

OAuth2/OIDC Provider: Design Document

Overview

Build a complete OAuth2 and OpenID Connect identity provider that enables secure third-party application authorization. This system solves the critical challenge of allowing applications to access user data without exposing user credentials, while maintaining strong security guarantees through cryptographic tokens and standardized flows.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): This provides foundational context for all milestones (1-4)

Mental Model: Digital Valet Key System

Before diving into the technical complexities of OAuth2 and OpenID Connect, let's establish an intuitive mental model that will help us understand the core authorization problem these protocols solve.

Imagine you're visiting a luxury hotel and need to leave your car with the valet. You could hand over your main car key, but that would give the valet complete access to everything: they could drive to Las Vegas, access your glove compartment, adjust your seat settings, and even steal your car. Instead, modern cars provide a **valet key** - a special key that starts the engine and unlocks the doors, but cannot access the trunk or glove compartment, and often has speed limitations programmed into the car's computer.

This valet key system embodies the core principle of OAuth2: **delegated authorization with limited scope**. Just as the valet key grants specific, limited access to your car without compromising your full ownership, OAuth2 allows applications to access specific user data without requiring the user's complete credentials.

Let's extend this analogy to understand the key participants in OAuth2:

- **You (the car owner)** represent the **Resource Owner** - the user who owns the data
- **Your car** represents the **Protected Resource** - the user's data (photos, contacts, emails)
- **The valet service** represents the **Client Application** - the third-party app requesting access
- **The valet key** represents the **Access Token** - a limited-capability credential
- **The hotel's key management system** represents the **Authorization Server** - the OAuth2 provider that issues tokens
- **The car's computer that validates the valet key** represents the **Resource Server** - the API that hosts the protected data

When you authorize the valet service, you're not giving them your master key (password). Instead, the hotel's key management system creates a temporary, limited-capability key specifically for this interaction. The car's computer can verify this key's authenticity and enforce its limitations automatically.

Key Insight: OAuth2 eliminates the need for password sharing by introducing a token-based delegation system where the user grants specific, revocable permissions to applications without exposing their primary credentials.

This mental model helps us understand why building a secure OAuth2 provider is challenging - we're essentially building the entire "hotel key management system" that can issue various types of limited-capability keys, track their usage, enforce their restrictions, and revoke them when necessary.

Existing Authentication Approaches

To appreciate why OAuth2 represents a significant improvement in application security, let's examine the authentication approaches that preceded it and understand their fundamental limitations.

Password Sharing Approach

The most primitive approach to third-party application authorization involves users sharing their actual username and password with the requesting application. While simple to implement, this approach creates severe security and usability problems.

Aspect	Password Sharing Approach	Problems
User Experience	User enters their Facebook/Google password directly into third-party app	Users must trust every app with their most sensitive credentials
Access Control	App gains complete access to user's account	No way to limit what the app can do - it can read, modify, or delete anything
Credential Management	App must store user's actual password (often in plaintext)	Creates honey pot targets for attackers; password breaches affect all apps
Revocation	User must change their password to revoke access	Breaks access for ALL apps and devices, not just the compromised one
Auditability	No distinction between user actions and app actions	Impossible to track which actions came from which app
Password Rotation	User changing password breaks all connected apps	Forces users to avoid password rotation, weakening overall security

Consider a concrete scenario: Sarah wants to use a photo printing service that can access her Google Photos. Under the password sharing model, she would enter her Google username and password into PhotoPrint's website. Now PhotoPrint can read her Gmail, modify her Google Drive files, change her account settings, and even lock her out by changing her password. If PhotoPrint's database gets breached, Sarah's Google credentials are compromised, forcing her to change her password and reconfigure every device and application that accesses her Google account.

API Key Approach

Recognizing the problems with password sharing, many services introduced API keys - long-lived tokens that applications could use instead of passwords. While an improvement, this approach still has significant limitations.

Aspect	API Key Approach	Limitations
Scope Control	API key typically grants broad access to user's account	Limited ability to restrict access to specific resources or actions
User Involvement	User must manually generate and copy-paste API keys	Complex setup process; users often confused about key generation
Expiration	Keys often never expire or have very long lifespans	Compromised keys remain valid indefinitely; no automatic rotation
Granular Permissions	Single key for all operations	Cannot grant read-only access or limit to specific data types
Revocation	User must manually revoke keys through provider's interface	No standardized way for apps to check if key is still valid
Multi-User Apps	Each user must generate separate API keys	Apps must manage complex key-to-user mapping; no unified flow

The API key approach works reasonably well for developer-to-service integration (like accessing a weather API), but breaks down for consumer applications that need to access data from multiple users across different services.

OAuth2 Approach

OAuth2 fundamentally changes the authorization model by introducing a standardized delegation protocol that addresses the limitations of both password sharing and API keys.

Aspect	OAuth2 Approach	Benefits
Credential Security	User never shares password with third-party apps	Eliminates password exposure; apps receive opaque tokens instead
Granular Permissions	Apps request specific scopes (read photos, send emails)	Users can approve limited access; principle of least privilege
Time-Limited Access	Access tokens have short expiration times (15 minutes - 1 hour)	Reduces blast radius of compromised tokens
Standardized Flow	Consistent authorization experience across all providers	Users learn one flow; developers implement one protocol
Revocation	Users can revoke access per-application without affecting others	Fine-grained access control; easy to remove misbehaving apps
Refresh Capability	Refresh tokens allow apps to get new access tokens	Long-running apps don't need user re-authorization
Auditability	Clear separation between user actions and app actions	Providers can track which app performed which operations

The OAuth2 flow works like this: when PhotoPrint wants to access Sarah's Google Photos, it redirects her to Google's authorization server. Google authenticates Sarah using her normal credentials (which PhotoPrint never sees), shows her exactly what PhotoPrint is requesting access to, and asks for her approval. If she approves, Google issues PhotoPrint a

limited-capability access token that can only read photos from her account. This token expires after an hour, and Sarah can revoke it at any time through her Google account settings.

Design Insight: OAuth2's key innovation is the introduction of an intermediary (the authorization server) that translates user consent into limited-capability tokens, enabling secure delegation without credential sharing.

Core Security Challenges

Building a secure OAuth2 provider involves solving several complex security challenges that don't exist in simpler authentication systems. Understanding these challenges is crucial for implementing a system that can withstand real-world attacks.

Cryptographic Token Management

OAuth2 providers must generate, sign, validate, and revoke millions of tokens while maintaining cryptographic security guarantees. This presents several specific challenges:

Secure Random Number Generation: Authorization codes, client secrets, and tokens must be generated using cryptographically secure random number generators. Weak randomness can enable attackers to predict future tokens or brute-force existing ones. The provider must ensure sufficient entropy is available and that the random number generator is properly seeded.

JWT Signing and Verification: Access tokens implemented as JWTs require asymmetric cryptography for signing and verification. The provider must securely manage private keys used for signing, ensure proper key rotation, and handle signature verification edge cases. Additionally, the choice of signing algorithm (RS256, ES256, etc.) affects both security and performance characteristics.

Token Binding and Replay Protection: Tokens must be cryptographically bound to specific clients and contexts to prevent replay attacks. This involves validating that the entity presenting a token is the same entity it was issued to, which becomes complex in distributed systems with multiple resource servers.

Security Property	Challenge	OAuth2 Solution
Confidentiality	Tokens contain sensitive authorization information	Opaque tokens for authorization codes; signed JWTs for access tokens
Integrity	Prevent token modification by attackers	Cryptographic signatures; constant-time comparison for validation
Authenticity	Verify tokens were issued by legitimate authorization server	Asymmetric signatures; well-known public key endpoints
Non-repudiation	Prove which entity performed which action	Token-to-client binding; audit logs with token identifiers
Freshness	Prevent replay of old tokens	Short expiration times; timestamp validation; nonce handling

Distributed Security State Management

Unlike simple session-based authentication where state exists on a single server, OAuth2 providers must maintain security-critical state across distributed systems and time periods.

Authorization Code State: Authorization codes must be single-use, short-lived, and bound to specific clients and redirect URLs. The provider must track which codes have been used and prevent replay attacks, while handling the race conditions that can occur when multiple systems attempt to exchange the same code simultaneously.

Token Family Tracking: Refresh token rotation requires tracking "token families" - chains of related refresh tokens issued for the same authorization grant. If an old refresh token is reused (indicating possible token theft), the entire token family must be revoked. This requires maintaining historical token relationships across potentially millions of active sessions.

Consent and Scope Management: The provider must remember which users have granted which permissions to which clients, enabling features like "skip consent screen for previously authorized scopes" while ensuring scope creep doesn't occur when apps request additional permissions.

