

GUÍA PARA DESARROLLADORES

Integrar tu app con este servidor de identidad

Cómo conectar Angular, NestJS, Flutter (o cualquier otro stack) a este servidor OAuth2 para usarlo como IAM centralizado: login, usuarios, roles globales y roles por-app.

Laravel 12 · Passport 13 (OAuth2, tokens JWT firmados RS256)
Generado el 25 de julio de 2026

Índice

1. Qué te da este servidor
2. Paso 1 — Registrar tu app como cliente OAuth
3. Elegí el flujo según tu arquitectura
4. Angular (SPA) — Authorization Code + PKCE
5. NestJS — backend con login de usuarios
6. NestJS — servicio backend-a-backend (Client Credentials)
7. Flutter / apps móviles
8. Validar el token en tu backend
9. Roles: global vs. por-app
10. Refrescar y revocar tokens
11. Checklist de seguridad
12. Referencia rápida

1. Qué te da este servidor

Es un **servidor de identidad OAuth2** (Laravel Passport). En vez de que cada app tuya tenga su propia tabla de usuarios y su propio login, todas hablan con este servidor: acá vive el usuario, la contraseña, y quién es qué.

- **Login centralizado** — un solo lugar donde el usuario pone su contraseña. Tu app nunca la ve: recibe un *access token* después de que el usuario aprueba el acceso.
- **Datos de usuario vía API** — `GET /api/user` con el token te devuelve nombre, email, y sus roles.
- **Roles globales y roles por-app** — un usuario puede ser `admin` en el sistema en general, pero además tener un rol *específico de tu app* (por ejemplo `editor` en la tuya y nada en la de otro equipo). Ver sección 9.
- **Gestión completa por API** — si querés armar tu propio panel de administración de usuarios (en vez de usar el que trae este servidor), toda la funcionalidad está expuesta en `/api/*` con Bearer token. No hace falta sesión ni cookies de este servidor.

Dos capas de permisos, no confundirlas: los *scopes* del token OAuth (`users:read` , `users:write` ...) definen qué puede hacer *ese token* contra la API de este servidor. Los *roles por-app* definen qué puede hacer *ese usuario dentro de tu app* — son datos que vos leés e interpretás como quieras, este servidor no te impone nada sobre cómo usarlos.

2. Paso 1 — Registrar tu app como cliente OAuth

Cada aplicación que se conecte necesita ser un *cliente* registrado. Se hace una sola vez, desde el panel (`/clients` , con cualquier cuenta) o por API:

```
curl -X POST http://localhost:8000/api/clients \
-H "Authorization: Bearer {tu_access_token}" \
-H "Content-Type: application/json" \
-d '{
  "name": "Mi App Angular",
  "grant_type": "authorization_code",
  "redirect_uris": ["https://mi-app.com/callback"],
  "confidential": false
}'
```

La respuesta trae `client.id` (el `client_id`) y, si es confidencial, `plain_secret` — se muestra una sola vez, guardalo ya.

Campo	Qué significa
<code>grant_type</code>	<code>authorization_code</code> , <code>client_credentials</code> o <code>password</code> — ver sección 3 para elegir.
<code>redirect_uris</code>	A dónde te devuelve el navegador después del login. Tiene que matchear exacto con lo que mandes en <code>/oauth/authorize</code> .
<code>confidential</code>	<code>false</code> = cliente público, sin secret (SPA, apps móviles — usan PKCE en su lugar). <code>true</code> = confidencial, con secret (backends que pueden guardar un secreto de forma segura).

3. Elegí el flujo según tu arquitectura

SPA pura (Angular, React, Vue)

Sin backend propio, corre 100% en el navegador.

Authorization Code + PKCE, cliente público. El token queda en el navegador.

App con backend (NestJS, Express)

El backend maneja el login por vos (patrón BFF).

Authorization Code, cliente confidencial — el secret vive en el backend, nunca llega al navegador.

App móvil (Flutter, RN)

Authorization Code + PKCE, cliente público, con un redirect por deep link / custom scheme.

Servicio sin usuario final

Un cron, un worker, otro backend que llama a la API por su cuenta. **Client Credentials**.

