

[How to Add JWT Authentication to Laravel APIs](#)

How to Add JWT Authentication to Laravel APIs

JWT (JSON Web Tokens) is a stateless auth mechanism ideal for APIs. Clients authenticate once, receive a signed token, and send it in the `Authorization: Bearer <token>` header. In this guide you'll install a JWT library, issue/verify tokens, protect routes, refresh/blacklist tokens, and build a tiny UI to test everything.

1 - Install & Configure the JWT Package

We'll use the popular `tymon/jwt-auth` package which provides guards, middleware, and helpers for issuing/validating JWTs.

```
composer require tymon/jwt-auth
```

```
php artisan vendor:publish --  
provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

```
php artisan jwt:secret language: Bash (bash)
```

Publishing `config/jwt.php` into your app. Running `jwt:secret` sets `JWT_SECRET` in `.env` and generates a signing key. Keep it private; changing it invalidates all existing tokens.

2 - Switch API Guard to JWT

Tell Laravel to use the JWT guard for API routes so `auth:api` resolves the bearer token into the authenticated user.

```
// config/auth.php (snippets)
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'jwt',    // <-- use the JWT driver
        'provider' => 'users',
    ],
],

'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\User::class,
    ],
],Code language: PHP (php)
```

The `jwt` driver hooks into request lifecycle to validate and decode tokens automatically, making `auth('api')` and `$request->user()` work as usual for protected endpoints.

3 - Auth Controller: Login, Me, Logout, Refresh

Create a dedicated controller for token issuance and lifecycle operations. We'll validate credentials, return a signed JWT, expose a `me` endpoint, invalidate tokens on logout, and refresh tokens before expiry.

```
// app/Http/Controllers/JwtAuthController.php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class JwtAuthController extends Controller
{
    public function login(Request $request)
    {
        $credentials = $request->validate([
            'email' => ['required','email'],
            'password' => ['required']
        ]);

        if (! $token = Auth::guard('api')->attempt($credentials)) {
            return response()->json(['message' => 'Invalid
credentials'], 401);
        }

        return $this->respondWithToken($token);
    }

    public function me()
    {
        return response()->json(Auth::guard('api')->user());
    }

    public function logout()
    {
        Auth::guard('api')->logout(); // adds token to blacklist if
enabled
        return response()->json(['message' => 'Logged out']);
    }

    public function refresh()
    {
        return $this->respondWithToken(Auth::guard('api')->refresh());
    }
}
```

```
}

protected function respondWithToken(string $token)
{
    return response()->json([
        'access_token' => $token,
        'token_type'   => 'bearer',
        'expires_in'   => Auth::guard('api')->factory()->getTTL()
* 60
    ]);
}
}Code language: PHP (php)
```

`attempt()` verifies credentials and returns a JWT on success. `logout()` invalidates the current token (blacklists it if configured). `refresh()` issues a new token using a still-valid one, extending the session without re-entering credentials.

4 - Routes & Middleware

Expose auth endpoints and protect your API routes with `auth:api`. You can also add throttling to slow down brute-force attempts.

```
// routes/api.php
use App\Http\Controllers\JwtAuthController;
use App\Http\Controllers\PostApiController;

Route::post('/auth/login', [JwtAuthController::class,
'login'])->middleware('throttle:20,1');
Route::post('/auth/refresh', [JwtAuthController::class, 'refresh']);
Route::post('/auth/logout', [JwtAuthController::class,
'logout'])->middleware('auth:api');
Route::get('/auth/me', [JwtAuthController::class,
'me'])->middleware('auth:api');
```

```
// Example protected resource
Route::middleware('auth:api')->group(function () {
    Route::get('/posts', [PostApiController::class, 'index']);
    Route::post('/posts', [PostApiController::class, 'store']);
});Code language: PHP (php)
```

Only authenticated requests with a valid bearer token can reach the protected group. The login route is throttled to mitigate password spraying; adjust limits for your environment.

5 - Token TTL, Refresh & Blacklist

Configure expiry and blacklist behavior to balance security and UX. Short TTL + refresh rotation is a good default.

```
// config/jwt.php (snippets)
'ttl' => env('JWT_TTL', 60), // minutes, e.g. 60
'refresh_ttl' => env('JWT_REFRESH_TTL', 20160), // minutes (14 days)
'blacklist_enabled' => env('JWT_BLACKLIST_ENABLED', true),Code language:
PHP (php)
```

`ttl` controls access token lifetime. `refresh_ttl` controls how long a token can be refreshed. With blacklist enabled, `logout()` immediately invalidates the token even if it hasn't expired, preventing reuse.

6 - Optional: Add Custom Claims

You can embed extra, non-sensitive info in the JWT (e.g., role, plan). Use claims for

authorization hints—not for secrets.