State Management Challenge	Complexity	Failure Impact
Cross-Server Consistency	Multiple servers must see token revocations immediately	Revoked tokens remain valid on some servers; security breach
Partial System Failures	Network partitions, database failures, cache inconsistencies	Users locked out; tokens valid longer than intended
Race Conditions	Concurrent token exchanges, refresh operations, revocations	Double-spending of authorization codes; token family corruption
State Explosion	Millions of tokens, codes, and consent records	Memory exhaustion; slow lookups; data consistency challenges

Protocol-Level Attack Resistance

OAuth2 and OpenID Connect specifications address numerous attack vectors, but implementing these defenses correctly requires deep understanding of web security principles.

PKCE (Proof Key for Code Exchange): Modern OAuth2 implementations must support PKCE to protect against authorization code interception attacks, particularly on mobile devices and single-page applications. This requires generating cryptographically secure code challenges, properly validating code verifiers using SHA256 hashing, and enforcing PKCE for public clients.

Redirect URI Validation: One of the most critical security controls in OAuth2 is strict validation of redirect URLs. The provider must perform exact string matching against pre-registered URLs, resist various bypass attempts (subdomain attacks, open redirects, homograph attacks), and handle edge cases like custom URL schemes for mobile apps.

State Parameter Handling: The state parameter prevents CSRF attacks by allowing clients to verify that authorization responses correspond to their requests. The provider must preserve state values across the authorization flow and ensure they are validated by clients.

Security Principle: OAuth2 security depends on the weakest link in a complex chain of validations. A single bypass in redirect URI validation, PKCE verification, or token binding can compromise the entire system.

Real-World Implementation Pitfalls

The OAuth2 and OIDC specifications leave many security-critical implementation details to the provider, creating opportunities for subtle vulnerabilities.

Timing Attack Vulnerabilities: Token validation, client secret verification, and authorization code lookup must use constant-time comparison algorithms to prevent timing-based attacks where attackers can determine valid prefixes of secrets by measuring response times.

Token Leakage Through Logs: Access tokens and refresh tokens must never appear in application logs, error messages, or debugging output. This requires careful sanitization of log entries and error responses while maintaining sufficient debugging information for operational support.

Cryptographic Agility: The provider must support algorithm evolution (moving from RS256 to ES256, increasing key sizes) without breaking existing clients. This requires versioned key management and graceful algorithm migration strategies.

Attack Vector	Technical Challenge	Mitigation Complexity
Authorization Code Injection	Attacker intercepts codes via network, malware, or device sharing	PKCE implementation; short code lifetimes; TLS enforcement
Token Sidejacking	Tokens stolen via XSS, network interception, or server logs	Secure token storage; SameSite cookies; log sanitization
Redirect URI Manipulation	Bypass redirect validation to send codes to attacker domains	Exact string matching; homograph detection; scheme validation
Scope Creep	Apps gradually request more permissions than originally granted	Consent re-prompting; scope diff detection; audit logging
Client Impersonation	Malicious apps pretend to be legitimate clients	Client authentication; app attestation; dynamic registration controls

The complexity of these security challenges explains why many organizations choose to use hosted OAuth2 providers (like Auth0, Okta, or cloud provider services) rather than building their own. However, understanding these challenges is essential for developers who need to implement custom authorization logic, integrate with existing systems, or meet specific compliance requirements that hosted solutions cannot address.

Design Philosophy: A secure OAuth2 provider must assume that every component can be compromised and design defenses that limit the blast radius of any single point of failure. This defense-in-depth approach requires multiple layers of validation, cryptographic protection, and audit logging.

Implementation Guidance

Technology Recommendations

Building a production-ready OAuth2 provider requires careful selection of cryptographic libraries, storage systems, and web frameworks that provide the security primitives needed for token management.

Component	Simple Option	Advanced Option
Web Framework	<code>net/http</code> with <code>gorilla/mux</code> for routing	<code>gin-gonic/gin</code> or <code>echo</code> for middleware support
Cryptography	Go's <code>crypto</code> package for JWT signing	<code>golang-jwt/jwt/v5</code> for JWT convenience methods
Database	SQLite with <code>database/sql</code> for development	PostgreSQL with <code>lib/pq</code> for production
Caching	In-memory <code>sync.Map</code> for token blacklist	Redis with <code>go-redis/redis/v8</code> for distributed cache
Secure Random	<code>crypto/rand</code> for all token generation	Hardware security module (HSM) integration
Password Hashing	<code>golang.org/x/crypto/bcrypt</code> for client secrets	<code>golang.org/x/crypto/argon2</code> for stronger protection
Configuration	Environment variables with <code>os.Getenv</code>	<code>viper</code> for structured configuration management
Logging	Standard <code>log</code> package with careful sanitization	<code>logrus</code> or <code>zap</code> for structured logging

Recommended File Structure

Organize your OAuth2 provider codebase to separate concerns clearly and make testing straightforward. This structure scales from prototype to production deployment.

```

oauth-provider/
├── cmd/
│   └── server/
│       └── main.go           ← Application entry point and server startup
├── internal/
│   ├── auth/
│   │   ├── authorization.go    ← Core authorization logic (Milestones 1-4)
│   │   ├── token.go            ← Authorization endpoint and consent flow
│   │   ├── introspection.go    ← Token endpoint and JWT generation
│   │   └── userinfo.go         ← Token introspection and revocation
│   ├── client/
│   │   ├── client.go          ← OAuth client management (Milestone 1)
│   │   └── storage.go          ← Client registration and validation
│   ├── crypto/
│   │   ├── jwt.go              ← Client credential storage
│   │   ├── random.go           ← Cryptographic operations
│   │   └── pkce.go              ← Secure random number generation
│   ├── storage/
│   │   ├── models.go           ← PKCE challenge/verifier handling
│   │   ├── tokens.go            ← Data persistence layer
│   │   └── consent.go          ← Data structure definitions
│   └── web/
│       ├── handlers.go         ← Token storage and retrieval
│       ├── middleware.go        ← User consent persistence
│       └── errors.go            ← HTTP handlers and middleware
└── pkg/
    └── validator/
        └── validator.go          ← OAuth endpoint implementations
                                    ← Authentication and rate limiting
                                    ← OAuth2 error response formatting
                                    ← JWT validation library for resource servers
├── web/
│   ├── templates/             ← HTML templates for consent screens
│   └── static/                ← CSS and JavaScript assets
├── configs/
└── docs/
    └── api.md                 ← Configuration file template
                                ← API documentation

```

Infrastructure Starter Code

The following components provide the foundation for your OAuth2 provider but aren't the core learning objectives. These implementations handle essential functionality so you can focus on the authorization logic.

Secure Random Number Generator (`internal/crypto/random.go`):

```
package crypto
```

```
import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
)
```

```
// TokenGenerator provides cryptographically secure random token generation
```

```
type TokenGenerator struct{}
```

```
// GenerateAuthorizationCode creates a 32-byte random authorization code
```

```
// encoded as URL-safe base64 (43 characters without padding)
```

```
func (tg *TokenGenerator) GenerateAuthorizationCode() (string, error) {
```

```
    // TODO 1: Create 32-byte buffer for random data
```

```
    // TODO 2: Fill buffer with crypto/rand.Read()
```

```
    // TODO 3: Encode as URL-safe base64 without padding
```

```
    // TODO 4: Return error if random generation fails
```

```
}
```

```
// GenerateClientSecret creates a 32-byte random client secret
```

```
// Returns both the raw secret and its bcrypt hash for storage
```

```
func (tg *TokenGenerator) GenerateClientSecret() (secret, hash string, err error) {
```

```
    // TODO 1: Generate 32 random bytes for the secret
```

```
    // TODO 2: Encode secret as base64 for client use
```

```
    // TODO 3: Hash secret with bcrypt cost 12
```

```
    // TODO 4: Return both plaintext secret (for client) and hash (for storage)
```

```
}
```

```
// GenerateState creates a 16-byte random state parameter for CSRF protection
```

```
func (tg *TokenGenerator) GenerateState() (string, error) {
```

```
    // TODO 1: Generate 16 random bytes
```

GO

```
// TODO 2: Encode as hex string  
  
// TODO 3: Return error if random generation fails  
}
```

OAuth2 Error Response Formatter (`internal/web/errors.go`):

