The file in app/config/packages/lucadegasperi/oauth2-server-laravel/oauth2.phpcontains plenty of options you can use to configure your OAuth 2.0 server implementation to suit the needs of your business. Here's the explanation for them.

**grant\_types**

options: array

An array of grant types supported by the authorization server to obtain an access token. See [Issuing access tokens](https://github.com/lucadegasperi/oauth2-server-laravel/wiki#authorization-server) for how to configure and use the different grant types.

**token\_type**

options: string

default: League\OAuth2\Server\TokenType\Bearer

This option informs the authorization server how the returned tokens should be formatted.

**state\_param**

options: true or false

default: false

If this option is true, each request to the authorization server should contain a&state=random\_string param. The state parameter is an additional security measure and the authorization server will reply back to your request with a response containing the same state param you passed. If the state param between the request and response doesn't match, the authorization server might have been compromised.

**scope\_param**

options: true or false

default: false

Whether or not the scope parameter is required in the query string. The scope(s) reflects the type of permission the client wants to access on user's behalf. See [Defining Scopes] for how to use scopes.

**scope\_delimiter**

options: string

default ','

The separator used to split the different scopes provided in the query string when multiple scopes are provided. See [Defining Scopes] for how to use scopes.

**default\_scope**

options: string or null

default: null

This option indicates the default scope each access token request has when no scope parameter is provided in the query string. null means the requests have no default scope. See [Defining Scopes] for how to use scopes.

**access\_token\_ttl**

options: integer

default: 3600

The number of seconds after an issued access token is not considered valid. Can be overwritten on a grant type basis.

**limit\_clients\_to\_grants**

options: true or false

default: false

This options sets whether or not a client is limited to specific grant types for obtaining an access token. The oauth\_client\_grants table regulates which clients can use which grant types.

**limit\_clients\_to\_scopes**

options: true or false

default: false

This options sets which clients can use which scopes. It is useful for allowing different grades of permissions to different clients. The oauth\_client\_scopes table regulates which clients can use which scopes.

**limit\_scopes\_to\_grants**

options: true or false

default: false

This options allows the use of certain scopes only when required with the appropriate grant type. this is due to the fact that different grant types have different grades of security. For example, a highly permissive scope should be allowed only to clients you trust or can request an access token securely. The oauth\_grant\_scopes table regulates which grant types can use which scopes.

**http\_headers\_only**

options: true or false

default: false

This options tells the resource server where to check for the access token. If set to true only the http headers will be checked.

# **Securing your API endpoints**

This package comes with a series of tools to help you protect your API endpoints using OAuth 2.0. This tools include the access token verification and the permissions verification. First let's talk about defining permissions (scopes).

**Defining scopes**

In the context of OAuth, scopes are the part of your API, the client (the third-party application) is trying to access. You can think of them as a sort of permission the client asks to have. Scopes are completely arbitrary string you define. When using this package, all your scopes should be saved into the oauth\_scopes table. When a client asks for an access token he'll pass the scopes he needs in order to work. The authorization server will then verify the scopes exist and the client has the right to use them.

Using scopes is optional, but any non trivial application will benefit from them.

**Checking the access token**

When requesting the resources of a protected endpoint the client should send an access token (previously issued to it) and the endpoint should check its validity. This is achieved by using the oauth before filter on any route of your API you want to protect with OAuth.

Route::get('protected-resource', ['before' => 'oauth', function() {

// return the protected resource

}]);

This filter will allow the access to the protected resource to any client with a valid access token. It will also send the client an error if he hasn't provided a valid access token. If you want to limit the access to the resource only to clients with certain permissions, here's where scopes come in handy.

**Checking the scopes**

Every access token is tied to the client, the resource owner and the scopes it can access. To check if the client can access a resource with its permission, use the oauth filter with the optional arguments. This will check the validity of the access token and the permissions.

Route::get('protected-resource', ['before' => 'oauth:scope1,scope2', function() {

// return the protected resource

}]);

When at least one of the scope doesn't match the permissions the client has, the filter will return an error to the client, informing it that it doesn't have the required permissions to access the endpoint.

# **Choosing a Grant**

OAuth 2.0 by it’s nature is a very flexible standard and can be adapted to work in many different scenarios. The [core specification](http://tools.ietf.org/html/rfc6749) describes four authorisation grants:

* Authorisation code grant
* Implicit grant
* Resource owner credentials grant
* Client credentials grant

The specification also details another grant called the refresh token grant.

Furthermore there are a number of other grants that have gone through the IETF ratification process (none of which at the time of writing have been formally standardised):

* Message authentication code (MAC) tokens
* SAML 2.0 Bearer Assertion Profiles
* JSON web token grant

The end goal of each of these grants (except the refresh token grant) is for the client application to have an access token (which represents a user’s permission for the client to access their data) which it can use to authenticate a request to an API endpoint.

This page describes each of the above grants and their appropriate use cases.

As a refresher here is a quick glossary of OAuth terms (taken from the core spec):

* **Resource owner (a.k.a. the User)** - An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
* **Resource server (a.k.a. the API server)** - The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
* **Client** - An application making protected resource requests on behalf of the resource owner and with its authorisation. The term client does not imply any particular implementation characteristics (e.g. whether the application executes on a server, a desktop, or other devices).
* **Authorisation server** - The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorisation.

