Laravel Passport

Introduction

Laravel already makes it easy to perform authentication via traditional login forms, but what about APIs? APIs typically use tokens to authenticate users and do not maintain session state between requests. Laravel makes API authentication a breeze using Laravel Passport, which provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the League OAuth2 server that is maintained by Andy Millington and Simon Hamp.

This documentation assumes you are already familiar with OAuth2. If you do not know anything about OAuth2, consider familiarizing yourself with the general terminology and features of OAuth2 before continuing.

Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

The Passport service provider registers its own database migration directory with the framework, so you should migrate your database after installing the package. The Passport migrations will create the tables your application needs to store clients and access tokens:

```
php artisan migrate
```

Next, you should run the `passport:install` command. This command will create the encryption keys needed to generate secure access tokens. In addition, the command will create "personal access" and "password grant" clients which will be used to generate access tokens:

```
php artisan passport:install
```

After running this command, add the `Laravel\Passport\HasApiTokens` trait to your `App\User` model. This trait will provide a few helper methods to your model which allow you to inspect the authenticated user's token and scopes:

```
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

Next, you should call the `Passport::routes` method within the `boot` method of your `AuthServiceProvider`. This method will register the routes necessary to issue access tokens and revoke access tokens, clients, and personal access tokens:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider
use Illuminate\Support\Facades\Gate;
use Laravel\Passport\Passport;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

Finally, in your `config/auth.php` configuration file, you should set the `driver` option of the `api` authentication guard to `passport`. This will instruct your application to use Passport's `TokenGuard` when authenticating incoming API requests:

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

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Migration Customization

If you are not going to use Passport's default migrations, you should call the `Passport::ignoreMigrations` method in the `register` method of your `AppServiceProvider`. You may export the default migrations using `php artisan vendor:publish --tag=passport-migrations`.

By default, Passport uses an integer column to store the `user_id`. If your application uses a different column type to identify users (for example: UUIDs), you should modify the default Passport migrations after publishing them.

Frontend Quickstart

In order to use the Passport Vue components, you must be using the Vue JavaScript framework. These components also use the Bootstrap CSS framework. However, even if you are not using these tools, the components serve as a valuable reference for your own frontend implementation.

Passport ships with a JSON API that you may use to allow your users to create clients and personal access tokens. However, it can be time consuming to code a frontend to interact with these APIs. So, Passport also includes pre-built Vue components you may use as an example implementation or starting point for your own implementation.

To publish the Passport Vue components, use the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag=passport-components
```

The published components will be placed in your `resources/js/components` directory. Once the components have been published, you should register them in your `resources/js/app.js` file:

```
Vue.component(
    'passport-clients',
    require('./components/passport/Clients.vue').default
);

Vue.component(
    'passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue').default
);

Vue.component(
    'passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue').default
);
```

Prior to Laravel v5.7.19, appending `.default` when registering components results in a console error. An explanation for this change can be found in the Laravel Mix v4.0.0 release notes.

After registering the components, make sure to run `npm run dev` to recompile your assets. Once you have recompiled your assets, you may drop the components into one of your application's templates to get started creating clients and personal access tokens:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Deploying Passport

When deploying Passport to your production servers for the first time, you will likely need to run the `passport:keys` command. This command generates the encryption keys Passport needs in order to generate access token. The generated keys are not typically kept in source control:

```
php artisan passport:keys
```

If necessary, you may define the path where Passport's keys should be loaded from. You may use the `Passport::loadKeysFrom` method to accomplish this:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::loadKeysFrom('/secret-keys/oauth');
```

```
}
```

Additionally, you may publish Passport's configuration file using `php artisan vendor:publish --tag=passport-config`, which will then provide the option to load the encryption keys from your environment variables:

```
PASSPORT_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
<private key here>
-----END RSA PRIVATE KEY-----"

PASSPORT_PUBLIC_KEY="-----BEGIN PUBLIC KEY-----
<public key here>
-----END PUBLIC KEY-----"
```

Configuration

Token Lifetimes

By default, Passport issues long-lived access tokens that expire after one year. If you would like to configure a longer / shorter token lifetime, you may use the `tokensExpireIn`, `refreshTokensExpireIn`, and `personalAccessTokensExpireIn` methods. These methods should be called from the `boot` method of your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(now()->addDays(15));

    Passport::refreshTokensExpireIn(now()->addDays(30));

    Passport::personalAccessTokensExpireIn(now()->addMonths(6));
}
```