GO

```
package web

import (
    "encoding/json"
    "net/http"
)

// OAuth2Error represents a standardized OAuth2 error response

type OAuth2Error struct {

    Error          string `json:"error"`
    ErrorDescription string `json:"error_description,omitempty"`
    ErrorURI       string `json:"error_uri,omitempty"`
    State          string `json:"state,omitempty"`
}

// WriteOAuth2Error sends a standardized OAuth2 error response

func WriteOAuth2Error(w http.ResponseWriter, statusCode int, errorCode, description string) {
    w.Header().Set("Content-Type", "application/json")
    w.Header().Set("Cache-Control", "no-store")
    w.Header().Set("Pragma", "no-cache")
    w.WriteHeader(statusCode)

    response := OAuth2Error{
        Error:          errorCode,
        ErrorDescription: description,
    }

    json.NewEncoder(w).Encode(response)
}

// Common OAuth2 error codes as constants
const (
```

```
    ErrorInvalidRequest      = "invalid_request"
    ErrorInvalidClient       = "invalid_client"
    ErrorInvalidGrant        = "invalid_grant"
    ErrorUnauthorizedClient = "unauthorized_client"
    ErrorUnsupportedGrantType= "unsupported_grant_type"
    ErrorInvalidScope         = "invalid_scope"
    ErrorAccessDenied         = "access_denied"
    ErrorUnsupportedResponseType= "unsupported_response_type"
    ErrorServerError          = "server_error"
    ErrorTemporarilyUnavailable= "temporarily_unavailable"
)

```

Basic HTTP Middleware (`internal/web/middleware.go`):

```
package web

import (
    "context"
    "net/http"
    "time"
)

// SecurityHeaders adds security-related HTTP headers to all responses

func SecurityHeaders(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("X-Content-Type-Options", "nosniff")
        w.Header().Set("X-Frame-Options", "DENY")
        w.Header().Set("X-XSS-Protection", "1; mode=block")
        w.Header().Set("Referrer-Policy", "strict-origin-when-cross-origin")
        next.ServeHTTP(w, r)
    })
}

// RequestTimeout enforces a maximum request processing time

func RequestTimeout(timeout time.Duration) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            ctx, cancel := context.WithTimeout(r.Context(), timeout)
            defer cancel()
            next.ServeHTTP(w, r.WithContext(ctx))
        })
    }
}
```

GO

Core Logic Skeleton Code

The following interfaces and function signatures define the core OAuth2 authorization logic you'll implement across the four milestones. Each function maps to specific algorithm steps described in later sections.

Client Registration Interface (`internal/client/client.go`):

```
package client

import "time"

// Client represents a registered OAuth2 application

type Client struct {

    ClientID      string      `json:"client_id"`

    ClientSecret  string      `json:"-"`          // Never serialize to JSON

    SecretHash    string      `json:"-"`          // Bcrypt hash for storage

    Name          string      `json:"name"`

    RedirectURIs []string    `json:"redirect_uris"`

    GrantTypes    []string    `json:"grant_types"`

    Scopes        []string    `json:"scopes"`

    CreatedAt     time.Time  `json:"created_at"`

}

// ClientStore handles OAuth client persistence and retrieval

type ClientStore interface {

    CreateClient(client *Client) error

    GetClient(clientID string) (*Client, error)

    ValidateClientCredentials(clientID, clientSecret string) (*Client, error)

}

// RegisterClient creates a new OAuth2 client with generated credentials

func RegisterClient(store ClientStore, name string, redirectURIs []string) (*Client, error) {

    // TODO 1: Generate cryptographically random client_id (16 bytes, base64)

    // TODO 2: Generate client_secret and its bcrypt hash

    // TODO 3: Validate that all redirect URIs are absolute URLs with https scheme

    // TODO 4: Create Client struct with generated values and current timestamp

    // TODO 5: Store client in persistent storage via ClientStore

    // TODO 6: Return client with plaintext secret for one-time display to developer

}
```

GO

```
}
```

Authorization Code Flow (`internal/auth/authorization.go`):

```
package auth

import "time"

// AuthorizationRequest represents the parsed authorization request parameters

type AuthorizationRequest struct {

    ClientID          string
    RedirectURI       string
    Scope             string
    State             string
    ResponseType      string
    CodeChallenge     string
    CodeChallengeMethod string
}

// AuthorizationCode represents an issued authorization code

type AuthorizationCode struct {

    Code          string `json:"code"`
    ClientID     string `json:"client_id"`
    UserID        string `json:"user_id"`
    RedirectURI   string `json:"redirect_uri"`
    Scope         string `json:"scope"`
    CodeChallenge string `json:"code_challenge,omitempty"`
    ExpiresAt     time.Time `json:"expires_at"`
    Used          bool `json:"used"`
}

// HandleAuthorizationRequest processes the initial OAuth2 authorization request

func HandleAuthorizationRequest(req *AuthorizationRequest, clientStore ClientStore) (*AuthorizationCode, error) {
    // TODO 1: Validate that response_type is "code"
    // TODO 2: Look up client by client_id and verify it exists
}
```

GO

```

    // TODO 3: Validate redirect_uri exactly matches one of client's registered URIs

    // TODO 4: Validate that requested scopes are subset of client's allowed scopes

    // TODO 5: If PKCE code_challenge present, validate it's base64url encoded and appropriate length

    // TODO 6: Generate cryptographically random authorization code

    // TODO 7: Create AuthorizationCode struct with 10-minute expiration

    // TODO 8: Store authorization code for later exchange

    // Hint: This function handles the authorization request but doesn't capture user consent yet

}

```

Language-Specific Security Hints

Constant-Time Comparison: Always use `subtle.ConstantTimeCompare()` for validating tokens, client secrets, and authorization codes to prevent timing attacks:

```

import "crypto/subtle" GO

// WRONG - vulnerable to timing attacks

if token == storedToken {

    return true

}

// CORRECT - constant time comparison

if subtle.ConstantTimeCompare([]byte(token), []byte(storedToken)) == 1 {

    return true

}

```

Secure Random Generation: Use `crypto/rand.Read()` for all security-critical random number generation. Never use `math/rand` for tokens, codes, or secrets:

```
import "crypto/rand" GO

// Generate 32 bytes of cryptographically secure random data

randomBytes := make([]byte, 32)

if _, err := rand.Read(randomBytes); err != nil {

    return fmt.Errorf("failed to generate random bytes: %w", err)
}
```

JWT Key Management: Store JWT signing keys securely and rotate them periodically. Use RS256 or ES256 algorithms:

```
import "crypto/rsa" GO

// Generate 2048-bit RSA key for JWT signing

privateKey, err := rsa.GenerateKey(rand.Reader, 2048)

if err != nil {

    return fmt.Errorf("failed to generate RSA key: %w", err)
}

// Store private key securely (encrypted at rest)

// Expose public key via /.well-known/jwks.json endpoint
```

