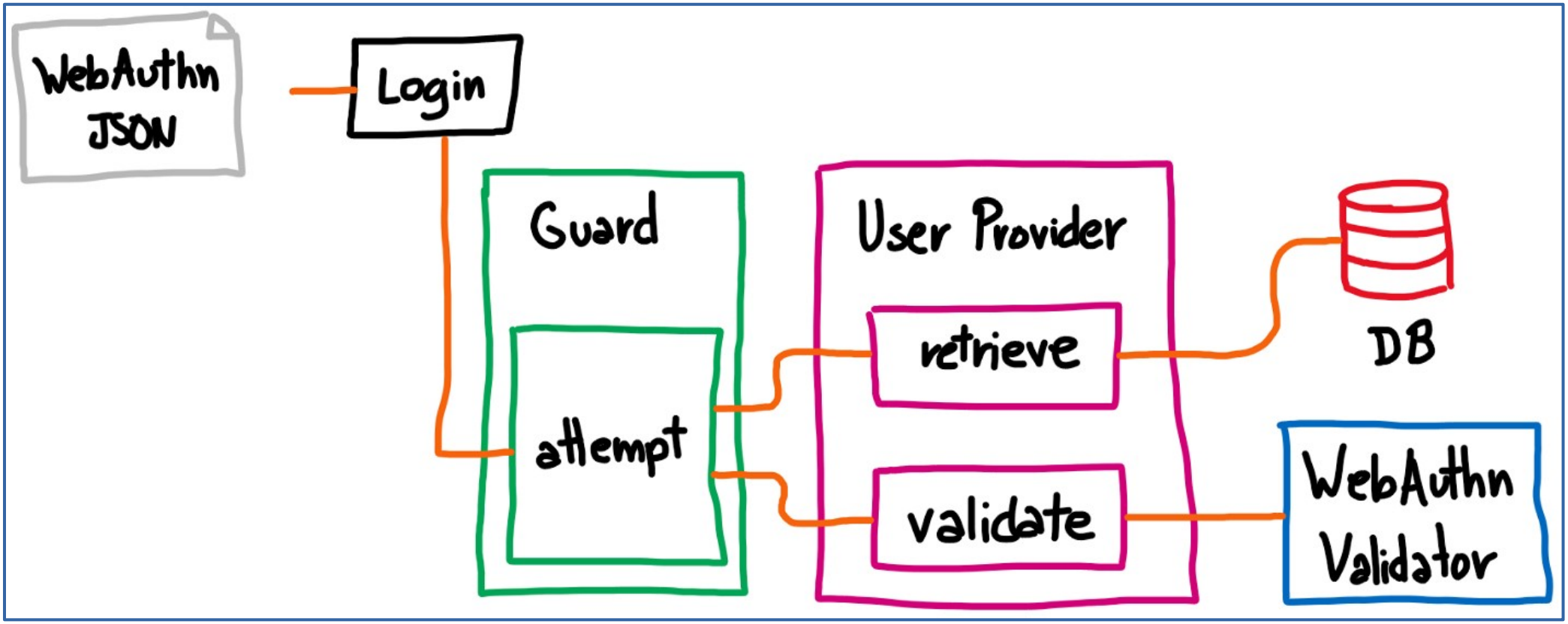
UECS3294 ADVANCED WEB APPLICATION DEVELOPMENT
CHAPTER 6 : AUTHENTICATION AND AUTHORIZATION

LOO YIM LING
ylloo@utar.edu.my

Authentication and Authorization

1) Authentication options
2) Authorization
3) Role-based Access Control (RBAC)

Laravel Authentication



Information available on https://itnext.io/laravel-login-users-with-their-fingerprints-5161ecdca63b

Authentication: Introduction

1) Many web applications provide a way for their users to authenticate with the application and "login". Implementing this feature in web applications can be a complex and potentially risky endeavor.
2) For this reason, Laravel strives to give the tools needed to implement authentication quickly, securely, and easily.
3) Application's authentication configuration file is located at config/auth.php. This file contains several well documented options for tweaking the behavior of Laravel's authentication services.

Authentication: Guards

1) At the core, Laravel's authentication facilities are made up of "guards" and "providers".
2) Guards define how users are authenticated for each request.
3) For example, Laravel ships with a `session` guard which maintains state using session storage and cookies.

