Implement OAuth for Okta with a Service App

What You'll Accomplish

By the end of this guide, you'll have a working OAuth 2.0 service app that can securely interact with Okta's APIs. Specifically, you'll be able to create an API Services app in Okta, set up key-based authentication, assign appropriate permissions, and make API calls to manage user profiles.

Understanding the Basics

Before we dive in, let's clarify a few important concepts:

Roles versus Scopes

Think of roles and scopes as two layers of security that work together. A role determines what your application is allowed to do within Okta's admin console, such as which users it can access or which operations it can perform. You can narrow this down even further using Resource Sets to limit access to specific groups of users.

Scopes, on the other hand, control what an access token can do when making API calls. For example, your token needs scopes like **okta.users.read** or **okta.users.manage** actually to call user-related endpoints.

Here's the key point: you need both working together. Your app must have a role that permits the action, and your access token must contain the right scope. If either piece is missing, you'll get blocked with a 403 error.

Client Credentials Flow

This is the authentication method for machine-to-machine communication. There's no human involved; your service authenticates itself using a private key and receives a temporary access token in return.

Private Key JWT Authentication

Instead of using a client secret, your application signs a JWT with its private key. Okta verifies this signature using the public key you've registered. This approach is more secure than traditional client secrets.

A Note on DPoP

DPoP is an additional security feature that prevents tokens from being reused on different servers. It's useful for mobile apps or browser-based clients, but typically unnecessary for server-to-server scenarios where you already have network-level security controls in place. For most backend services, you can safely leave this disabled.

Setting Up Your Okta Service App: A Step-by-Step Guide

Step 1: Create Your Service App

Log in to your Okta Admin Console and navigate to Applications, then click Create App Integration. Choose "API Services" as your application type and hit Next. Give your app a meaningful name, something like "users-svc-m2m" works well. Once you save, Okta automatically configures it to use the Client Credentials grant type.

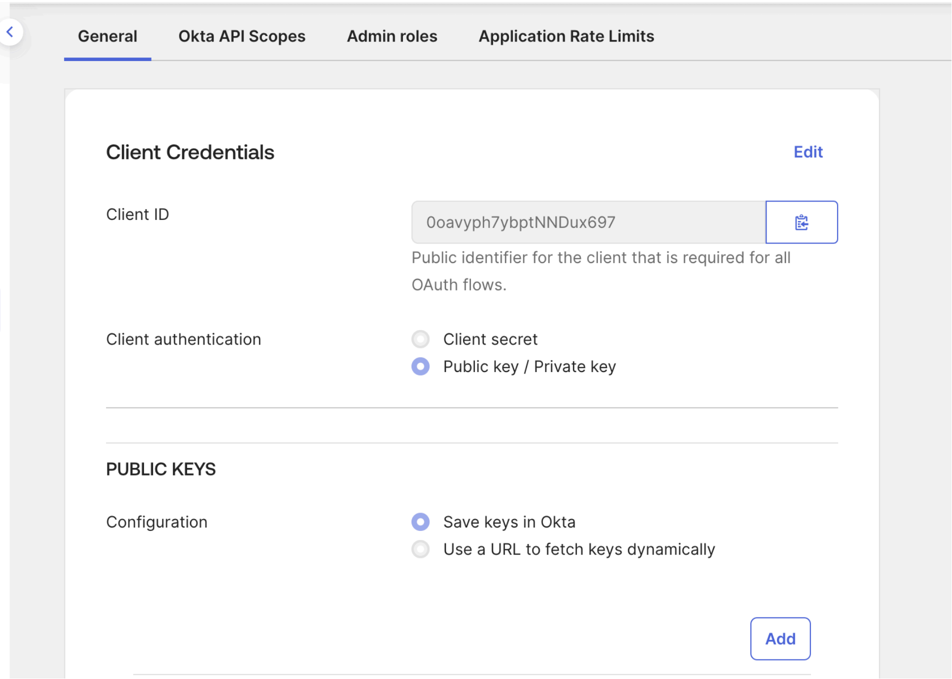
Step 2: Set Up Key-Based Authentication

For production environments, you'll want to generate an RSA 2048 key pair. Keep your private key secured in a secret management system and prepare your public key in JWK format with a key ID.