Database Connection Security: Use connection pooling and prepared statements to prevent SQL injection:

```
// Prepared statement prevents SQL injection GO

stmt, err := db.Prepare("SELECT client_id, secret_hash FROM clients WHERE client_id = ?")

if err != nil {

    return err
}

defer stmt.Close()

var clientID, secretHash string

err = stmt.QueryRow(providedClientID).Scan(&clientID, &secretHash)
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Authorization codes always invalid	Code not being stored or wrong lookup key	Check database after code generation	Verify storage key matches lookup key exactly
JWT tokens fail validation	Clock skew or wrong signing key	Compare token iat/exp with current time	Add clock skew tolerance (30 seconds)
PKCE verification fails	Code verifier/challenge mismatch	Log both verifier and stored challenge	Ensure base64url encoding without padding
Redirect URI validation fails	Exact match requirement not met	Log provided URI vs registered URI	Use exact string comparison, not pattern matching
Client authentication fails	bcrypt comparison wrong	Test hash generation separately	Use bcrypt.CompareHashAndPassword() correctly

Goals and Non-Goals

Milestone(s): This section establishes the scope for all milestones (1-4), defining success criteria and boundaries

This section establishes clear boundaries for what our OAuth2 and OpenID Connect provider will accomplish. Think of this as signing a contract with stakeholders - we're committing to specific capabilities while explicitly declining to solve certain problems. This clarity prevents scope creep and ensures we build a focused, secure system rather than an over-engineered monster that tries to solve every possible identity use case.

The goals are organized into three categories: functional requirements that define what the system does, security requirements that define how it protects users and data, and explicit non-goals that define what we will not build. Each goal maps to specific acceptance criteria in our milestones, creating a clear path from high-level objectives to concrete implementation tasks.

Functional Goals: Required OAuth2/OIDC Features and Flows

Our OAuth2 provider must implement the core authorization flows and endpoints required for modern web and mobile applications. These functional goals represent the minimum viable feature set for a production-ready identity provider that third-party applications can rely on for user authorization.

Core OAuth2 Authorization Flows

The system must support the **authorization code grant flow** as the primary mechanism for web applications to obtain user consent and access tokens. This flow involves redirecting users to our authorization server, displaying a consent screen, generating a short-lived authorization code, and exchanging that code for tokens. This is the most secure flow for applications that can protect a client secret.

We must implement **Proof Key for Code Exchange (PKCE)** as a mandatory security extension for all authorization requests. PKCE prevents authorization code interception attacks by requiring clients to prove they initiated the original

authorization request. Public clients like mobile apps and single-page applications cannot protect secrets, making PKCE their primary defense against token theft.

The **client credentials grant flow** enables machine-to-machine authentication where no user interaction is required. Background services, APIs calling other APIs, and automated systems use this flow to authenticate using only their client ID and secret. This flow bypasses user consent since the client owns the data it's accessing.

Token Management and Lifecycle

Our token endpoint must generate **JWT access tokens** with proper claims, signatures, and expiration times. JWTs enable resource servers to validate tokens locally without calling back to our authorization server, reducing latency and improving scalability. Each JWT must contain standard claims like subject (user ID), issuer (our server), audience (intended recipient), expiration time, and granted scopes.

Refresh token rotation provides long-term access while maintaining security through token family tracking. When a client uses a refresh token to obtain a new access token, we must issue a new refresh token and invalidate the old one. If an old refresh token is reused (indicating potential theft), we revoke the entire token family associated with that user and client.

The system must implement **token introspection** per RFC 7662, allowing resource servers to check token validity and retrieve metadata. This endpoint accepts an access or refresh token and returns whether it's active, along with information like expiration time, scopes, and client ID. Protected by client authentication, this endpoint serves as the authoritative source for token status.

Token revocation per RFC 7009 enables users and clients to immediately invalidate tokens. When a user logs out, uninstalls an app, or revokes consent, the associated tokens must become unusable instantly. This prevents lingering access after authorization is withdrawn.

Client Management and Registration

The system must provide **client registration APIs** that create OAuth2 client records with generated credentials and configured parameters. Each client receives a unique client ID, a securely hashed client secret (for confidential clients), and a list of allowed redirect URIs where users can be sent after authorization.

Client authentication must validate client credentials during token requests using HTTP Basic authentication or form parameters. We must support both confidential clients (web applications with secrets) and public clients (mobile apps and SPAs without secrets) with appropriate security measures for each type.

Redirect URI validation prevents authorization code theft by ensuring authorization responses only go to pre-registered, exact-match URIs. Partial matches, subdomain wildcards, and dynamic redirect URIs are explicitly rejected to prevent malicious clients from intercepting authorization codes.

OpenID Connect Identity Layer

As an **OpenID Connect provider**, we must implement the identity layer on top of OAuth2. This includes issuing ID tokens (JWTs containing user authentication information), providing a UserInfo endpoint for accessing user profile claims, and supporting OIDC discovery through well-known configuration endpoints.

The **Userinfo endpoint** returns user profile information based on the scopes granted during authorization. If a client receives consent for the "email" scope, the UserInfo endpoint returns the user's email address. If granted "profile" scope, it returns name and other profile fields. This scope-to-claims mapping ensures clients only access data they've been authorized to see.

OIDC discovery through the `/well-known/openid-configuration` endpoint enables client applications to automatically configure themselves by retrieving our server's capabilities, endpoints, and cryptographic keys. This reduces manual configuration and ensures clients stay synchronized with our server's capabilities.

User Consent and Authorization

The system must present **user consent screens** that clearly display what data and permissions an application is requesting. Users must be able to approve or deny each authorization request with full understanding of the implications. The consent screen shows the requesting application's name, the specific permissions (scopes) requested, and allows granular approval or complete denial.

Consent persistence remembers user authorization decisions so returning users aren't repeatedly prompted for the same client and scopes. If Alice previously granted the "Calendar App" access to her email and calendar, subsequent authorizations for the same permissions proceed automatically without re-prompting.

Consent revocation enables users to withdraw previously granted permissions through a management interface. When consent is revoked, associated tokens must be invalidated, and future authorization requests must re-prompt for user approval.

Functional Goal	Acceptance Criteria	Associated Milestone
Authorization Code Grant	Redirect user to consent, generate codes, exchange for tokens	Milestone 1
PKCE Implementation	Validate code challenges, enforce for public clients	Milestone 1
Client Credentials Grant	Authenticate clients, issue tokens without user consent	Milestone 2
JWT Access Tokens	Generate signed JWTs with proper claims and expiration	Milestone 2
Refresh Token Rotation	Issue new refresh tokens, invalidate old ones, track families	Milestone 2
Token Introspection	RFC 7662 compliant endpoint returning token metadata	Milestone 3
Token Revocation	RFC 7009 compliant endpoint invalidating tokens immediately	Milestone 3
UserInfo Endpoint	Return user claims based on granted scopes	Milestone 4
Consent Management	Display consent screens, persist decisions, enable revocation	Milestone 4
OIDC Discovery	Publish well-known configuration with endpoints and keys	Milestone 4

Security Goals: Security Properties the System Must Maintain

Security represents the most critical aspect of an identity provider - any compromise could expose user data across multiple applications. Our security goals establish the non-negotiable properties that the system must maintain under all circumstances, including attack scenarios and operational failures.

Cryptographic Security Properties

All **client secrets must be stored using cryptographically strong hashing** with per-secret salts. We will never store plaintext secrets that could be compromised in a data breach. Client secret verification must use constant-time comparison to prevent timing attacks that could leak information about secret values.

Authorization codes must be cryptographically random with sufficient entropy to resist brute force attacks. Codes must be single-use only and expire within 10 minutes to minimize the attack window. Each code must be cryptographically bound to the specific client and redirect URI to prevent theft and misuse.

JWT access tokens must be signed using asymmetric cryptography (RS256 or ES256) to enable distributed verification by resource servers. Private keys must be securely stored and never exposed through APIs or logs. Token signatures prevent tampering and ensure authenticity without requiring resource servers to contact our authorization server.

Refresh tokens must have high entropy and be unpredictable to resist guessing attacks. Unlike JWTs, refresh tokens should be opaque strings stored in our database with associated metadata like expiration time, client ID, and user ID.

Attack Resistance Requirements

The system must resist **authorization code interception attacks** through mandatory PKCE implementation. Even if an attacker intercepts an authorization code through app link hijacking or network sniffing, they cannot exchange it for tokens without the original code verifier that only the legitimate client possesses.

Cross-Site Request Forgery (CSRF) protection must be enforced through mandatory state parameters in authorization requests. Clients must provide unpredictable state values that are validated when users return from the authorization server. This prevents attackers from tricking users into authorizing malicious applications.

Token replay attack prevention requires tracking used authorization codes and implementing refresh token rotation. Once an authorization code is exchanged for tokens, it becomes permanently invalid. Similarly, refresh tokens become invalid after a single use, requiring clients to use the newly issued refresh token for subsequent refreshes.

Timing attack resistance mandates constant-time comparison for all secret validation operations. Whether comparing client secrets, authorization codes, or any other sensitive values, the comparison time must not leak information about correct vs. incorrect values.

Data Protection Requirements

Sensitive data must never appear in JWT payloads since JWTs are base64-encoded, not encrypted. Access tokens should contain only non-sensitive claims like user ID, expiration time, and granted scopes. Detailed user information like email addresses, phone numbers, or profile data must only be available through the protected UserInfo endpoint.

Audit logging must record all security-relevant events including client registrations, authorization grants, token issuances, token revocations, and authentication failures. Logs must include sufficient context for forensic analysis while avoiding sensitive data like tokens or user passwords.