Password Grant existe pero evitalo salvo que sea estrictamente first-party (una app que vos mismo controlás de punta a punta): tu app le pide el usuario y contraseña directamente al usuario y se los manda a este servidor. Ningún cliente de terceros debería tocar la contraseña del usuario.

4. Angular (SPA) — Authorization Code + PKCE

Ejemplo con `angular-oauth2-oidc` (también podés implementarlo a mano, son cuatro pasos: generar verifier/challenge, redirigir, recibir el `code`, canjearlo).

APP.CONFIG.TS

```
import { provideOAuthClient, OAuthModule } from 'angular-oauth2-oidc';

export const authConfig = {
  issuer: 'http://localhost:8000',
  loginUrl: 'http://localhost:8000/oauth/authorize',
  tokenEndpoint: 'http://localhost:8000/oauth/token',
  redirectUri: window.location.origin + '/callback',
  clientId: '{tu_client_id}',
  responseType: 'code',
  scope: 'users:read account:read',
  useHttpBasicAuth: false,
  disablePKCE: false, // clave: PKCE prendido, es cliente público
  oidc: false, // este servidor es OAuth2 puro, no expone id_token/OIDC
};
```

AUTH.SERVICE.TS

```
import { Injectable } from '@angular/core';
import { OAuthService } from 'angular-oauth2-oidc';

@Injectable({ providedIn: 'root' })
export class AuthService {
  constructor(private oauth: OAuthService) {}

  login() {
    this.oauth.initCodeFlow(); // redirige a /oauth/authorize con PKCE
  }

  async me() {
    const token = this.oauth.getAccessToken();
    const res = await fetch('http://localhost:8000/api/user', {
      headers: { Authorization: `Bearer ${token}` },
    });
    return res.json(); // { id, name, email, roles, app_roles, ... }
  }
}
```

Si preferís no traer una librería, el flujo manual es: generar `code_verifier` (random) y `code_challenge` (SHA-256 + base64url del verifier), armar la URL de `/oauth/authorize` con esos parámetros + `state`, y en el callback hacer `POST /oauth/token` con `grant_type=authorization_code`, el `code` recibido y el `code_verifier` original.

5. NestJS — backend con login de usuarios

Tu backend actúa de cliente **confidencial** (guarda el secret) y maneja el flujo Authorization Code por el usuario — patrón "backend for frontend": el navegador nunca ve el access token, solo una cookie de sesión propia de tu app.

AUTH.CONTROLLER.TS

```
@Controller('auth')
export class AuthController {
  private readonly IAM = 'http://localhost:8000';
  private readonly CLIENT_ID = process.env.IAM_CLIENT_ID;
  private readonly CLIENT_SECRET = process.env.IAM_CLIENT_SECRET;

  @Get('login')
  login(@Res() res: Response) {
    const params = new URLSearchParams({
      client_id: this.CLIENT_ID,
      redirect_uri: 'https://mi-backend.com/auth/callback',
      response_type: 'code',
      scope: 'users:read',
      state: crypto.randomUUID(),
    });
    res.redirect(`${this.IAM}/oauth/authorize?${params}`);
  }

  @Get('callback')
  async callback(@Query('code') code: string, @Session() session: Record) {
    const { data } = await axios.post(`${this.IAM}/oauth/token`, {
      grant_type: 'authorization_code',
      client_id: this.CLIENT_ID,
      client_secret: this.CLIENT_SECRET,
      code,
      redirect_uri: 'https://mi-backend.com/auth/callback',
    });

    session.accessToken = data.access_token; // guardalo del lado del servidor
    session.refreshToken = data.refresh_token;
  }
}
```

Como es cliente confidencial, no hace falta PKCE (aunque no molesta agregarlo igual). El token queda guardado en la sesión del backend, nunca expuesto al navegador.

6. NestJS — servicio backend-a-backend (Client Credentials)

Para un servicio que consulta/gestiona usuarios por su cuenta, sin un usuario logueado de por medio — scheduler, worker, integración entre sistemas.