```
// app/Models/User.php (snippet)
use Tymon\JWTAuth\Contracts\JWTSubject;

class User extends Authenticatable implements JWTSubject
{
    // ...

    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    public function getJWTCustomClaims()
    {
        return [
            'role' => $this->role ?? 'user',
            'plan' => $this->plan ?? 'free',
        ];
    }
}
}Code language: PHP (php)
```

Implementing JWTSubject allows the library to serialize the user into the token. Custom claims travel with the token and are available after decode, handy for quick policy checks.

7 - CORS & Frontend Usage

SPAs/mobile apps must send the bearer token on every request. If your frontend runs on a different origin, enable CORS so browsers allow the calls.

```
// app/Http/Kernel.php (snippet)
protected $middleware = [
```

```
// ...  
\Fruitcake\Cors\HandleCors::class,  
];Code language: PHP (php)
```

Use the official CORS middleware (already present in new Laravel apps). Configure allowed origins, methods, and headers in `config/cors.php`. Always send Authorization: Bearer <token> from the client.

8 - UI: Minimal JWT Tester

This tiny page logs in to get a token, calls a protected endpoint, refreshes the token, and logs out — all from the browser.

```
<!-- resources/views/jwt/tester.blade.php -->  
@extends('layouts.app')  
  
@section('content')  
<div class="container">  
  <h1>JWT Tester</h1>  
  
  <div class="row g-2">  
    <div class="col-md-4"><input id="email" class="form-control"  
placeholder="Email"></div>  
    <div class="col-md-4"><input id="password" type="password"  
class="form-control" placeholder="Password"></div>  
  </div>  
  
  <div class="mt-3">  
    <button class="btn btn-theme" onclick="login()">Login</button>  
    <button class="btn btn-secondary ms-2" onclick="me()">Call  
/auth/me</button>  
    <button class="btn btn-warning ms-2" onclick="refresh()">Refresh  
Token</button>
```

```
<button class="btn btn-danger ms-2"
onclick="logout()">Logout</button>
</div>
```

```
<pre id="out" class="mt-3"></pre>
</div>
```

```
<script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script>
let token = null;
```

```
function login() {
  axios.post('/api/auth/login', {
    email: document.getElementById('email').value,
    password: document.getElementById('password').value
  }).then(r => {
    token = r.data.access_token;
    document.getElementById('out').textContent =
JSON.stringify(r.data, null, 2);
  }).catch(e => out(e));
}
```

```
function me() {
  axios.get('/api/auth/me', { headers: { Authorization: `Bearer
${token}` } })
  .then(r => document.getElementById('out').textContent =
JSON.stringify(r.data, null, 2))
  .catch(e => out(e));
}
```

```
function refresh() {
  axios.post('/api/auth/refresh', {}, { headers: { Authorization:
`Bearer ${token}` } })
  .then(r => { token = r.data.access_token;
document.getElementById('out').textContent = JSON.stringify(r.data,
null, 2) })
  .catch(e => out(e));
}
```

```
}

function logout() {
  axios.post('/api/auth/logout', {}, { headers: { Authorization:
`Bearer ${token}` }})
    .then(r => document.getElementById('out').textContent =
JSON.stringify(r.data, null, 2))
    .catch(e => out(e));
}

function out(e) {
  document.getElementById('out').textContent = e.response ?
JSON.stringify(e.response.data, null, 2) : e;
}
</script>
@endsectionCode language: HTML, XML (xml)
```

The page demonstrates the full lifecycle: obtain, use, refresh, and revoke tokens. Perfect for quick end-to-end verification before wiring up your SPA or mobile client.

Wrapping Up

You added JWT authentication to a Laravel API using `tymon/jwt-auth`, switched the guard, built endpoints to login/refresh/logout, protected routes, and tested the flow with a simple UI. JWT keeps servers stateless and scales well. Combine it with HTTPS, short TTLs, token rotation, and blacklist on logout to balance security and usability.

What's Next

- [Building a Mobile App Backend with Laravel API](#)
- [Integrating Laravel with Third-Party APIs \(Mail, SMS, Payment\)](#)
- [How to Build a Multi-Auth API with Sanctum](#)