Secure credential storage requires that all persistent sensitive data uses appropriate encryption or hashing. Client secrets use bcrypt or similar password hashing. Refresh tokens should be stored as hashed values to prevent compromise even if the database is breached.

Session and State Management Security

Authorization code binding ensures codes cannot be used by unintended parties. Each code must be cryptographically bound to the specific client ID, redirect URI, user ID, and PKCE code challenge that were present during authorization. Attempts to use codes with different parameters must fail validation.

Token family tracking for refresh tokens enables breach detection. When a client exchanges a refresh token, we track the old and new tokens as part of the same family. If the old token is later used (indicating theft), we immediately revoke all tokens in that family to limit the blast radius.

Concurrent session limits prevent token proliferation by limiting how many active refresh tokens a single user can have with each client. When limits are exceeded, the oldest tokens are automatically revoked, ensuring users maintain control over their authorization grants.

Security Property	Implementation Requirement	Attack Prevention
Client Secret Security	bcrypt hashing, constant-time comparison	Database breach, timing attacks
Authorization Code Security	Cryptographic randomness, single-use, 10-minute expiry	Code interception, replay
JWT Signature Security	Asymmetric signing (RS256/ES256), secure key storage	Token tampering, forgery
PKCE Enforcement	Mandatory for all flows, SHA256 code challenge	Code interception
CSRF Protection	Mandatory state parameter validation	Cross-site request forgery
Token Replay Prevention	Single-use codes, refresh token rotation	Replay attacks
Timing Attack Resistance	Constant-time secret comparison	Information leakage
Data Exposure Prevention	No sensitive data in JWTs, protected UserInfo endpoint	Unintended data disclosure
Audit Trail	Security event logging with context	Forensic analysis support
Token Family Tracking	Refresh token breach detection and revocation	Token theft mitigation

Critical Security Insight: The security of an OAuth2 provider extends beyond just protecting its own data - any compromise could expose user data across every application that trusts our tokens. This responsibility amplifies the importance of defense-in-depth strategies and assumes-breach security models.

Non-Goals: Features explicitly excluded from this implementation

Clearly defining what we will not build prevents scope creep and manages stakeholder expectations. These non-goals represent valuable features that could be added in future iterations but are explicitly excluded from our initial implementation to maintain focus on core security and functionality.

Advanced OAuth2 Grant Types and Flows

We will **not implement the implicit grant flow** which directly returns access tokens in the URL fragment without an authorization code exchange step. While historically used for single-page applications, the implicit flow has known security vulnerabilities and has been superseded by the authorization code flow with PKCE for all client types.

Device authorization grant (RFC 8628) for input-constrained devices like smart TVs and IoT devices is excluded. This flow involves displaying codes on one device while users complete authorization on a separate device with better input capabilities. While useful for modern IoT scenarios, it adds complexity without being essential for web and mobile applications.

Resource Owner Password Credentials grant where users directly provide their username and password to client applications is explicitly rejected. This grant type undermines the fundamental OAuth2 principle of never sharing user credentials with third parties and should only be used when no redirect-based flow is possible.

Advanced Security and Enterprise Features

Pushed Authorization Requests (PAR) from RFC 9126, which allows clients to push authorization request parameters directly to the authorization server instead of passing them through browser redirects, is not included. While PAR improves security by keeping request parameters confidential, it adds complexity and is not required for basic OAuth2 compliance.

Rich Authorization Requests that support fine-grained permissions beyond simple scopes (e.g., "access calendar events from January to March for work calendar only") are excluded. The standard scope-based authorization model provides sufficient granularity for most applications.

Financial-grade API (FAPI) compliance with its additional security requirements like mutual TLS authentication, signed request objects, and enhanced token binding is not included. FAPI represents the highest security tier for financial applications but exceeds the requirements for general-purpose identity providers.

Multi-tenant architecture supporting multiple isolated organizations within a single deployment is excluded. Each deployment will serve a single organization or user base, simplifying data isolation and access control models.

Identity Federation and Protocol Integration

SAML 2.0 integration for enterprise single sign-on scenarios is not included. While SAML remains important in enterprise environments, supporting it would require implementing an entirely different protocol stack alongside OAuth2/OIDC.

Social login providers like "Sign in with Google" or "Login with Facebook" are excluded from the core implementation. While valuable for user experience, integrating external identity providers adds operational dependencies and complexity that can be addressed through separate modules.

LDAP or Active Directory integration for enterprise user stores is not included in the initial scope. The system will maintain its own user database, though the architecture should allow for pluggable user stores in future versions.

Cross-domain identity federation protocols like SAML federation or OpenID Connect federation are excluded. Our provider will serve as an authoritative identity source rather than participating in federated trust relationships with other providers.

Advanced User Experience and Management Features

Multi-factor authentication (MFA) during the authorization flow is excluded from core functionality. Users may have MFA configured in their user accounts, but the OAuth2 authorization flow itself will not enforce or manage additional authentication factors.

Progressive consent where applications can request additional scopes after initial authorization without full re-consent is not implemented. Each authorization request is treated independently, requiring explicit user consent for any new or changed scope requests.

Administrative interfaces for managing clients, viewing user consents, monitoring token usage, or configuring server settings are excluded. Initial deployments will rely on direct database access or API calls for administrative tasks.

User self-service portals for managing authorized applications, reviewing permissions, or configuring account settings are not included. Consent revocation will be available through API endpoints but not through a web interface.

Operational and Deployment Features

High availability clustering with leader election, state synchronization, and automatic failover is excluded. Initial deployments assume single-instance operation with traditional database backup and recovery procedures.

Metrics and monitoring integration with systems like Prometheus, Grafana, or application performance monitoring (APM) tools is not included. Basic logging will be available, but structured metrics export requires additional development.

Database migration tooling for upgrading schemas, migrating between database engines, or handling data transformations during version upgrades is excluded. Schema changes will require manual database administration.

Containerization and orchestration support with Docker images, Kubernetes manifests, or cloud-native deployment patterns is not included in the core deliverable. The application will be designed as a standard executable that can be containerized separately.

Non-Goal Category	Excluded Feature	Rationale
Grant Types	Implicit Grant Flow	Security vulnerabilities, superseded by PKCE
Grant Types	Device Authorization Grant	Complexity without broad necessity
Grant Types	Password Credentials Grant	Violates OAuth2 security principles
Security	Pushed Authorization Requests	Adds complexity beyond basic compliance
Security	Financial-grade API (FAPI)	Exceeds general-purpose requirements
Security	Rich Authorization Requests	Standard scopes provide sufficient granularity
Federation	SAML 2.0 Integration	Different protocol stack, enterprise complexity
Federation	Social Login Providers	External dependencies, operational complexity
Federation	LDAP/AD Integration	Enterprise complexity, pluggable in future
User Experience	Multi-factor Authentication	Authentication vs. authorization separation
User Experience	Administrative Web Interface	API-first approach sufficient initially
User Experience	User Self-service Portal	Core functionality focus
Operations	High Availability Clustering	Single-instance operational model
Operations	Built-in Metrics/Monitoring	External tooling integration preferred
Operations	Database Migration Tools	Manual administration acceptable initially

Design Philosophy: By explicitly excluding these features, we're prioritizing a secure, compliant, and maintainable core implementation over feature completeness. Each excluded item represents a potential future enhancement that can be added without compromising the foundational architecture.

Pitfall: Scope Creep Through "Simple" Additions

A common mistake is treating excluded features as "quick additions" during development. For example, adding implicit grant support "just for testing" or including basic MFA "since we're already handling user authentication." Each excluded feature was deliberately omitted due to complexity, security implications, or maintenance burden. Resist the temptation to include them without proper design and testing - they inevitably consume more effort than anticipated and can compromise the security or maintainability of core features.

Pitfall: Misunderstanding OAuth2 vs. OIDC Scope

Developers sometimes confuse OAuth2 authorization with general authentication features. OAuth2 is specifically about delegating access to resources, not about user authentication, password management, or session handling. Features like password reset, account lockout, or user registration are authentication concerns that exist outside the OAuth2 specification. Keep the OAuth2 provider focused on authorization delegation rather than expanding into a general authentication system.

Implementation Guidance

This section provides concrete technology recommendations and starter code to begin implementing the goals outlined above. The recommendations prioritize simplicity and security over performance optimization, following the principle that a correct, maintainable implementation should be achieved before optimization efforts.

Technology Recommendations

Component	Simple Option	Advanced Option	Security Consideration
HTTP Framework	net/http (Go standard library)	Gin or Echo framework	Standard library provides better security auditability
Database	SQLite with migrations	PostgreSQL with connection pooling	Both support transactions required for token atomicity
Client Secret Hashing	bcrypt (golang.org/x/crypto)	Argon2 with configurable parameters	bcrypt provides good defaults, Argon2 offers more tuning
JWT Signing	RS256 with crypto/rsa	ES256 with crypto/ecdsa	Both provide asymmetric verification, ES256 has smaller signatures
Random Generation	crypto/rand with encoding/base64	crypto/rand with custom encoding	Never use math/rand for security-sensitive values
Session Storage	In-memory maps with sync.RWMutex	Redis with TTL expiration	In-memory simpler for single-instance deployments
Configuration	Environment variables + flag package	YAML/JSON config files with validation	Environment variables prevent secrets in source code
Logging	Go standard log package	Structured logging (logrus, zap)	Ensure no sensitive data appears in logs

Recommended File Structure

Organize the codebase to separate concerns clearly and support the milestone-based development approach:

```

oauth2-provider/
  cmd/
    server/
      main.go           ← Server entry point, configuration loading
  internal/
    client/
      client.go        ← Milestone 1: Client Registration
      registration.go ← Client data structures and storage
      store.go         ← Client registration API handlers
      client_test.go   ← Client persistence interface and implementations
    auth/
      authorization.go ← Client registration tests
      consent.go       ← Milestone 1: Authorization Endpoint
      pkce.go          ← Authorization request handling
      auth_test.go     ← User consent screen and processing
    token/
      token.go         ← PKCE validation utilities
      jwt.go          ← Authorization flow tests
      refresh.go       ← Authorization endpoint tests
      token_test.go    ← Milestone 2: Token Endpoint
    introspect/
      introspect.go   ← Token generation and exchange
      revoke.go        ← JWT creation and signing utilities
      validation.go    ← Refresh token rotation logic
      introspect_test.go ← Introspection and revocation tests
  userinfo/
    userinfo.go      ← Milestone 3: Token Introspection
    claims.go         ← RFC 7662 introspection endpoint
    consent_mgmt.go   ← RFC 7009 revocation endpoint
    userinfo_test.go  ← JWT validation utilities
  common/
    errors.go         ← Introspection and revocation tests
    random.go         ← UserInfo and consent tests
    database.go       ← Milestone 4: UserInfo Endpoint
    middleware.go     ← Shared utilities
  web/
    templates/
      consent.html    ← OAuth2 error response handling
      error.html      ← Cryptographic random generation
    static/
      styles.css       ← Database connection and migration
    migrations/
      001_clients.sql  ← HTTP middleware (CORS, logging, etc.)
      002_codes.sql    ← OAuth2 error page template
      003_tokens.sql   ← Basic styling for consent pages
      004_consent.sql  ← Client registration table schema
  docs/
    api.md            ← Authorization codes table schema
    deployment.md     ← Refresh tokens table schema
  docker/
    Dockerfile        ← User consent records schema
    docker-compose.yml ← Container build definition
                           ← Deployment and configuration guide
                           ← Development environment setup

```

Core Infrastructure Starter Code

Error Handling Infrastructure (Complete implementation):

```
// internal/common/errors.go

package common

import (
    "encoding/json"
    "fmt"
    "net/http"
)

// OAuth2Error represents a standard OAuth2 error response

type OAuth2Error struct {

    Error          string `json:"error"`

    ErrorDescription string `json:"error_description,omitempty"`

    ErrorURI        string `json:"error_uri,omitempty"`

    State          string `json:"state,omitempty"`

}

// Standard OAuth2 error codes as constants

const (

    ErrorInvalidRequest      = "invalid_request"

    ErrorInvalidClient       = "invalid_client"

    ErrorInvalidGrant         = "invalid_grant"

    ErrorUnauthorizedClient   = "unauthorized_client"

    ErrorUnsupportedGrantType = "unsupported_grant_type"

    ErrorInvalidScope         = "invalid_scope"

    ErrorAccessDenied          = "access_denied"

    ErrorUnsupportedResponseType = "unsupported_response_type"

    ErrorServerError           = "server_error"

    ErrorTemporarilyUnavailable = "temporarily_unavailable"

)

// WriteOAuth2Error sends a standardized OAuth2 error response
```

```
func WriteOAuth2Error(w http.ResponseWriter, statusCode int, errorCode, description string) {

    w.Header().Set("Content-Type", "application/json")

    w.Header().Set("Cache-Control", "no-store")

    w.Header().Set("Pragma", "no-cache")

    w.WriteHeader(statusCode)

    errorResp := OAuth2Error{

        Error:           errorCode,
        ErrorDescription: description,
    }

    json.NewEncoder(w).Encode(errorResp)

}

// WriteOAuth2ErrorWithState includes state parameter for authorization endpoint errors

func WriteOAuth2ErrorWithState(w http.ResponseWriter, statusCode int, errorCode, description, state string) {

    w.Header().Set("Content-Type", "application/json")

    w.Header().Set("Cache-Control", "no-store")

    w.Header().Set("Pragma", "no-cache")

    w.WriteHeader(statusCode)

    errorResp := OAuth2Error{

        Error:           errorCode,
        ErrorDescription: description,
        State:          state,
    }

    json.NewEncoder(w).Encode(errorResp)

}
```

Cryptographic Random Generation (Complete implementation):

GO

```
// internal/common/random.go

package common

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
)

// TokenGenerator provides secure random token generation

type TokenGenerator struct{}


// GenerateAuthorizationCode creates a 32-byte random authorization code

func (tg *TokenGenerator) GenerateAuthorizationCode() (string, error) {

    return tg.generateRandomString(32)
}

// GenerateClientID creates a 16-byte random client identifier

func (tg *TokenGenerator) GenerateClientID() (string, error) {

    return tg.generateRandomString(16)
}

// GenerateClientSecret creates a 32-byte random client secret

func (tg *TokenGenerator) GenerateClientSecret() (string, error) {

    return tg.generateRandomString(32)
}

// GenerateRefreshToken creates a 32-byte random refresh token

func (tg *TokenGenerator) GenerateRefreshToken() (string, error) {

    return tg.generateRandomString(32)
}

// generateRandomString creates a cryptographically secure random string
```

```
func (tg *TokenGenerator) generateRandomString(byteLength int) (string, error) {

    bytes := make([]byte, byteLength)

    if _, err := rand.Read(bytes); err != nil {

        return "", fmt.Errorf("failed to generate random bytes: %w", err)
    }

    // Use URL-safe base64 encoding without padding

    return base64.RawURLEncoding.EncodeToString(bytes), nil
}
```

Core Logic Skeleton Code

Client Registration (Signatures and TODOs only):

GO

```
// internal/client/registration.go

package client

import (
    "time"

    "golang.org/x/crypto/bcrypt"
)

// Client represents an OAuth2 client application

type Client struct {

    ClientID      string      `json:"client_id" db:"client_id"`

    ClientSecret  string      `json:"-" db:"client_secret"`          // Never serialize

    SecretHash    string      `json:"-" db:"secret_hash"`           // bcrypt hash

    Name          string      `json:"name" db:"name"`

    RedirectURIs []string    `json:"redirect_uris" db:"redirect_uris"`

    GrantTypes    []string    `json:"grant_types" db:"grant_types"`

    Scopes        []string    `json:"scopes" db:"scopes"`

    CreatedAt     time.Time   `json:"created_at" db:"created_at"`

}

// ClientStore defines the interface for client persistence

type ClientStore interface {

    CreateClient(client *Client) error

    GetClientByID(clientID string) (*Client, error)

    ValidateClientSecret(clientID, clientSecret string) error

}

// RegisterClient creates a new OAuth2 client with generated credentials

func RegisterClient(store ClientStore, name string, redirectURIs []string) (*Client, error) {

    // TODO 1: Generate a unique client_id using TokenGenerator

    // TODO 2: Generate a random client_secret using TokenGenerator

    // TODO 3: Hash the client_secret using bcrypt with cost 12
```

```
// TODO 4: Validate that redirectURIs is not empty and contains valid URLs

// TODO 5: Create Client struct with generated values and current timestamp

// TODO 6: Store the client using ClientStore.CreateClient()

// TODO 7: Return the client with the plaintext secret (only time it's available)

// Hint: Never store the plaintext secret, only the bcrypt hash

}

// GenerateClientSecret creates a client secret and its bcrypt hash

func GenerateClientSecret() (secret, hash string, err error) {

    // TODO 1: Use TokenGenerator to create a 32-byte random secret

    // TODO 2: Hash the secret using bcrypt.GenerateFromPassword with cost 12

    // TODO 3: Return both the plaintext secret and hash

    // Hint: bcrypt cost 12 provides good security/performance balance

}
```

Authorization Request Processing (Signatures and TODOs only):

```
// internal/auth/authorization.go

package auth

import "time"

// AuthorizationRequest represents an OAuth2 authorization request

type AuthorizationRequest struct {

    ClientID          string `json:"client_id"`
    RedirectURI       string `json:"redirect_uri"`
    Scope             string `json:"scope"`
    State             string `json:"state"`
    ResponseType      string `json:"response_type"`
    CodeChallenge     string `json:"code_challenge"`
    CodeChallengeMethod string `json:"code_challenge_method"`

}

// AuthorizationCode represents a generated authorization code

type AuthorizationCode struct {

    Code          string   `json:"code" db:"code"`
    ClientID     string   `json:"client_id" db:"client_id"`
    UserID        string   `json:"user_id" db:"user_id"`
    RedirectURI   string   `json:"redirect_uri" db:"redirect_uri"`
    Scope         string   `json:"scope" db:"scope"`
    CodeChallenge string   `json:"code_challenge" db:"code_challenge"`
    ExpiresAt     time.Time `json:"expires_at" db:"expires_at"`
    Used          bool     `json:"used" db:"used"`

}

// HandleAuthorizationRequest processes an OAuth2 authorization request

func HandleAuthorizationRequest(req *AuthorizationRequest, clientStore ClientStore) (*AuthorizationCode, error) {