Authentication: Providers

1) Providers define how users are retrieved from persistent storage.
2) Laravel ships with support for retrieving users using Eloquent and the database query builder.
3) However, one is free to define additional providers as needed for the web application.
4) Guards and providers should not be confused with "roles" and "permissions". More about authorizing user actions via permissions, will be discussed in authorization.

Authentication: Database Considerations

1) By default, Laravel includes an `App\Models\User` Eloquent model in `app\Models` directory.
2) This model may be used with the default Eloquent authentication driver.
3) If the web application is not using Eloquent, use the `database` authentication provider which uses the Laravel query builder

Authentication: Database Considerations

1) When building the database schema for the `App\Models\User` model, make sure the password column is at least 60 characters in length.
2) Verify that `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters.
3) This column will be used to store a token for users that select the "remember me" option for logins.

Authentication: Laravel's Built-in Browser Authentication Services

1) Laravel includes built-in authentication and session services which are typically accessed via the `Auth` and `Session` facades.
2) These features provide cookie based authentication for requests that are initiated from web browsers. They provide methods that allow to verify a user's credentials and authenticate the user.
3) In addition, these services will automatically store the proper authentication data in the user's session and issue the user's session cookie.

Authentication: Laravel Authentication Packages

1) Laravel Breeze
   - Tailwind CSS
2) Laravel Jetstream
   - Tailwind CSS
   - Livewire
   - Inertia js
3) Laravel Fortify
4) laravel/ui package
   - vue.js
   - React.js
   - bootstrap

Authentication: Quickstart

1) **Laravel Breeze** is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is made up of simple Blade templates styled with Tailwind CSS.
2) **Laravel Jetstream** is a more robust application starter kit that includes support for scaffolding your application with Livewire or Inertia.js and Vue. In addition, **Jetstream** features optional support for two-factor authentication, teams, profile management, browser session management, API support via **Laravel Sanctum**, account deletion, and more.

Authentication: Controllers

1) Laravel ships with several pre-built authentication controllers, which are located in the `App\Http\Controllers\Auth` namespace.
   - The `RegisterController` handles new user registration.
   - The `LoginController` handles authentication.
   - The `ForgotPasswordController` handles e-mailing links for resetting passwords.
   - The `ResetPasswordController` contains the logic to reset passwords.
2) Each of these controllers uses a trait to include their necessary methods. For many applications, there is no need to modify these controllers at all.

Authentication: Routes

1) Laravel provides a quick way to scaffold all the routes and views needed for authentication one simple command. `php artisan make:auth`
2) The command should be used on fresh applications and will install a `layout view`, `registration` and `login` views, as well as `routes` for all authentication end-points.
3) A `HomeController` will also be generated to handle post-login requests to your application's dashboard.

Authentication: Views

1) The `make:auth` Artisan command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.
2) The `make:auth` command will also create a `resources/views/layouts` directory containing a base layout for your application.
3) All of these views use the Bootstrap CSS framework, but they are free to be customized.

Authentication: Path Configuration

1) When a user is successfully authenticated, by default, they will be redirected to the `/home` URI.
2) One can customize the post-authentication redirect location by defining `redirectTo` property in `LoginController`, `RegisterController`, and `ResetPasswordController`

```
protected $redirectTo = '/';

protected function redirectTo()
{
    return '/path';
}
```

Authentication: Username Configuration

1) By default, Laravel uses the `email` field for authentication
2) Define a `username` and method in `LoginController`:

```
public function username()
{
    return 'username';
}
```

Authentication: Guard Configuration

1) One may also customize the guard that is used to authenticate and register users.
2) To get started, define a `guard` method on your `LoginController`, `RegisterController`, and `ResetPasswordController`. The method should return a `guard` instance

```
use Illuminate\Support\Facades\Auth;
protected function guard()
{
    return Auth::guard('guard-name');
}
```

Authentication: Validation Configuration

1) To modify the form fields that are required when a new user registers with the web application, or to customize how new users are stored into database, modify the `RegisterController` class. The class is reponsible for validating and creating new users of the applicaiton.
2) The `validator` method of `RegisterController` contains validation rules for new users of the applicaiton. The method is free to be modified for specific purposes.
3) The `create` method of the `RegisterController` is responsible for creating new `App\User` records in database using the Eloquent ORM.