Overriding Default Models

You are free to extend the models used internally by Passport. Then, you may instruct Passport to use your custom models via the `Passport` class:

```
use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessClient;
use App\Models\Passport\Token;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::useTokenModel(Token::class);
    Passport::useClientModel(Client::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::usePersonalAccessClientModel(PersonalAccessClient::class);
}
```

Issuing Access Tokens

Using OAuth2 with authorization codes is how most developers are familiar with OAuth2. When using authorization codes, a client application will redirect a user to your server where they will either approve or deny the request to issue an access token to the client.

Managing Clients

First, developers building applications that need to interact with your application's API will need to register their application with yours by creating a "client". Typically, this consists of providing the name of their application and a URL that your application can redirect to after users approve their request for authorization.

The `passport:client` Command

The simplest way to create a client is using the `passport:client` Artisan command. This command may be used to create your own clients for testing your OAuth2 functionality. When you run the `client` command, Passport will prompt you for more information about your client and will provide you with a client ID and secret:

```
php artisan passport:client
```

Redirect URLs

If you would like to whitelist multiple redirect URLs for your client, you may specify them using a comma-delimited list when prompted for the URL by the `passport:client` command:

```
http://example.com/callback,http://examplefoo.com/callback
```

Any URLs which contains commas must be encoded.

JSON API

Since your users will not be able to utilize the `client` command, Passport provides a JSON API that you may use to create clients. This saves you the trouble of having to manually code controllers for creating, updating, and deleting clients.

However, you will need to pair Passport's JSON API with your own frontend to provide a dashboard for your users to manage their clients. Below, we'll review all of the API endpoints for managing clients. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

If you don't want to implement the entire client management frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`GET /oauth/clients`

This route returns all of the clients for the authenticated user. This is primarily useful for listing all of the user's clients so that they may edit or delete them:

```
axios.get('/oauth/clients')
    .then(response => {
        console.log(response.data);
    });
```

`POST /oauth/clients`

This route is used to create new clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization.

When a client is created, it will be issued a client ID and client secret. These values will be used when requesting access tokens from your application. The client creation route will return the new client instance:

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`PUT /oauth/clients/{client-id}`

This route is used to update clients. It requires two pieces of data: the client's `name` and a `redirect` URL. The `redirect` URL is where the user will be redirected after approving or denying a request for authorization. The route will return the updated client instance:

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
```

`DELETE /oauth/clients/{client-id}`

This route is used to delete clients:

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        //
    });
```

Requesting Tokens

Redirecting For Authorization

Once a client has been created, developers may use their client ID and secret to request an authorization code and access token from your application. First, the consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Approving The Request

When receiving authorization requests, Passport will automatically display a template to the user allowing them to approve or deny the authorization request. If they approve the request, they will be redirected back to the `redirect_uri` that was specified by the consuming application. The `redirect_uri` must match the `redirect` URL that was specified when the client was created.

If you would like to customize the authorization approval screen, you may publish Passport's views using the `vendor:publish` Artisan command. The published views will be placed in `resources/views/vendor/passport`:

```
php artisan vendor:publish --tag=passport-views
```

Sometimes you may wish to skip the authorization prompt, such as when authorizing a first-party client. You may accomplish this by defining a `skipsAuthorization` method on the client model. If `skipsAuthorization` returns `true` the client will be approved and the user will be redirected back to the `redirect_uri` immediately:

```
<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;
```

```
class Client extends BaseClient
{
    /**
     * Determine if the client should skip the authorization prompt.
     *
     * @return bool
     */
    public function skipsAuthorization()
    {
        return $this->firstParty();
    }
}
```