    // TODO 1: Validate that response_type equals "code"
```

```

    // TODO 2: Lookup client by client_id and verify it exists

    // TODO 3: Validate redirect_uri exactly matches one of client's registered URIs

    // TODO 4: Validate that code_challenge is present and uses "S256" method (PKCE mandatory)

    // TODO 5: Parse and validate requested scopes against client's allowed scopes

    // TODO 6: Generate cryptographically random authorization code

    // TODO 7: Create AuthorizationCode with 10-minute expiration

    // TODO 8: Store the authorization code in database

    // TODO 9: Return the authorization code for redirect

    // Hint: All validation failures should return OAuth2Error with appropriate error codes

}

```

Milestone Checkpoints

Milestone 1 Checkpoint: After implementing client registration and authorization endpoint:

```

# Test client registration

curl -X POST http://localhost:8080/oauth2/register \
-H "Content-Type: application/json" \
-d '{"name": "Test App", "redirect_uris": ["https://example.com/callback"]}'


# Expected response: JSON with client_id, client_secret, and registration details

# Verify: Client secret should be 43+ characters, client_id should be shorter


# Test authorization endpoint

curl "http://localhost:8080/oauth2/authorize?
response_type=code&client_id=CLIENT_ID&redirect_uri=https://example.com/callback&scope=openid%20profile&state=xyz&code_challenge=E9Me1hoa20wvFrEMTJguChaoeK1t8URWbuGJSstw-cM&code_challenge_method=S256"

# Expected response: Redirect to consent screen or error page

# Verify: Invalid client_id returns invalid_client error

# Verify: Mismatched redirect_uri returns invalid_request error

```

Milestone 2 Checkpoint: After implementing token endpoint:

```

# Test token exchange                                         BASH

curl -X POST http://localhost:8080/oauth2/token \
      -H "Authorization: Basic BASE64(client_id:client_secret)" \
      -d
"grant_type=authorization_code&code=AUTHORIZATION_CODE&redirect_uri=https://example.com/callback&code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_ww1gFWFOEjXk"

# Expected response: JSON with access_token (JWT), refresh_token, token_type=Bearer, expires_in

# Verify: JWT has three parts separated by dots

# Verify: JWT payload contains sub, iss, aud, exp, iat, scope claims

# Verify: Used authorization code cannot be exchanged again

```

Language-Specific Security Tips

Go-Specific Security Considerations:

- Use `crypto/rand` instead of `math/rand` for all token generation
- Use `crypto/subtle.ConstantTimeCompare()` for comparing secrets to prevent timing attacks
- Use `golang.org/x/crypto/bcrypt` with cost 12 for hashing client secrets
- Use `crypto/rsa` with 2048+ bit keys or `crypto/ecdsa` with P-256 curves for JWT signing
- Use `encoding/base64.RawURLEncoding` for tokens to avoid padding characters in URLs
- Set appropriate HTTP headers: `Cache-Control: no-store`, `Pragma: no-cache` for token responses
- Use `context.Context` with timeouts for database operations to prevent resource exhaustion
- Validate all inputs using explicit checks rather than relying on SQL or template injection protection

High-Level Architecture

Milestone(s): This section provides architectural foundation for all milestones (1-4), establishing the component structure that will be built incrementally

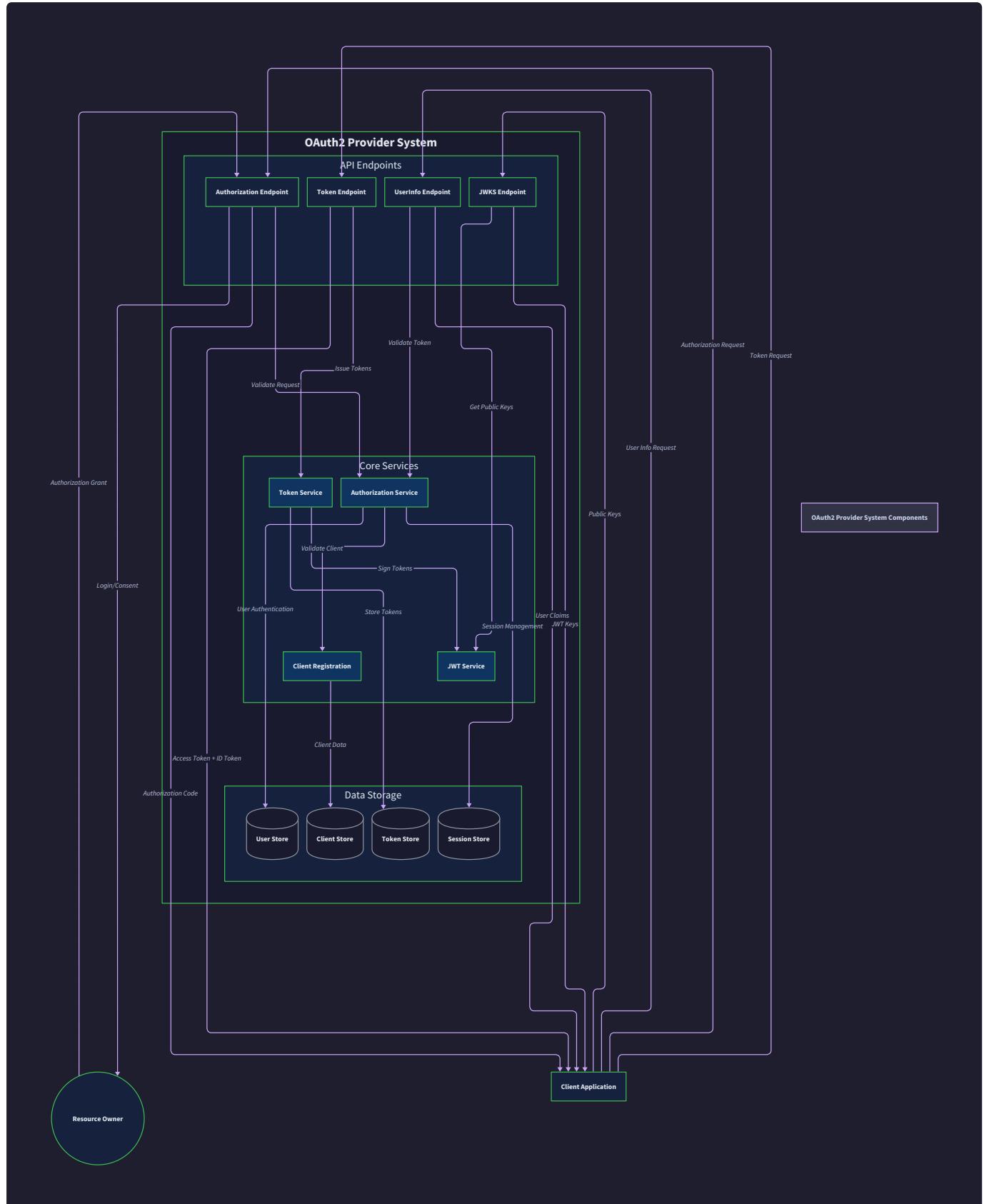
Mental Model: Digital Bank Infrastructure

Think of our OAuth2 provider as a **digital bank infrastructure** that manages authorization rather than money. Just as a physical bank has distinct departments working together - the front desk registers new customers, the loan department approves credit applications, the vault manages assets, and the audit department tracks all transactions - our OAuth2 provider has specialized components that collaborate to securely manage digital permissions.

The **Authorization Endpoint** is like the loan application desk where customers (users) come to approve third-party access requests. The **Token Endpoint** acts like the vault, exchanging approved authorizations for valuable digital tokens. The **Client Registration** component functions like the business accounts department, vetting and registering third-party

applications that want to access customer data. The **UserInfo Endpoint** operates like the customer service desk, providing account information to authorized parties.

Each department has its own specialized storage systems, security protocols, and communication channels, but they all work together through well-defined interfaces to ensure that every authorization transaction is secure, auditable, and follows banking regulations (OAuth2/OIDC specifications).



Component Overview: Main Components and Their Responsibilities

Our OAuth2 provider consists of six primary components that work together to implement the complete authorization server functionality. Each component has distinct responsibilities and maintains its own data while communicating through well-defined interfaces.

Core Authorization Components

The **Authorization Endpoint Component** serves as the entry point for OAuth2 flows, handling initial authorization requests from client applications. This component validates client credentials, manages PKCE security mechanisms, presents consent screens to users, and generates authorization codes bound to specific clients and users. It implements the authorization server role defined in RFC 6749, ensuring that all authorization requests follow proper security protocols and user consent is obtained before any access is granted.

The **Token Endpoint Component** functions as the secure token exchange facility, responsible for converting authorization codes into JWT access tokens and refresh tokens. This component handles multiple OAuth2 grant types including authorization code grant and client credentials grant, implements refresh token rotation for enhanced security, and generates cryptographically signed JWTs with appropriate claims and expiration times. It serves as the primary interface for client applications to obtain the credentials they need to access protected resources.

The **Client Registration Component** manages the lifecycle of OAuth2 client applications, handling registration, credential generation, and ongoing client management. This component generates unique client identifiers, securely hashes and stores client secrets, validates redirect URIs, and maintains client configuration including allowed grant types and scopes. It ensures that only properly registered and authenticated clients can participate in OAuth2 flows.

Token Management Components

The **Token Introspection and Revocation Component** provides RFC 7662 and RFC 7009 compliant endpoints for token validation and invalidation. The introspection functionality allows resource servers to validate tokens and retrieve metadata about their status and associated permissions. The revocation functionality enables immediate token invalidation for security incidents or user-initiated logouts, with support for refresh token family tracking to detect token theft.

The **UserInfo and Consent Management Component** implements the OpenID Connect UserInfo endpoint and manages user consent decisions throughout their lifecycle. This component maps granted scopes to specific user claims, filters profile information based on authorized permissions, and maintains persistent records of user consent decisions. It ensures that users retain control over their data and can revoke previously granted permissions.

Supporting Infrastructure Components

The **Data Storage Layer** provides persistent storage for all system entities including clients, users, tokens, authorization codes, and consent records. This layer abstracts the underlying storage technology and provides consistent interfaces for data access, ensuring data integrity and supporting concurrent access patterns required by the OAuth2 flows.

Each component maintains clear boundaries and communicates through well-defined interfaces, enabling the system to scale horizontally and be maintained by different development teams. The architecture supports both synchronous request-response patterns for user-facing flows and asynchronous processing for token cleanup and audit logging.

Component Name	Primary Responsibility	Key Data Managed	External Interfaces
Authorization Endpoint	Handle authorization requests, PKCE, consent	Authorization codes, PKCE challenges	HTTP endpoints for auth flow
Token Endpoint	Exchange codes for tokens, refresh flows	Access tokens, refresh tokens, JWT signing	HTTP endpoints for token operations
Client Registration	Register and authenticate OAuth2 clients	Client credentials, redirect URIs, scopes	Registration API, client validation
Token Introspection/Revocation	Validate and invalidate tokens	Token status, revocation lists	RFC 7662/7009 compliant endpoints
UserInfo/Consent	Provide user claims, manage consent	User profiles, consent decisions	OIDC UserInfo endpoint, consent UI
Data Storage	Persist all system entities	Clients, users, tokens, codes, consent	Storage interfaces for all components

Component Interaction Patterns

Components interact through **synchronous HTTP APIs** for user-facing operations and **shared data storage** for state coordination. The Authorization Endpoint communicates with Client Registration to validate client applications and with Data Storage to persist authorization codes. The Token Endpoint retrieves authorization codes and client information from storage while generating new tokens that are tracked by the Introspection component.

Event-driven patterns handle asynchronous operations like token cleanup and audit logging. When tokens are created, revoked, or expire, events are published to trigger appropriate cleanup actions and security monitoring. This ensures that the system can scale beyond simple request-response patterns while maintaining consistency.

Security boundaries are enforced at component interfaces, with each component validating inputs and applying appropriate security controls. Client authentication is centralized in the Client Registration component, token validation is handled by the Introspection component, and user authentication is managed by the Authorization Endpoint.

Decision: Microservice vs Monolith Architecture

- **Context:** OAuth2 providers can be built as single monolithic services or distributed microservice architectures
- **Options Considered:** Single process monolith, containerized microservices, serverless functions
- **Decision:** Start with modular monolith, architect for microservice extraction
- **Rationale:** OAuth2 flows require tight consistency and low latency between components. A monolith reduces network overhead and simplifies deployment while maintaining clear component boundaries for future extraction
- **Consequences:** Faster development and simpler operations initially, with ability to extract high-load components (like token validation) into separate services as scale requirements emerge

Recommended File Structure: How to Organize the Codebase for Maintainability

A well-organized file structure is crucial for an OAuth2 provider because the codebase must handle complex security protocols, multiple HTTP endpoints, cryptographic operations, and data persistence while remaining maintainable as the

system evolves. The structure should clearly separate concerns, make dependencies explicit, and support both unit testing and integration testing.

Project Root Organization

The project follows Go's standard layout conventions with clear separation between application entry points, internal business logic, external APIs, and supporting infrastructure. This organization makes it easy for new developers to understand the system architecture by examining the directory structure.