Authentication: Retrieving The Authenticated User

1) Access the authenticated user via the `Auth` facade:

```
use Illuminate\Support\Facades\Auth;

// Get the currently authenticated user...
$user = Auth::user();

// Get the currently authenticated user's ID...
$id = Auth::id();
```

Authentication: Retrieving The Authenticated User

1) Alternatively, once a user is authenticated, access the authenticated user via an `Illuminate\Http\Request` instance

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class ProfileController extends Controller {
  public function update(Request $request) {
  //returns an instance of the authenticated
     user...
     $request->user()
  }
}
```

Authentication: Determining If The Current User Is Authenticated

1) To determine if the user making the incoming HTTP request is authenticated, use the `check` method on the `Auth` facade. This method will return `true` if the user is authenticated.

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

Authentication: Protecting Routes

1) Route middleware can be used to only allow authenticated users to access a given route.
2) Laravel ships with an `auth` middleware, which references the `Illuminate\Auth\Middleware\Authenticate` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition

```
Route::get('profile', function ()
{ // Only authenticated users may enter...
})->middleware('auth');
```

Authentication: Specifying A Guard

1) When attaching the `auth` middleware to a route, one may specify which "guard" should be used to authenticate the user.
2) The `guard` specified should correspond to one of the keys in the `guards` array of `auth.php` configuration file

```
public function __construct()
{
    $this->middleware('auth:api');
}
```

Authentication: Login Throttling

1) Using Laravel Breeze or Laravel Jetstream starter kits, rate limiting will automatically be applied to login attempts.
2) By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts.
3) The throttling is unique to the user's username / email address and their IP address

```
Illuminate\Foundation\Auth\ThrottlesLogins
```

Authentication: Manually Authenticating Users

1) Access Laravel's authentication services via the `Auth` facade; make sure to import the `Auth` facade at the top of the class.
2) The `attempt` method is normally used to handle authentication attempt's from application's "login" form. The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.
3) The `intended` method provided by Laravel's `redirector` will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware.

Authentication: Manually Authenticating Users

```
namespace App\Http\Controllers;
use Illuminate\Support\Facades\Auth;
class LoginController extends Controller {
public function authenticate()
{
  if (Auth::attempt(['email' => $email,
    'password' => $password]))
  { // Authentication passed...
   return redirect()->intended('dashboard');
  }
 }
}
```

Authentication: Remembering Users

1) Many web applications provide a "remember me" checkbox on their login form and pass a boolean value as second argument in `attempt` method.
2) When this value is true, Laravel will keep the user authenticated indefinitely or until they manually logout.
3) Database table must include the string remember_token column, which will be used to store the "remember me" token.

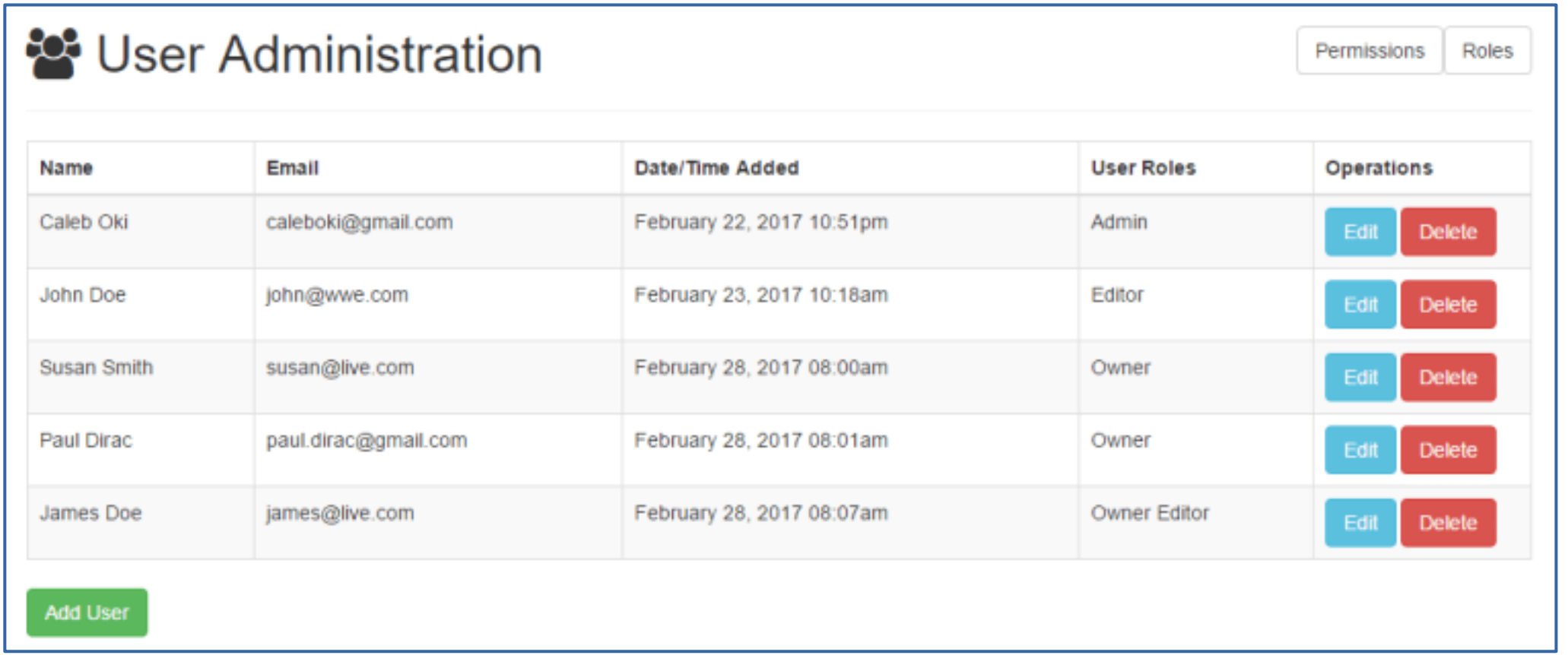
```
if (Auth::attempt(['email' => $email,
'password' => $password], $remember)) {
}
```

Authentication: Logging Out

1) To log users out of web application, use the `logout` method on the `Auth` facade.
2) This will clear the authentication information in the user's session