Here's how to register your keys in Okta:

Open your app's General settings and find the Client Credentials section. Click Edit, then change the client authentication method to "Public key / Private key." You can either paste your existing public key in JWKS format or have Okta generate a new key pair for you. If you choose the generation option, make sure to copy and securely store the private key immediately.

Make note of three important pieces of information: your client ID, the key ID for your signing key, and your private key in PEM format.



Client Credentials

Client ID: 0oavyph7ybptNNDux697

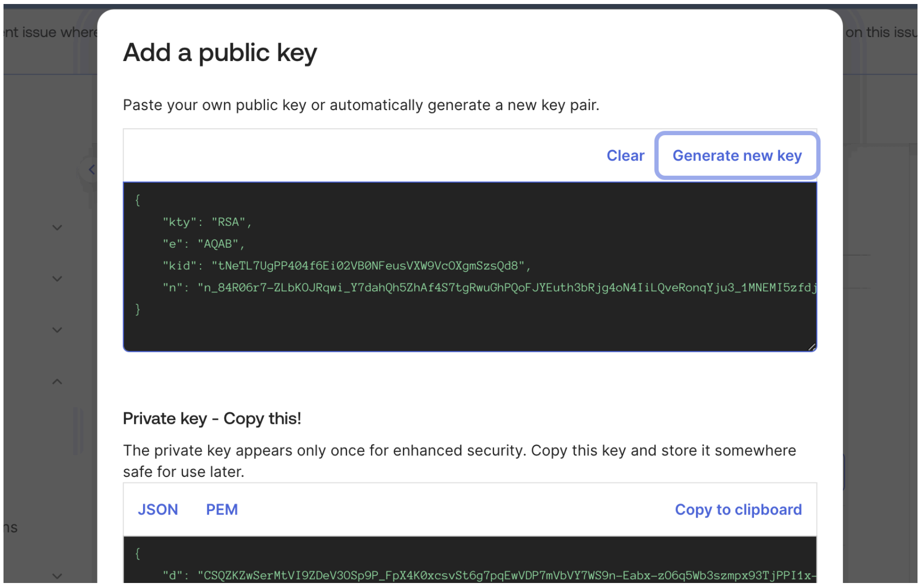
Public identifier for the client that is required for all OAuth flows.

Client authentication: ○ Client secret ● Public key / Private key

PUBLIC KEYS

Configuration: ● Save keys in Okta ○ Use a URL to fetch keys dynamically

Fig 1: Create Public key/Private key



Add a public key

Paste your own public key or automatically generate a new key pair.

```
{
    "kty": "RSA",
    "e": "AQAB",
    "kid": "tNeTL7UgPP404f6Ei02VB0NFeusVXW9VcOXgmSzsQd8",
    "n": "n_84R06r7-ZLbKOJRqwi_Y7dahQh5ZhAf4S7tgRwuGhPQoFJYEuth3bRjg4oN4IiLQveRonqYju3_1MNEMI5zfdj
}
```

Private key - Copy this!

The private key appears only once for enhanced security. Copy this key and store it somewhere safe for use later.

```
{
    "d": "CSQZKZwSerMtVI9ZDeV3OSp9P_FpX4K0xcsvSt6g7pqEwVDP7mVbVY7WS9n-Eabx-z06q5Wb3szmpx93TjPPI1x-
```

