Authentication

Introduction

Want to get started fast? Just run `php artisan make:auth` and `php artisan migrate` in a fresh Laravel application. Then, navigate your browser to `http://your-app.dev/register` or any other URL that is assigned to your application. These two commands will take care of scaffolding your entire authentication system!

Laravel makes implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at `config/auth.php`, which contains several well documented options for tweaking the behavior of the authentication services.

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

Providers define how users are retrieved from your persistent storage. Laravel ships with support for retrieving users using Eloquent and the database query builder. However, you are free to define additional providers as needed for your application.

Don't worry if this all sounds confusing now! Many applications will never need to modify the default authentication configuration.

Database Considerations

By default, Laravel includes an `App\User`Eloquent model in your `app` directory. This model may be used with the default Eloquent authentication driver. If your application is not using Eloquent, you may use the `database` authentication driver which uses the Laravel query builder.

When building the database schema for the `App\User` model, make sure the password column is at least 60 characters in length. Maintaining the default string column length of 255 characters would be a good choice.

Also, you should verify that your `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for users that select the "remember me" option when logging into your application.

Authentication Quickstart

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, and the `ResetPasswordController` contains the logic to reset passwords. Each of these controllers uses a trait to include their necessary methods. For many applications, you will not need to modify these controllers at all.

Routing

Laravel provides a quick way to scaffold all of the routes and views you need for authentication using one simple command:

```
php artisan make:auth
```

This 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. A `HomeController` will also be generated to handle post-login requests to your application's dashboard.

Views

As mentioned in the previous section, the `php artisan make:auth` command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.

The `make:auth` command will also create a `resources/views/layouts` directory containing a base layout for your application. All of these views use the Bootstrap CSS framework, but you are free to customize them however you wish.

Authenticating

Now that you have routes and views setup for the included authentication controllers, you are ready to register and authenticate new users for your application! You may simply access your application in a browser since the authentication controllers already contain the logic (via their traits) to authenticate existing users and store new users in the database.

Path Customization

When a user is successfully authenticated, they will be redirected to the `/home` URI. You can customize the post-authentication redirect location by defining a `redirectTo` property on the `LoginController`, `RegisterController`, and `ResetPasswordController`:

```
protected$redirectTo='/';
```

When a user is not successfully authenticated, they will be automatically redirected back to the login form.

Username Customization

By default, Laravel uses the `email` field for authentication. If you would like to customize this, you may define a `username` method on your `LoginController`:

```
publicfunctionusername()
{
return'username';
}
```

Guard Customization

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

```
useIlluminate\Support\Facades\Auth;

protectedfunctionguard()
{
returnAuth::guard('guard-name');
}
```

Validation / Storage Customization

To modify the form fields that are required when a new user registers with your application, or to customize how new users are stored into your database, you may modify the `RegisterController` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `RegisterController` contains the validation rules for new users of the application. You are free to modify this method as you wish.

The `create` method of the `RegisterController` is responsible for creating new `App\`User records in your database using the Eloquent ORM. You are free to modify this method according to the needs of your database.

Retrieving The Authenticated User

You may access the authenticated user via the `Auth` facade:

```
useIlluminate\Support\Facades\Auth;

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

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

Alternatively, once a user is authenticated, you may access the authenticated user via an `Illuminate\`Http\Request instance. Remember, type-hinted classes will automatically be injected into your controller methods:

```
<?php

namespaceApp\Http\Controllers;

useIlluminate\Http\Request;

classProfileControllerextendsController
{
/**
     * Update the user's profile.
     *
     * @param  Request  $request
     * @return Response
     */
publicfunctionupdate(Request $request)
{
 // $request->user() returns an instance of the authenticated user...
}
}
```

Determining If The Current User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method on the `Auth` facade, which will return true if the user is authenticated:

```
useIlluminate\Support\Facades\Auth;

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

Even though it is possible to determine if a user is authenticated using the `check` method, you will typically use a middleware to verify that the user is authenticated before allowing the user access to certain routes / controllers. To learn more about this, check out the documentation on protecting routes.

Protecting Routes

Route middleware can be used to only allow authenticated users to access a given route. Laravel ships with an `auth` middleware, which is defined at `Illuminate\`Auth\Middleware\Authenticate. Since this middleware is already registered in your 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');
```

Of course, if you are using controllers, you may call the `middleware` method from the controller's constructor instead of attaching it in the route definition directly:

```
publicfunction__construct()
{
$this->middleware('auth');
}
```

Specifying A Guard

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

```
publicfunction__construct()
{
$this->middleware('auth:api');
}
```

Login Throttling

If you are using Laravel's built-in `LoginController` class, the `Illuminate\`Foundation\Auth\ThrottlesLogins trait will already be included in your controller. By default, the user will not be able to login for one minute if they fail to

provide the correct credentials after several attempts. The throttling is unique to the user's username / e-mail address and their IP address.

Manually Authenticating Users

Of course, you are not required to use the authentication controllers included with Laravel. If you choose to remove these controllers, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the `Auth`facade, so we'll need to make sure to import the `Auth` facade at the top of the class. Next, let's check out the `attempt` method:

```
<?php

namespaceApp\Http\Controllers;

useIlluminate\Support\Facades\Auth;

classLoginControllerextendsController
{
/**
     * Handle an authentication attempt.
     *
     * @return Response
     */
publicfunctionauthenticate()
{
if(Auth::attempt(['email'=>$email,'password'=>$password])){
 // Authentication passed...
returnredirect()->intended('dashboard');
}
}
}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the hashed `password` value passed to the method via the array. If the two hashed passwords match an authenticated session will be started for the user.

The `attempt` method will return true if authentication was successful. Otherwise, false will be returned.

The `intended` method on the redirector will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware. A fallback URI may be given to this method in case the intended destination is not available.

Specifying Additional Conditions

If you wish, you also may add extra conditions to the authentication query in addition to the user's e-mail and password. For example, we may verify that user is marked as "active":

```
if(Auth::attempt(['email'=>$email,'password'=>$password,'active'=>1])){
 // The user is active, not suspended, and exists.
}
```

In these examples, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database.

Accessing Specific Guard Instances

You may specify which guard instance you would like to utilize using the `guard` method on the `Auth` facade. This allows you to manage authentication for separate parts of your application using entirely separate authenticatable models or user tables.

The guard name passed to the `guard` method should correspond to one of the guards configured in your `auth.php` configuration file:

```
if(Auth::guard('admin')->attempt($credentials)){
 //
}
```

Logging Out

To log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

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

Remembering Users

If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method, which will keep the user authenticated indefinitely, or until they manually logout. Of course, your `users` 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)){
 // The user is being remembered...
}
```

If you are using the built-in `LoginController` that is shipped with Laravel, the proper logic to "remember" users is already implemented by the traits used by the controller.

If you are "remembering" users, you may use the `viaRemember` method to determine if the user was authenticated using the "remember me" cookie:

```
if(Auth::viaRemember()){
 //
```

```
}
```

Other Authentication Methods

Authenticate A User Instance

If you need to log an existing user instance into your application, you may call the `login` method with the user instance. The given object must be an implementation of the `Illuminate\`Contracts\Auth\Authenticatable[contract](). Of course, the `App\`User model included with Laravel already implements this interface:

```
Auth::login($user);

// Login and "remember" the given user...
Auth::login($user,true);
```

Of course, you may specify the guard instance you would like to use:

```
Auth::guard('admin')->login($user);
```

Authenticate A User By ID

To log a user into the application by their ID, you may use the `loginUsingId` method. This method simply accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);

// Login and "remember" the given user...
Auth::loginUsingId(1,true);
```

Authenticate A User Once

You may use the `once` method to log a user into the application for a single request. No sessions or cookies will be utilized, which means this method may be helpful when building a stateless API:

```
if(Auth::once($credentials)){
 //
}
```

HTTP Basic Authentication

HTTP Basic Authentication provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the `auth.basic`[middleware]() to your route. The `auth.basic` middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('profile',function(){
```

```
 // Only authenticated users may enter...
})->middleware('auth.basic');
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the `auth.basic` middleware will use the `email` column on the user record as the "username".

A Note On FastCGI

If you are using PHP FastCGI, HTTP Basic authentication may not work correctly out of the box. The following lines should be added to your `.htaccess` file:

```
RewriteCond %{HTTP:Authorization}^(.+)$
RewriteRule .*-[E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session, which is particularly useful for API authentication. To do so, define a middleware that calls the `onceBasic` method. If no response is returned by the `onceBasic` method, the request may be passed further into the application:

```
<?php

namespaceIlluminate\Auth\Middleware;

useIlluminate\Support\Facades\Auth;

classAuthenticateOnceWithBasicAuth
{
/**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
publicfunctionhandle($request,$next)
{
returnAuth::onceBasic()?:$next($request);
}

}
```

Next, register the route middleware and attach it to a route:

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

Adding Custom Guards

You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place this call to `provider` within a service provider. Since Laravel already ships with an `AuthServiceProvider`, we can place the code in that provider:

```
<?php

namespaceApp\Providers;

useApp\Services\Auth\JwtGuard;
useIlluminate\Support\Facades\Auth;
useIlluminate\Foundation\Support\Providers\AuthServiceProvideras
ServiceProvider;

classAuthServiceProviderextendsServiceProvider
{
/**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
publicfunctionboot()
{
$this->registerPolicies();

Auth::extend('jwt',function($app,$name,array$config){
 // Return an instance of Illuminate\Contracts\Auth\Guard...