```
Auth::logout();
```

Authorization



Information available in: https://scotch.io/tutorials/user-authorization-in-laravel-54-with-spatie-laravel-permission

Authorization: Introduction

1) Laravel provides a simple way to authorize user actions against a given resource. Like authentication.
2) There are two primary ways of authorizing actions: gates and policies.
   - Gates provide a simple, `Closure` based approach to authorization.
   - Policies group their logic around a particular model or resource.

Authorization: Introduction

1) There is no need to choose between exclusively using gates or exclusively using policies when building an application.
2) Most applications will most likely contain a mixture of gates and policies.
3) Gates are most applicable to actions which are not related to any model or resource, such as viewing an administrator dashboard.
4) Policies should be used to authorize an action for a particular model or resource.

Authorization: Gates (Writing Gates)

1) Gates are simply closures that determine if a user is authorized to perform a given action.
2) Typically, gates are defined within the `boot` method of the `App\Providers\AuthServiceProvider` class using the `Gate` facade.
3) Gates always receive a user instance as their first argument and may optionally receive additional arguments such as a relevant Eloquent model

Authorization: Gates (Writing Gates)

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;
public function boot()
{
    $this->registerPolicies();

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

Authorization: Gates (Writing Gates)

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

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post',
      [PostPolicy::class, 'update']);
}
```

Authorization: Gates (Authorizing Actions)

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

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

Authorization: Gates (Authorizing Actions)

1) To determine if a user other than the currently authenticated user is authorized to perform an action, use the `forUser` method on the `Gate` facade

```
if (Gate::forUser($user)->allows('update-
post', $post)) {
    // The user can update the post...
}

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

Authorization: Policies (Generating Policies)

1) Policies are classes that organize authorization logic around a particular model or resource.
   - E.g., if web application is a blog, there might be an `App\Models\Post` model and a corresponding `App\Policies\PostPolicy` to authorize user actions such as creating or updating posts.
2) Generate a policy using the `make:policy` Artisan command. The generated policy will be placed in the `app/Policies` directory.

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

Authorization: Policies (Generating Policies)

1) The `make:policy` command will generate an empty policy class.
2) If one would like to generate a class with basic policy methods related to viewing, creating, updating, and deleting (CRUD) the resource, provide a `--model` option when executing the command

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

Authorization: Policies (Registering Policies)

1) Once the policy class has been created, it needs to be registered
2) The `App\Providers\AuthServiceProvider` included with fresh Laravel applications contains a policies property which maps the Eloquent models to their corresponding policies.
3) Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given Eloquent model.

Authorization: Policies (Registering Policies)