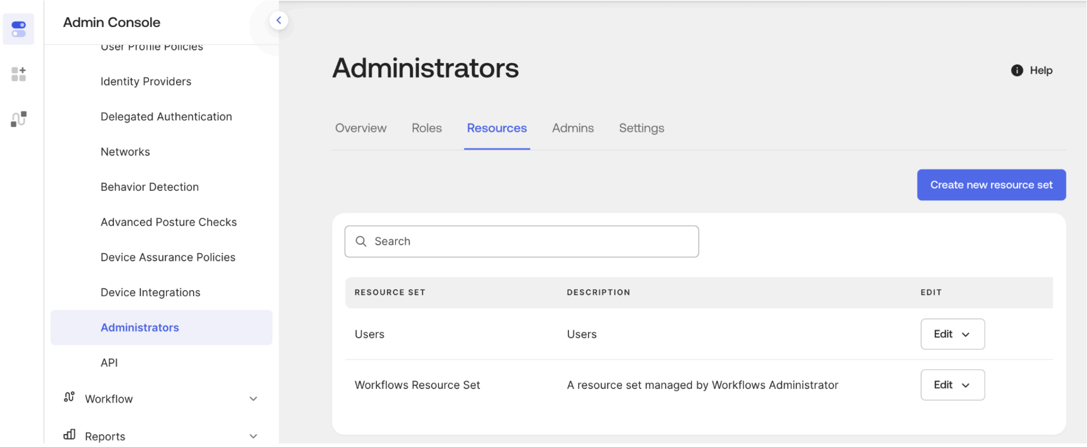
Fig 2: Generate a new key

Step 3: Assign the Right Permissions

This is where we apply the principle of least privilege. Since we only need to read user information and update first and last names, we'll create a custom admin role with just those capabilities.

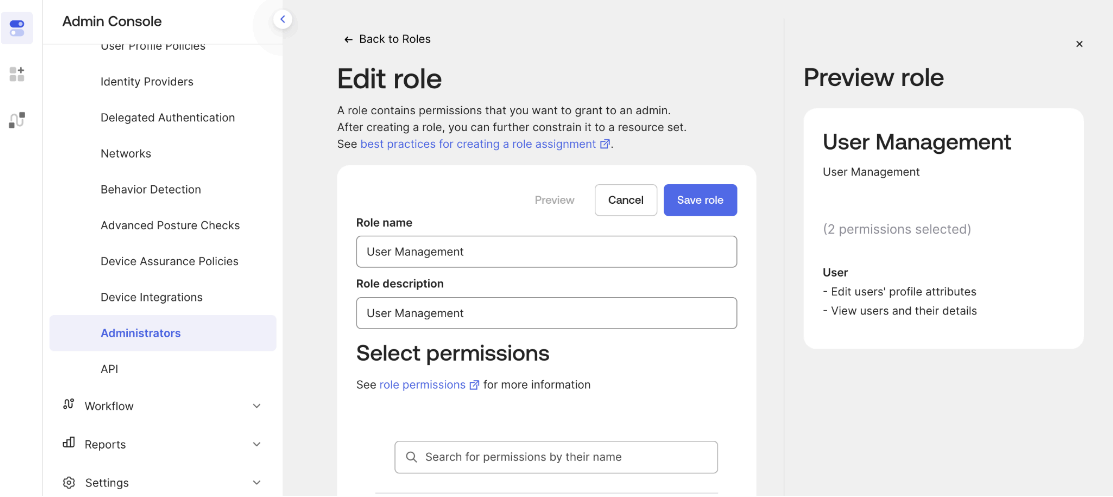
Your custom role needs two specific permissions: "View users and their details" and "Edit users' profile attributes."

To lock this down further, create a Resource Set that includes only the users your app actually needs to access. To assign the role, go to your app's Admin Roles tab, click Edit assignments, and add your custom role. If you're using Resource Sets, you'll find those under Security → Administrators → Resources in the Admin Console.



RESOURCE SET	DESCRIPTION	EDIT
Users	Users	Edit ⌄
Workflows Resource Set	A resource set managed by Workflows Administrator	Edit ⌄

Fig 3: Create a resource set



A role contains permissions that you want to grant to an admin.
After creating a role, you can further constrain it to a resource set.
See best practices for creating a role assignment.

Role name: User Management

Role description: User Management

See role permissions for more information

User Management

User Management

(2 permissions selected)

User
- Edit users' profile attributes
- View users and their details

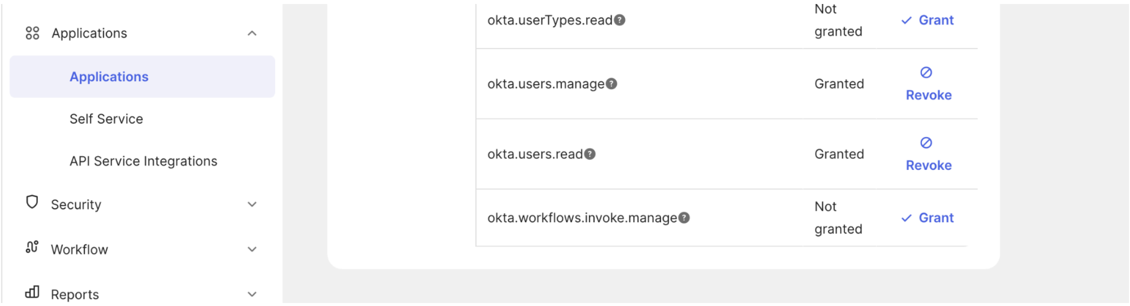
Fig 4: Create a role

The benefit here is clear: your app can't accidentally access or modify users outside its designated scope.