Converting Authorization Codes To Access Tokens

If the user approves the authorization request, they will be redirected back to the consuming application. The consumer should first verify the `state` parameter against the value that was stored prior to the redirect. If the state parameter matches the consumer should issue a `POST` request to your application to request an access token. The request should include the authorization code that was issued by your application when the user approved the authorization request. In this example, we'll use the Guzzle HTTP library to make the `POST` request:

```
Route::get('/callback', function (Request $request) {
    $state = $request->session()->pull('state');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);

    return json_decode((string) $response->getBody(), true);
});
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

Like the `/oauth/authorize` route, the `/oauth/token` route is defined for you by the `Passport::routes` method. There is no need to manually define this route. By default, this route is throttled using the settings of the `ThrottleRequests` middleware.

Refreshing Tokens

If your application issues short-lived access tokens, users will need to refresh their access tokens via the refresh token that was provided to them when the access token was issued. In this example, we'll use the Guzzle HTTP library to refresh the token:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

This `/oauth/token` route will return a JSON response containing `access_token`, `refresh_token`, and `expires_in` attributes. The `expires_in` attribute contains the number of seconds until the access token expires.

Password Grant Tokens

The OAuth2 password grant allows your other first-party clients, such as a mobile application, to obtain an access token using an e-mail address / username and password. This allows you to issue access tokens securely to your first-party clients without requiring your users to go through the entire OAuth2 authorization code redirect flow.

Creating A Password Grant Client

Before your application can issue tokens via the password grant, you will need to create a password grant client. You may do this using the `passport:client` command with the `--password` option. If you have already run the `passport:install` command, you do not need to run this command:

```
php artisan passport:client --password
```

Requesting Tokens

Once you have created a password grant client, you may request an access token by issuing a `POST` request to the `/oauth/token` route with the user's email address and password. Remember, this route is already registered by the `Passport::routes` method so there is no need to define it manually. If the request is successful, you will receive an `access_token` and `refresh_token` in the JSON response from the server:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

Remember, access tokens are long-lived by default. However, you are free to configure your maximum access token lifetime if needed.

Requesting All Scopes

When using the password grant or client credentials grant, you may wish to authorize the token for all of the scopes supported by your application. You can do this by requesting the `*` scope. If you request the `*` scope, the `can` method on the token instance will always return `true`. This scope may only be assigned to a token that is issued using the `password` or `client_credentials` grant:

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
]);
```

Customizing The Username Field

When authenticating using the password grant, Passport will use the `email` attribute of your model as the "username". However, you may customize this behavior by defining a `findForPassport` method on your model:

```
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
```

```
     * Find the user instance for the given username.
     *
     * @param  string  $username
     * @return \App\User
     */
    public function findForPassport($username)
    {
        return $this->where('username', $username)->first();
    }
}
```

Customizing The Password Validation

When authenticating using the password grant, Passport will use the `password` attribute of your model to validate the given password. If your model does not have a `password` attribute or you wish to customize the password validation logic, you can define a `validateForPassportPasswordGrant` method on your model:

```
<?php

namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Hash;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * Validate the password of the user for the Passport password grant.
     *
     * @param  string  $password
     * @return bool
     */
    public function validateForPassportPasswordGrant($password)
    {
        return Hash::check($password, $this->password);
    }
}
```

Implicit Grant Tokens

The implicit grant is similar to the authorization code grant; however, the token is returned to the client without exchanging an authorization code. This grant is most commonly used for JavaScript or mobile applications where the client credentials can't be securely stored. To enable the grant, call the `enableImplicitGrant` method in your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Once a grant has been enabled, developers may use their client ID to request an access token from your application. The consuming application should make a redirect request to your application's `/oauth/authorize` route like so:

```
Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
        'state' => $state,
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Remember, the `/oauth/authorize` route is already defined by the `Passport::routes` method. You do not need to manually define this route.

Client Credentials Grant Tokens

The client credentials grant is suitable for machine-to-machine authentication. For example, you might use this grant in a scheduled job which is performing maintenance tasks over an API.

Before your application can issue tokens via the client credentials grant, you will need to create a client credentials grant client. You may do this using the `--client` option of the `passport:client` command:

```
php artisan passport:client --client
```

Next, to use this grant type, you need to add the `CheckClientCredentials` middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

Then, attach the middleware to a route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client');
```

To restrict access to the route to specific scopes you may provide a comma-delimited list of the required scopes when attaching the `client` middleware to the route:

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client:check-status,your-scope');
```

Retrieving Tokens

To retrieve a token using this grant type, make a request to the `oauth/token` endpoint:

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

return json_decode((string) $response->getBody(), true)['access_token'];
```

Personal Access Tokens

Sometimes, your users may want to issue access tokens to themselves without going through the typical authorization code redirect flow. Allowing users to issue tokens to themselves via your application's UI can be useful for allowing users to experiment with your API or may serve as a simpler approach to issuing access tokens in general.