IAM-CLIENT.SERVICE.TS

```
@Injectable()
export class IamClientService {
  private token: string | null = null;
  private expiresAt = 0;

  private async getToken(): Promise {
    if (this.token && Date.now() < this.expiresAt) return this.token;

    const { data } = await axios.post('http://localhost:8000/oauth/token', {
      grant_type: 'client_credentials',
      client_id: process.env.IAM_SERVICE_CLIENT_ID,
      client_secret: process.env.IAM_SERVICE_CLIENT_SECRET,
      scope: 'users:read users:write',
    });

    this.token = data.access_token;
    this.expiresAt = Date.now() + (data.expires_in - 60) * 1000; // renovar 1 min antes
    return this.token;
  }

  async listUsers() {
    const token = await this.getToken();
    const { data } = await axios.get('http://localhost:8000/api/users', {
      headers: { Authorization: `Bearer ${token}` },
    });
    return data;
  }
}
```

Un token de `client_credentials` no tiene usuario asociado: actúa como "servicio de confianza". En este servidor eso significa que puede usar `/api/users` y `/api/apps` con el scope como único chequeo (sin necesitar un permiso de usuario admin) — pero **no** puede tocar recursos "de cuenta propia" (`/api/clients`, personal access tokens, apps autorizadas), porque esos son inherentemente de un usuario y no tiene ninguno asociado.

7. Flutter / apps móviles

Con el paquete `flutter_appauth` (implementa Authorization Code + PKCE con el navegador del sistema, que es lo recomendado — evita usar un WebView embebido).

AUTH_SERVICE.DART

```
final appAuth = FlutterAppAuth();

Future<String?> login() async {
  final result = await appAuth.authorizeAndExchangeCode(
    AuthorizationTokenRequest(
      '{tu_client_id}',
      'com.miapp://callback', // custom scheme registrado en tu app
      serviceConfiguration: const AuthorizationServiceConfiguration(
        authorizationEndpoint: 'http://localhost:8000/oauth/authorize',
        tokenEndpoint: 'http://localhost:8000/oauth/token',
      ),
      scopes: ['users:read'],
    ),
  );

  return result.accessToken;
}

Future<Map<String, dynamic>> me(String token) async {
  final res = await http.get(
    Uri.parse('http://localhost:8000/api/user'),
    headers: {'Authorization': 'Bearer $token'},
  );
  return jsonDecode(res.body);
}
```

El `redirect_uri` (`com.miapp://callback`) tiene que estar registrado exactamente igual en el cliente OAuth (sección 2) y configurado en tu app nativa (Android: intent filter; iOS: URL scheme en el `Info.plist`).

8. Validar el token en tu backend

Cuando tu backend recibe un request con `Authorization: Bearer {token}` y necesita confirmar que es válido, hay dos caminos:

Opción A — Preguntarle a este servidor (más simple)

```
GET /api/user
Authorization: Bearer {token}
```

Si responde `200` con los datos del usuario, el token es válido. Es un request de red extra por cada llamada, pero es siempre exacto: refleja revocaciones al instante.

Opción B — Validar el JWT localmente (sin red)

Los access tokens de este servidor son **JWT firmados con RS256** (no opacos). Tu backend puede verificar la firma con la clave pública de Passport (`storage/oauth-public.key` en este proyecto — pedísela al equipo que administra este servidor) sin llamarlo por red:

```
// Node.js, con la librería "jsonwebtoken"
import jwt from 'jsonwebtoken';
import fs from 'fs';

const publicKey = fs.readFileSync('oauth-public.key');

function verify(token: string) {
  return jwt.verify(token, publicKey, { algorithms: ['RS256'] });
  // -> { aud, jti, iat, nbf, exp, sub, scopes: string[] }
}
```

Claims relevantes: `sub` = id del usuario (ausente en tokens de `client_credentials`), `aud` = `client_id` que lo pidió, `scopes` = array de scopes otorgados, `exp` = expiración (unix timestamp).

Trade-off: la validación local es más rápida (sin red) pero no se entera si el token fue *revocado* antes de su expiración natural — solo valida firma y expiración. Para casos sensibles (borrar acceso ya mismo), usá la Opción A o combiná: validación local + chequeo periódico contra el servidor.