Step 4: Grant API Scopes

Navigate to your app's Okta API Scopes tab and grant these two scopes:

- okta.users.read (lets you retrieve user information)
- okta.users.manage (allows you to update user profiles)



okta.userTypes.read	Not granted	✓ Grant
okta.users.manage	Granted	⊘ Revoke
okta.users.read	Granted	⊘ Revoke
okta.workflows.invoke.manage	Not granted	✓ Grant

Fig 5: Grant API Scopes

Remember, you'll need to explicitly request these scopes when getting your access token. The token will only include scopes that you've both requested and granted to the app.

Step 5: Create Your Client Assertion

This is where you build and sign the JWT that proves your app's identity. Your JWT needs specific claims in both the header and payload.

The header should specify RS256 as the algorithm and include your key ID. In the payload, set both the issuer and subject to your client ID. The audience should be your Okta token endpoint, typically https://your-domain.okta.com/oauth2/v1/token. Set an expiration time between 5 and 10 minutes from now (Okta requires it to be within an hour). Finally, include a unique identifier in the jti claim to prevent replay attacks.

Sign this JWT using your private key with the RS256 algorithm.

Step 6: Request Your Access Token

Now you're ready to get an access token. Make a POST request to your token endpoint at https://your-domain.okta.com/oauth2/v1/token with a content type of application/x-www-form-urlencoded.

Your request body should include:

- grant_type set to client_credentials
- scope listing the scopes you need (okta.users.read okta.users.manage)
- client_assertion_type as urn:ietf:params:oauth:client-assertion-type:jwt-bearer
- client_assertion containing your signed JWT

If everything's configured correctly, you'll receive a response with a Bearer token, the access token itself, an expiration time (usually 3600 seconds), and the scopes that were granted.

Step 7: Make API Calls

With your access token in hand, you can now call the Okta Users API. The base URL is https://your-domain.okta.com/api/v1.

To get a specific user, send a GET request to /users/{id} with your access token in the Authorization header as a Bearer token.

For listing users with pagination, use GET /users?limit=10&after={cursor}, again including your Bearer token.

To update a user's first and last name, POST to /users/{id} with your Bearer token. The request body should be JSON formatted with the profile information nested properly firstName and lastName should be inside a profile object.

Troubleshooting Common Issues

If you get a 403 Forbidden error, your token is likely missing a required scope or your app doesn't have the necessary admin role assigned.

A 400 error with code E0000003 usually means your request body isn't structured correctly. Remember to wrap your profile fields in a profile object.

Getting 401 or 403 errors from the token endpoint typically indicates a mismatch between your key ID and private key, an incorrect audience claim, or the issuer and subject not matching your client ID.

Why Do We Need Both Roles and Scopes?

You might wonder why Okta requires both admin roles and OAuth scopes. The answer is defense in depth.

The admin role controls what your application is authorized to do within your Okta organization. It's about which resources and operations your app can access at the organizational level.

The OAuth scope controls what a specific access token is allowed to do when calling APIs; it's about what operations that particular token can perform.

Both checks must pass for your API call to succeed. This dual-layer approach provides stronger security by ensuring that even if a token is compromised, it can only do what both the role and scope allow.

Preparing for Production

Before you deploy your service app to production, there are several important security and reliability considerations to address.

Key Management

Generate your cryptographic keys through your organization's internal PKI system and establish a regular rotation schedule. Store your private keys in a dedicated secret management system; never hardcode them or commit them to version control. This might seem obvious, but it's worth emphasizing since exposed keys are one of the most common security incidents.

Permission Boundaries

Use Resource Sets to limit the scope of what your app can access. Even if something goes wrong, the damage is contained to only the users or resources you've explicitly included. Always follow the principle of least privilege. If your app only needs to update names, don't permit it to delete accounts.