Creating A Personal Access Client

Before your application can issue personal access tokens, you will need to create a personal access client. You may do this using the `passport:client` command with the `--personal` option. If you have already run the `passport:install` command, you do not need to run this command:

```
php artisan passport:client --personal
```

If you have already defined a personal access client, you may instruct Passport to use it using the `personalAccessClientId` method. Typically, this method should be called

from the `boot` method of your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::personalAccessClientId('client-id');
}
```

Managing Personal Access Tokens

Once you have created a personal access client, you may issue tokens for a given user using the `createToken` method on the `User` model instance. The `createToken` method accepts the name of the token as its first argument and an optional array of scopes as its second argument:

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport also includes a JSON API for managing personal access tokens. You may pair this with your own frontend to offer your users a dashboard for managing personal access tokens. Below, we'll review all of the API endpoints for managing personal access tokens. For convenience, we'll use Axios to demonstrate making HTTP requests to the endpoints.

The JSON API is guarded by the `web` and `auth` middleware; therefore, it may only be called from your own application. It is not able to be called from an external source.

If you don't want to implement the personal access token frontend yourself, you can use the frontend quickstart to have a fully functional frontend in a matter of minutes.

`GET /oauth/scopes`

This route returns all of the scopes defined for your application. You may use this route to list the scopes a user may assign to a personal access token:

```
axios.get('/oauth/scopes')
    .then(response => {
        console.log(response.data);
    });
```

`GET /oauth/personal-access-tokens`

This route returns all of the personal access tokens that the authenticated user has created. This is primarily useful for listing all of the user's tokens so that they may edit or delete them:

```
axios.get('/oauth/personal-access-tokens')
    .then(response => {
        console.log(response.data);
    });
```

`POST /oauth/personal-access-tokens`

This route creates new personal access tokens. It requires two pieces of data: the token's `name` and the `scopes` that should be assigned to the token:

```
const data = {
    name: 'Token Name',
    scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
```

```
    .then(response => {
        console.log(response.data.accessToken);
    })
    .catch (response => {
        // List errors on response...
    });
```

`DELETE /oauth/personal-access-tokens/{token-id}`

This route may be used to delete personal access tokens:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protecting Routes

Via Middleware

Passport includes an authentication guard that will validate access tokens on incoming requests. Once you have configured the `api` guard to use the `passport` driver, you only need to specify the `auth:api` middleware on any routes that require a valid access token:

```
Route::get('/user', function () {
    //
})->middleware('auth:api');
```

Passing The Access Token

When calling routes that are protected by Passport, your application's API consumers should specify their access token as a `Bearer` token in the `Authorization` header of their request. For example, when using the Guzzle HTTP library:

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);
```

Token Scopes

Scopes allow your API clients to request a specific set of permissions when requesting authorization to access an account. For example, if you are building an e-commerce application, not all API consumers will need the ability to place orders. Instead, you may allow the consumers to only request authorization to access order shipment statuses. In other words, scopes allow your application's users to limit the actions a third-party application can perform on their behalf.

Defining Scopes

You may define your API's scopes using the `Passport::tokensCan` method in the `boot` method of your `AuthServiceProvider`. The `tokensCan` method accepts an array of scope names and scope descriptions. The scope description may be anything you wish and will be displayed to users on the authorization approval screen:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

Default Scope

If a client does not request any specific scopes, you may configure your Passport server to attach a default scope to the token using the `setDefaultScope` method. Typically, you should call this method from the `boot` method of your `AuthServiceProvider`:

```
use Laravel\Passport\Passport;

Passport::setDefaultScope([
    'check-status',
    'place-orders',
]);
```

Assigning Scopes To Tokens

When Requesting Authorization Codes

When requesting an access token using the authorization code grant, consumers should specify their desired scopes as the `scope` query string parameter. The `scope` parameter should be a space-delimited list of scopes:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

