllers specify views to display by default. However, in order to specify a custom view to display we need to overwrite the default `showRegistrationForm()` method. The name of this view will be `auth.register`. We can also pass any variables we like using this method.

```
    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showRegistrationForm() {
        return view('auth.register', [
            'title' => 'Artists: Register'
        ]);
```

```
    }
}
```

6.2 Login Controller

Next we will alter the `LoginController` which is used to manage the process of authenticating user accounts and logging them in.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller {
    use AuthenticatesUsers;
```

We can change the `$redirectTo` property to alter where the controller redirects to after a user logs in.

```
    /**
     * Where to redirect users after login.
     *
     * @var string
     */
    protected $redirectTo = '/artists';

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct() {
        $this->middleware('guest')->except('logout');
    }
```

In order to specify a custom view to display we need to overwrite the default `showLoginForm()` method. The name of this view will be `auth.login`. We can also pass any variables we like using this method.

```
    /**
     * Show the application's login form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showLoginForm() {
        return view('auth.login', [
            'title' => 'Artists: Login'
        ]);
```

```
    }
```

Because Laravel uses the email field as the username by default, in order to use a different field as the username, such as our "username" field, it is necessary to overwrite the default `username()` method to return the name of the field you wish to use as the username.

```
    /**
     * Get the login username to be used by the controller.
     *
     * @return string
     */
    public function username() {
        return 'username';
    }
}
```

6.3 Forgot Password Controller

The `ForgotPasswordController` controller is used to allow users to send an email to users when they forget their password with a link to a page where they can reset their password.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\SendsPasswordResetEmails;

class ForgotPasswordController extends Controller {
    use SendsPasswordResetEmails;

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct() {
        $this->middleware('guest');
    }
```

In order to specify a custom view to display we need to overwrite the default `showLinkRequestForm()` method. The name of this view will be `auth.passwords.email`. We can also pass any variables we like using this method.

```
    /**
     * Display the form to request a password reset link.
     *
     * @return \Illuminate\Http\Response
     */
    public function showLinkRequestForm() {
        return view('auth.passwords.email', [
            'title' => 'Artists: Forgot Password'
        ]);
```

```
    }
}
```

6.4 Reset Password Controller

After the user clicks the link in the email which they were sent the `ResetPasswordController` controller is used to allow users to reset their password.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use Illuminate\Foundation\Auth\ResetsPasswords;

// This is not provided by default and MUST be added to customize the
// view file which is returned
use Illuminate\Http\Request;

class ResetPasswordController extends Controller {
    use ResetsPasswords;

    /**
     * Where to redirect users after resetting their password.
     *
     * @var string
     */
    protected $redirectTo = '/login';

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct() {
        $this->middleware('guest');
    }
```

In order to specify a custom view to display we need to overwrite the default `showResetForm()` method. The name of this view will be `auth.passwords.reset`. We can also pass any variables we like using this method. We must pass `$token` and `$request->email` which are trasferred to the page through the URL the user was sent as variables.

```
    /**
     * Display the password reset view for the given token.
     *
     * @param \Illuminate\Http\Request  $request
     * @param string|null  $token
     * @return \Illuminate\Contracts\View\Factory|\Illuminate\View\View
     */
    public function showResetForm(Request $request, $token = null) {
```

```
        return view('auth.passwords.reset', [
            'token' => $token,
            'email' => $request->email,
            'title' => 'Artists: Reset Password'
        ]);
    }
}
```

6.5 Verification Controller

The `VerificationController` controller is used for handling email verification.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use Illuminate\Foundation\Auth\VerifiesEmails;

// This is not provided by default and MUST be added to customize the
// view file which is returned
use Illuminate\Http\Request;

class VerificationController extends Controller {
    use VerifiesEmails;

    /**
     * Where to redirect users after resetting their password.
     *
     * @var string
     */
    protected $redirectTo = RouteServiceProvider::HOME;

    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct() {
        $this->middleware('auth');
        $this->middleware('signed')->only('verify');
        $this->middleware('throttle:6,1')->only('verify', 'resend');
    }
```

In order to specify a custom view to display we need to overwrite the default `show()` method. The name of this view will be `auth.verify`. We can also pass any variables we like using this method.

```
    /**
     * Show the email verification notice.
     *
     * @param \Illuminate\Http\Request  $request
```

```
     * @return \Illuminate\Http\Response
     */
    public function showResetForm(Request $request, $token = null) {
        return $request->user()->hasVerifiedEmail()
        ? redirect($this->redirectPath())
        : view('auth.passwords.reset', [
            'title' => 'Artists: Verify Email'
        ]);
    }

}
```

6.6 Artist, Artwork and Gallery Controllers

6.6.1 Artist Controller

Whenever a user adds a new artist, artwork or gallery to thet database we want that database entry to be associated with the user. In order to do so, we need to import the `Illuminate\Support\Facades\Auth` namespace so that we have access to information about the current user.

```
namespace App\Http\Controllers;

use App\Http\Requests\ArtistRequest;

// MUST use this namespace to have access to Auth class/current user
use Illuminate\Support\Facades\Auth;

use App\Artist;
use App\Artwork;

use URL;

class ArtistController extends Controller {
```

We then need to set the `$user_id` property to the ID of the currently logged in user by using the static `id()` method of the `Auth` class.

```
public function store(ArtistRequest $request) {
    $artist = new Artist;
    $artist->name = $request->name;
    $artist->styles = $request->styles;

    $artist->user_id = Auth::id();

    // Image upload code...

    $artist->save();
```

6.6.2 Artwork Controller

```
namespace App\Http\Controllers;
```

```
use App\Http\Requests\ArtworkRequest;

// MUST use this namespace to have access to Auth class/current user
use Illuminate\Support\Facades\Auth;

use App\Artwork;
use App\Artist;
use App\Gallery;

use URL;

class ArtistController extends Controller {
```

```
public function store(ArtworkRequest $request) {
   $artwork = new Artwork;
   $artwork->title = $request->title;
   $artwork->statement = $request->statement;
   $artwork->artist_id = $request->artist_id;

   $artwork->user_id = Auth::id();

   // Image upload code...

   $artwork->save();
```

6.6.3 Gallery Controller

```
namespace App\Http\Controllers;

use App\Http\Requests\ArtworkRequest;

// MUST use this namespace to have access to Auth class/current user
use Illuminate\Support\Facades\Auth;

use App\Gallery;
use App\Artwork;

use URL;
```

```
public function store(ArtworkRequest $request) {
   $gallery = new Gallery;
   $artwork->title = $request->title;

   $artwork->user_id = Auth::id();

   $gallery->save();
```