```php
<?php
namespace App\Providers;
use App\Models\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\
AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;
class AuthServiceProvider extends ServiceProvider {
    protected $policies = [
        Post::class => PostPolicy::class,
    ];
    public function boot()
    {
        $this->registerPolicies();
    }
}
```

Authorization: Policies (Writing Policies)

1) Once the policy class has been registered, one may add methods for each action it authorizes.
   - For example, let's define an `update` method on `PostPolicy` which determines if a given `App\Models\User` can update a given `App\Models\Post` instance.
   - 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`.

Authorization: Policies (Writing Policies)

```php
<?php
namespace App\Policies;

use App\Models\Post;
use App\Models\User;

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

Authorization: Policies (Methods Without Models)

1) Some policy methods only receive an instance of the currently authenticated user. This situation is most common when authorizing create actions.
   - E.g.: if one is creating a blog, one may wish to determine if a user is authorized to create any posts at all. In these situations, the policy method should only expect to receive a user instance:

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

Authorization: Policies (Policy Filters)

1) For certain users, one may wish to authorize all actions within a given policy.
2) To accomplish this, define a `before` method on the policy.
   - The `before` method will be executed before any other methods on the policy, to authorize the action before the intended policy method is actually called.
   - This feature is most commonly used for authorizing application administrators to perform any action

Authorization: Policies (Policy Filters)

1) If would like to deny all authorization checks for a particular type of user then, return `false` from the `before` method. If `null` is returned, the authorization check will fall through to the policy method.

```
public function before(User $user, $ability)
{
    if ($user->isAdministrator()) {
        return true;
    }
}
```

Authorization: Policies (Authorizing Actions Using Policies: Via User Model)

1) The `App\Models\User` model that is included with Laravel application includes two helpful methods for authorizing actions: `can` and `cannot`.

- E.g.: Let's determine if a user is authorized to update a given `App\Models\Post model`. Typically, this will be done within a controller method

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

Authorization: Policies (Authorizing Actions That Don't Require Models)

1) Remember, some actions may correspond to policy methods like `create` that do not require a model instance. In these situations, pass a class name to the `can` method. The class name will be used to determine which policy to use when authorizing the action:

```
public function store(Request $request){
 if ($request->user()->cannot('create', Post::class))
   {
       abort(403);
   }
 // Create the post...
}
```

Authorization: Policies (Authorizing Actions Using Policies: Via Controller Helpers)

1) In addition to helpful methods provided to the `App\Models\User` model, Laravel provides a helpful `authorize` method to any of the controllers which extend the `App\Http\Controllers\Controller` base class.
2) If the action is not authorized, the `authorize` method will throw an `Illuminate\Auth\Access\AuthorizationException` exception: HTTP response with a 403 status code

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

Authorization: Policies (Authorizing Actions Using Policies: Via Controller Helpers)

1) Some policy methods like `create` do not require a model instance. In these situations, pass a class name to the `authorize` method.
2) The class name will be used to determine which policy to use when authorizing the action:

```
public function create(Request $request)
{
  $this->authorize('create', Post::class);
  // The current user can create blog posts...
}
```

Authorization: Policies (Authorizing Actions Using Policies: Via Middleware)

1) Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers.
2) By default, the `Illuminate\Auth\Middleware\Authorize` middleware assigned the `can` key in `App\Http\Kernel` class.

```
use App\Models\Post;

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

Authorization: Policies (Authorizing Actions Using Policies: Via Middleware)

1) Again, some policy methods like `create` do not require a model instance. In these situations, pass a class name to the middleware.
2) The class name will be used to determine which policy to use when authorizing the action.

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

Authorization: Policies (Authorizing Actions Using Policies: Via Blade Templates)

1) There are instances to display a portion of the page only if the user is authorized to perform a given action.
   - E.g.: to show an update form for a blog post only if the user can actually update the post. In this situation, 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
```

Authorization: Policies (Authorizing Actions Using Policies: Via Blade Templates)

1) These directives are convenient shortcuts for writing `@if` and `@unless` statements.
2) The previous `@can` statements can be respectively translate to the following statements

```
@if (Auth::user()->can('update', $post))
<!-- The Current User Can Update The Post -->
@endif
```

Authorization: Policies (Authorizing Actions Using Policies: Blade Templates)

1) Like most of the other authorization methods, one 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
```

Authentication and Authorization: Role-based Access Control

1) Laravel does NOT come with Role-based Access Control (RBAC) feature by default.
2) One may develop own implementation of RBAC, which involves defining various tables and their respective model classes as well as controlllers for the authorization logics.
3) Otherwise, packages for implementing RBAC in Laravel are available:
   - Bouncer
   - Spatie laravel-permission

END OF LECTURE 07