When Issuing Personal Access Tokens

If you are issuing personal access tokens using the `User` model's `createToken` method, you may pass the array of desired scopes as the second argument to the method:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Checking Scopes

Passport includes two middleware that may be used to verify that an incoming request is authenticated with a token that has been granted a given scope. To get started, add the following middleware to the `$routeMiddleware` property of your `app/Http/Kernel.php` file:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Check For All Scopes

The `scopes` middleware may be assigned to a route to verify that the incoming request's access token has *all* of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
})->middleware('scopes:check-status,place-orders');
```

Check For Any Scopes

The `scope` middleware may be assigned to a route to verify that the incoming request's access token has *at least one* of the listed scopes:

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
})->middleware('scope:check-status,place-orders');
```

Checking Scopes On A Token Instance

Once an access token authenticated request has entered your application, you may still check if the token has a given scope using the `tokenCan` method on the authenticated `User` instance:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        //
    }
});
```

Additional Scope Methods

The `scopeIds` method will return an array of all defined IDs / names:

```
Laravel\Passport\Passport::scopeIds();
```

The `scopes` method will return an array of all defined scopes as instances of `Laravel\Passport\Scope`:

```
Laravel\Passport\Passport::scopes();
```

The `scopesFor` method will return an array of `Laravel\Passport\Scope` instances matching the given IDs / names:

```
Laravel\Passport\Passport::scopesFor(['place-orders', 'check-status']);
```

You may determine if a given scope has been defined using the `hasScope` method:

```
Laravel\Passport\Passport::hasScope('place-orders');
```

Consuming Your API With JavaScript

When building an API, it can be extremely useful to be able to consume your own API from your JavaScript application. This approach to API development allows your own application to consume the same API that you are sharing with the world. The same API may be consumed by your web application, mobile applications, third-party applications, and any SDKs that you may publish on various package managers.

Typically, if you want to consume your API from your JavaScript application, you would need to manually send an access token to the application and pass it with each request to your application. However, Passport includes a middleware that can handle this for you. All you need to do is add the `CreateFreshApiToken` middleware to your `web` middleware group in your `app/Http/Kernel.php` file:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

You should ensure that the `EncryptCookies` middleware is listed prior to the `CreateFreshApiToken` middleware in your middleware stack.

This Passport middleware will attach a `laravel_token` cookie to your outgoing responses. This cookie contains an encrypted JWT that Passport will use to authenticate API requests from your JavaScript application. Now, you may make requests to your application's API without explicitly passing an access token:

```
axios.get('/api/user')
    .then(response => {
        console.log(response.data);
    });
```

Customizing The Cookie Name

If needed, you can customize the `laravel_token` cookie's name using the `Passport::cookie` method. Typically, this method should be called from the `boot` method of your `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::cookie('custom_name');
}
```

CSRF Protection

When using this method of authentication, you will need to ensure a valid CSRF token header is included in your requests. The default Laravel JavaScript scaffolding includes an Axios instance, which will automatically use the encrypted `XSRF-TOKEN` cookie value to send a `X-XSRF-TOKEN` header on same-origin requests.

If you choose to send the `X-CSRF-TOKEN` header instead of `X-XSRF-TOKEN`, you will need to use the unencrypted token provided by `csrf_token()`.

Events

Passport raises events when issuing access tokens and refresh tokens. You may use these events to prune or revoke other access tokens in your database. You may attach listeners to these events in your application's `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],

    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

Testing

Passport's `actingAs` method may be used to specify the currently authenticated user as well as its scopes. The first argument given to the `actingAs` method is the user instance and the second is an array of scopes that should be granted to the user's token:

```
use App\User;
use Laravel\Passport\Passport;

public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(201);
}
```

Passport's `actingAsClient` method may be used to specify the currently authenticated client as well as its scopes. The first argument given to the `actingAsClient` method is the client instance and the second is an array of scopes that should be granted to the client's token:

```
use Laravel\Passport\Client;
use Laravel\Passport\Passport;

public function testGetOrders()
{
    Passport::actingAsClient(
        factory(Client::class)->create(),
        ['check-status']
    );

    $response = $this->get('/api/orders');

    $response->assertStatus(200);
}
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.