returnnewJwtGuard(Auth::createUserProvider($config['provider']));
});
}
}
```

As you can see in the example above, the callback passed to the `extend` method should return an implementation of `Illuminate\`Contracts\Auth\Guard. This interface contains a few methods you will need to implement to define a custom guard. Once your custom guard has been defined, you may use the guard in the `guards` configuration of your `auth.php` configuration file:

```
'guards'=>[
'api'=>[
'driver'=>'jwt',
'provider'=>'users',
],
],
```

Adding Custom User Providers

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication user provider. We will use the `provider` method on the `Auth` facade to define a custom user provider:

```
<?php
```

```
namespaceApp\Providers;

useIlluminate\Support\Facades\Auth;
useApp\Extensions\RiakUserProvider;
useIlluminate\Support\ServiceProvider;

classAuthServiceProviderextendsServiceProvider
{
/**
     * Register any application authentication / authorization services.
*
     * @return void
     */
publicfunctionboot()
{
$this->registerPolicies();

Auth::provider('riak',function($app,array$config){
 // Return an instance of Illuminate\Contracts\Auth\UserProvider...

returnnewRiakUserProvider($app->make('riak.connection'));
});
}
}
```

After you have registered the provider using the `provider` method, you may switch to the new user provider in your `auth.php` configuration file. First, define a `provider` that uses your new driver:

```
'providers'=>[
'users'=>[
'driver'=>'riak',
],
],
```

Finally, you may use this provider in your `guards` configuration:

```
'guards'=>[
'web'=>[
'driver'=>'session',
'provider'=>'users',
],
],
```

The User Provider Contract

The `Illuminate\`Contracts\Auth\UserProvider implementations are only responsible for fetching a `Illuminate\`Contracts\Auth\Authenticatable implementation out of a persistent storage system, such as MySQL, Riak, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent it.

Let's take a look at the `Illuminate\Contracts\Auth\UserProvider` contract:

```
<?php

namespaceIlluminate\Contracts\Auth;

interfaceUserProvider{

publicfunctionretrieveById($identifier);
publicfunctionretrieveByToken($identifier,$token);
publicfunctionupdateRememberToken(Authenticatable $user,$token);
publicfunctionretrieveByCredentials(array$credentials);
publicfunctionvalidateCredentials(Authenticatable
$user,array$credentials);

}
```

The `retrieveById` function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The `Authenticatable` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByToken` function retrieves a user by their unique $identifier and "remember me" $token, stored in a field `remember_token`. As with the previous method, the `Authenticatable` implementation should be returned.

The `updateRememberToken` method updates the $user field `remember_token` with the new $token. The new token can be either a fresh token, assigned on a successful "remember me" login attempt, or null when the user is logging out.

The `retrieveByCredentials` method receives the array of credentials passed to the Auth::`attempt` method when attempting to sign into an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition on $credentials['username']. The method should then return an implementation of `Authenticatable`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given $user with the $credentials to authenticate the user. For example, this method should probably use Hash::`check` to compare the value of $user->getAuthPassword() to the value of $credentials['password']. This method should return true or false indicating on whether the password is valid.

The Authenticatable Contract

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable` contract. Remember, the provider should return implementations of this interface from the `retrieveById` and `retrieveByCredentials` methods:

```
<?php
```

```
namespaceIlluminate\Contracts\Auth;

interfaceAuthenticatable{

publicfunctiongetAuthIdentifierName();
publicfunctiongetAuthIdentifier();
publicfunctiongetAuthPassword();
publicfunctiongetRememberToken();
publicfunctionsetRememberToken($value);
publicfunctiongetRememberTokenName();

}
```

This interface is simple. The `getAuthIdentifierName` method should return the name of the "primary key" field of the user and the `getAuthIdentifier` method should return the "primary key" of the user. In a MySQL back-end, again, this would be the auto-incrementing primary key. The `getAuthPassword` should return the user's hashed password. This interface allows the authentication system to work with any User class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a `User` class in the `app` directory which implements this interface, so you may consult this class for an implementation example.

Events

Laravel raises a variety of events during the authentication process. You may attach listeners to these events in your `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected$listen=[
'Illuminate\Auth\Events\Registered'=>[
'App\Listeners\LogRegisteredUser',
],

'Illuminate\Auth\Events\Attempting'=>[
'App\Listeners\LogAuthenticationAttempt',
],

'Illuminate\Auth\Events\Authenticated'=>[
'App\Listeners\LogAuthenticated',
],

'Illuminate\Auth\Events\Login'=>[
'App\Listeners\LogSuccessfulLogin',
],

'Illuminate\Auth\Events\Logout'=>[
'App\Listeners\LogSuccessfulLogout',
],
```

```
'Illuminate\Auth\Events\Lockout'=>[
'App\Listeners\LogLockout',
],
];
```