Scope Your Tokens

Only request the OAuth scopes you actually need. For this use case, that's okta.users.read and okta.users.manage. Requesting fewer scopes reduces your security footprint.

Network Security

Restrict your service's outbound traffic to only communicate with Okta's endpoints. If your infrastructure supports it, use mutual TLS or private connectivity options for an additional layer of security.

Logging Practices

Always redact access tokens from your logs. These are sensitive credentials that should never be written to disk. Do log HTTP status codes and error response bodies from Okta, though. These details are invaluable when troubleshooting issues in production.

Resilience and Reliability

Implement timeouts and retry logic for token requests. If Okta returns a 5xx server error, use exponential backoff with jitter before retrying. Consider adding a circuit breaker pattern to prevent cascading failures when Okta's API experiences issues.

Observability

Add metrics to track token fetch latency, cache hit and miss rates, and the distribution of response codes from Okta's API. Breaking down responses into 2xx, 4xx, and 5xx categories helps you quickly identify whether issues are on your end or Okta's.

Handling Secrets

Ideally, avoid storing secrets on disk entirely. If you must write secrets to the filesystem, set strict file permissions and implement host-level access controls to limit who can read those files.

Understanding DPoP (Demonstrating Proof of Possession)

DPoP is an advanced security feature that binds access tokens to a specific client and prevents stolen tokens from being replayed against different endpoints. Here's what you need to know.

What It Does

When DPoP is enabled, the access token becomes tied to a public key that only your client holds. Each API request must include a proof JWT demonstrating possession of the corresponding private key. This makes stolen tokens essentially useless to attackers.

When You Don't Need It

For most backend services, DPoP adds complexity without meaningful security benefits. If you're running server-to-server communication in a controlled network, using private_key_jwt authentication, working with short-lived tokens, and already securing traffic with TLS or mutual TLS, you're probably fine without DPoP.

When You Should Consider It

DPoP makes sense for public or native clients where tokens might be intercepted, think mobile apps or browser-based applications. It's also valuable when there's a realistic risk of tokens being replayed across different hosts or systems.

How to Enable DPoP

If you need to enable DPoP, here's the general process:

First, you'll need to use a custom authorization server with an access policy rather than the default organization authorization server. In your client app configuration, enable the option for requiring DPoP headers in token requests.

Generate a separate JWK key pair specifically for DPoPl don't reuse your client authentication keys. The first time you request a token with a DPoP header, Okta will respond with an error containing a nonce value. You'll need to use this nonce in your subsequent request.

Build a DPoP proof JWT with the proper claims: set the type to dpop+jwt, include your public key in the header, and add claims for the HTTP method, target URL, timestamp, nonce, and a unique identifier. Submit your token request again with this proof.

Once you receive your DPoP-bound access token, every API call must include two headers: the Authorization header with your DPoP token, and a DPoP header with a fresh proof JWT that matches the specific HTTP method and URL you're calling.

The resource server validates everything: the signature, whether the HTTP method and URL match, that each proof JWT is used only once, and that the token's key thumbprint matches the proof key.

The Trade-offs

DPoP provides strong protection against token replay attacks, but it comes at a cost. Your client code becomes more complex since you need to generate a new proof for every request and handle nonce refresh logic. Resource servers also need additional validation logic.

How This Maps to the Sample Spring Boot Service

The sample application demonstrates these concepts in practice. Configuration comes from environment variables, including your Okta domain, client ID, scopes, JWT audience, key ID, and the path to your private key file.

The TokenClient component handles building the private_key_jwt assertion and exchanging it at Okta's token endpoint. It includes basic in-memory caching to avoid unnecessary token requests.

The UsersClient component makes calls to Okta's Users API for listing and retrieving users, as well as updating first and last names by posting to the user endpoint with the profile data properly nested.

Error handling logs the upstream HTTP status and response body, then throws structured exceptions that your application code can catch and handle appropriately.

Fixing Common 403 Errors

When you encounter a 403 Forbidden error, work through these steps systematically:

First, verify that you've granted the necessary scopes to your app in the Okta console. Then check that your access token actually contains both required scopes in its claims.

Next, confirm that you've assigned the appropriate custom admin role to your service app. Verify that your Resource Set includes the users you're trying to access.

Finally, restart your application to force it to obtain a fresh token with the updated permissions. Old cached tokens won't reflect the permission changes you just made.