9. Roles: global vs. por-app

GET `/api/user` (o `/api/users/{id}`) devuelve, entre otros campos:

```
{
  "id": 2,
  "name": "Carla Gómez",
  "email": "carla@example.com",
  "roles": ["user"],
  "app_roles": ["editor"]
}
```

- `roles` — el rol **global** del usuario en este servidor (`admin` o `user`). No dice nada sobre qué puede hacer en *tu* app.
- `app_roles` — los roles que un admin le asignó a ese usuario **específicamente para el cliente OAuth que hizo la request** (o sea: tu app). Se resuelve solo, según el `client_id` del token usado — no hace falta pedir nada extra.

Un admin asigna estos roles desde `/admin/users` → editar usuario → "Roles por app", eligiendo tu app de la lista. Interpretalos como quieras del lado tuyo (un guard de NestJS, un route guard de Angular, etc.) — este servidor solo los guarda y te los devuelve.

NESTJS GUARD (EJEMPLO)

```
@Injectable()
export class AppRoleGuard implements CanActivate {
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const req = context.switchToHttp().getRequest();
    const user = await this.iam.me(req.headers.authorization);
    return user.app_roles.includes('editor');
  }
}
```

10. Refrescar y revocar tokens

Los access tokens expiran (15 días por defecto en este servidor). Si tu cliente pidió el scope `refresh_token` implícito del grant (viene incluido en `authorization_code` y `password`), la respuesta de `/oauth/token` trae un `refresh_token` — usalo así, sin volver a pasar por el login:

```
curl -X POST http://localhost:8000/oauth/token \  
  -d grant_type=refresh_token \  
  -d refresh_token={tu_refresh_token} \  
  -d client_id={tu_client_id} \  
  -d client_secret={tu_client_secret}  # si tu cliente es confidencial
```

`client_credentials` no tiene refresh token — simplemente pedís uno nuevo cuando el anterior expira (ver el ejemplo de NestJS en la sección 6).

El usuario puede revocarte el acceso en cualquier momento desde `/authorized-apps` en el panel de este servidor — a partir de ahí tus requests con ese token empiezan a devolver `401`.

11. Checklist de seguridad

- ☐ Clientes públicos (SPA, móvil) — **siempre PKCE**, nunca les asignes un secret que tendrían que embeber en código que corre en el dispositivo del usuario.
- ☐ Pedí solo los **scopes que realmente necesitás** — no hay scope por defecto en este servidor, es intencional.
- ☐ En una SPA, si podés, no guardes el access token en `localStorage` (vulnerable a XSS) — preferí el patrón backend-for-frontend (sección 5) con cookie `httpOnly`.
- ☐ Verificá siempre el parámetro `state` en el callback de Authorization Code (protección CSRF del flujo).
- ☐ Los `client_secret` de clientes confidenciales van en variables de entorno del backend, nunca en el repo ni en el bundle del frontend.
- ☐ Si tu servicio backend-a-backend solo necesita leer, pedile a un admin que te dé un cliente con scope `users:read` nada más (no `users:write`).

12. Referencia rápida

Endpoint	Para qué
<code>GET /oauth/authorize</code>	Inicio del flujo Authorization Code (redirige al login/consentimiento).
<code>POST /oauth/token</code>	Canjear código, credenciales o refresh token por un access token.
<code>GET /api/user</code>	Whoami — usuario del token actual + roles global y por-app.
<code>GET/POST/PUT/DELETE /api/users</code>	Directorio de usuarios (scope <code>users:read</code> / <code>write</code>).
<code>GET/POST/DELETE /api/users/{id}/app-roles</code>	Roles por-app de un usuario (para asignarlos vos mismo por API, si tenés permiso).
<code>GET /api/apps</code>	Listado de apps (clientes OAuth) registradas.
<code>GET/POST/PUT/DELETE /api/clients</code>	Gestionar tus propios clientes OAuth por API.

Scope	Da acceso a
<code>users:read</code>	Leer usuarios y sus roles (global + por-app).
<code>users:write</code>	Crear/editar/borrar usuarios, asignar roles por-app.
<code>account:read</code>	Leer tus propios clientes OAuth, tokens y apps autorizadas.
<code>account:write</code>	Gestionar tus propios clientes OAuth, tokens y apps autorizadas.

Documentación interactiva (Swagger, para probar todo esto desde el navegador) en `/api/documentation` . Manual completo del panel de administración en `docs/manual-usuario.pdf` .