## Authorisation code grant ([section 4.1](http://tools.ietf.org/html/rfc6749#section-4.1))

To enable this grant:

'grant\_types' => [

'authorization\_code' => [

'class' => '\League\OAuth2\Server\Grant\AuthCodeGrant',

'access\_token\_ttl' => 3600,

'auth\_token\_ttl' => 60

]

]

The authorisation code grant is the grant that most people think of when OAuth is described.

If you’ve ever signed into a website or application with your Twitter/Facebook/Google/(insert major Internet company here) account then you’ll have experienced using this grant.

Essentially a user will click on a “sign in with Facebook” (or other IdP) and then be redirected from the application/website (the “client”) to the IdP authorisation server. The user will then sign in to the IdP with their credentials, and then - if they haven’t already - authorise the client to allow it to use the user’s data (such as their name, email address, etc). If they authorise the request the user will be redirected back to the client with a token (called the authorisation code) in the query string (e.g. http://client.com/redirect?code=XYZ123) which the client will capture and exchange for an access token in the background.

This grant is suitable where the resource owner is a user and they are using a client which is allows a user to interact with a website in a browser. An obvious example is the client being another website, but desktop applications such as Spotify or Reeder use embedded browsers.

Some mobile applications use this flow and again use an embedded browser (or redirect the user to the native browser and then are redirected back to the app using a custom protocol).

In this grant the access token is kept private from the resource owner.

If you have a mobile application that is for your own service (such as the official Spotify or Facebook apps on iOS) it isn’t appropriate to use this grant as the app itself should already be trusted by your authorisation server and so the \_resource owner credentials grant would be more appropriate.

## Implicit grant ([section 4.2](http://tools.ietf.org/html/rfc6749#section-4.2))

The implicit grant is similar to the authentication code grant described above. The user will be redirected in a browser to the IdP authorisation server, sign in, authorise the request but instead of being returned to the client with an authentication code they are redirected with an access token straight away.

The purpose of the implicit grant is for use by clients which are not capable of keeping the client’s own credentials secret; for example a JavaScript only application.

**If you decide to implement this grant then you must be aware that the access token should be treated as “public knowledge” (like a public RSA key)** and therefore it must have a very limited permissions when interacting with the API server. For example an access token that was granted using the authentication code grant could have permission to be used to delete resources owned by the user, however an access token granted through the implicit flow should only be able to “read” resources and never perform any destructive operations (i.e. non-idempotent method).

## Resource owner credentials grant ([section 4.3](http://tools.ietf.org/html/rfc6749#section-4.3))

To enable this grant:

'grant\_types' => [

'password' => [

'class' => '\League\OAuth2\Server\Grant\PasswordGrant',

'callback' => function($username, $password) {

return Auth::validate([

'email' => $username,

'password' => $password,

]);

},

'access\_token\_ttl' => 3600

]

]

When this grant is implemented the client itself will ask the user for their username and password (as opposed to being redirected to an IdP authorisation server to authenticate) and then send these to the authorisation server along with the client’s own credentials. If the authentication is successful then the client will be issued with an access token.

This grant is suitable for trusted clients such as a service’s own mobile client (for example Spotify’s iOS app). You could also use this in software where it’s not easy to implement the authorisation code - for example we bolted this authorisation grant into [OwnCloud](http://owncloud.org/) so we could retrieve details about a user that we couldn’t access over LDAP from the university’s Active Directory server.

## Client credentials grant ([section 4.4](http://tools.ietf.org/html/rfc6749#section-4.4))

To enable this grant:

'grant\_types' => [

'client\_credentials' => [

'class' => '\League\OAuth2\Server\Grant\ClientCredentialsGrant',

'access\_token\_ttl' => 3600

]

]

This grant is similar to the resource owner credentials grant except only the client’s credentials are used to authenticate a request for an access token. Again this grant should only be allowed to be used by trusted clients.

This grant is suitable for machine-to-machine authentication, for example for use in a cron job which is performing maintenance tasks over an API. Another example would be a client making requests to an API that don’t require user’s permission.

When someone visits a member of staff’s page on the [University of Lincoln staff directory](http://staff.lincoln.ac.uk/)the website uses it’s own access token (that was generated using this grant) to authenticate a request to the API server to get the data about the member of staff that is used to build the page. When a member of staff signs in to update their profile however their own access token is used to retrieve and update their data. Therefore there is a good separation of concerns and we can easily restrict permissions that each type of access token has.

## Refresh token grant ([section 1.5](http://tools.ietf.org/html/rfc6749#section-1.5))

To enable this grant:

'grant\_types' => [

'refresh\_token' => [

'class' => '\League\OAuth2\Server\Grant\RefreshTokenGrant',

'access\_token\_ttl' => 3600,

'refresh\_token\_ttl' => 36000

]

]

The OAuth 2.0 specification also details a fifth grant which can be used to “refresh” (i.e. renew) an access token which has expired.

Authorisation servers which support this grant will also issue a “refresh token” when it returns an access token to a client. When the access token expires instead of sending the user back through the authorisation code grant the client can use to the refresh token to retrieve a new access token with the same permissions as the old one.

A problem with the grant is that it means the client has to maintain state of each token and then either on a cron job keep access tokens up to date or when it tries to make a request and it fails then go and update the access token and repeat the request.

This page was originally posted at <http://alexbilbie.com/2013/02/a-guide-to-oauth-2-grants/>. It has been modified to suite this package configuration.