```
oauth2-provider/
├── cmd/                                # Application entry points
│   └── server/                           # Main OAuth2 server
│       ├── main.go                        # Server startup and configuration
│       └── config.yaml                   # Default configuration
│   └── migrate/                          # Database migration utility
│       └── main.go                        # Schema setup and migrations
├── internal/                            # Private application code
├── pkg/                                 # Public library code
├── api/                                 # API definitions and schemas
├── web/                                 # Static web assets
├── deployments/                         # Deployment configurations
├── docs/                                # Additional documentation
└── scripts/                             # Build and development scripts
```

Internal Component Organization

The `internal/` directory contains all private business logic organized by component boundaries. Each component is self-contained with its own models, handlers, storage interfaces, and tests. This structure supports the component architecture while preventing circular dependencies.

```
internal/
├── auth/                                # Authorization Endpoint Component
│   ├── handlers.go                      # HTTP handlers for authorization flow
│   ├── pkce.go                           # PKCE implementation
│   ├── consent.go                        # User consent management
│   ├── codes.go                          # Authorization code generation
│   ├── models.go                         # Authorization request/response models
│   ├── handlers_test.go                  # Handler unit tests
│   └── pkce_test.go                      # PKCE unit tests
├── token/                                # Token Endpoint Component
│   ├── handlers.go                      # Token endpoint HTTP handlers
│   ├── jwt.go                            # JWT generation and signing
│   ├── refresh.go                        # Refresh token rotation
│   ├── grants.go                         # Grant type implementations
│   ├── models.go                         # Token request/response models
│   └── handlers_test.go                  # Token endpoint tests
├── clients/                             # Client Registration Component
│   ├── registration.go                 # Client registration logic
│   ├── validation.go                   # Client credential validation
│   ├── storage.go                      # Client storage interface
│   ├── models.go                       # Client data models
│   └── registration_test.go           # Registration tests
├── introspection/                      # Token Introspection/Revocation
│   ├── handlers.go                      # RFC 7662/7009 endpoint handlers
│   ├── validation.go                   # Token validation logic
│   ├── revocation.go                  # Token revocation tracking
│   └── handlers_test.go                # Introspection tests
├── userinfo/                            # UserInfo/Consent Component
│   ├── handlers.go                      # OIDC UserInfo endpoint
│   ├── claims.go                        # Scope-to-claims mapping
│   ├── consent.go                       # Consent persistence
│   └── handlers_test.go                # UserInfo tests
├── storage/                             # Data Storage Layer
│   ├── interfaces.go                  # Storage interface definitions
│   ├── memory/                         # In-memory implementation
│   │   ├── clients.go                  # Client storage
│   │   ├── tokens.go                  # Token storage
│   │   └── users.go                   # User storage
│   ├── postgres/                       # PostgreSQL implementation
│   │   ├── clients.go                  # Client queries
│   │   ├── tokens.go                  # Token queries
│   │   └── migrations/               # Database schema
│   │       └── postgres.go            # Database connection
│   └── redis/                           # Redis for caching/sessions
│       └── cache.go                   # Cache implementation
├── crypto/                              # Cryptographic Operations
│   ├── keys.go                          # Key management
│   ├── jwt.go                           # JWT signing/verification
│   ├── random.go                        # Secure random generation
│   └── crypto_test.go                  # Crypto tests
├── middleware/                          # HTTP Middleware
│   ├── auth.go                          # Client authentication
│   ├── cors.go                          # CORS handling
│   ├── logging.go                       # Request logging
│   └── ratelimit.go                    # Rate limiting
└── config/                              # Configuration Management
    ├── config.go                        # Configuration loading
    ├── validation.go                   # Config validation
    └── defaults.go                     # Default values
```

```
└── server/          # HTTP Server Setup
    ├── server.go    # Server initialization
    ├── routes.go    # Route registration
    └── middleware.go # Middleware chain setup
```

Public Library Organization

The `pkg/` directory contains reusable components that could be imported by other projects, such as OAuth2 client libraries or JWT validation utilities. This separation allows the core business logic to remain private while providing useful tools to the broader ecosystem.

```
pkg/
├── oauth2/          # OAuth2 client library
│   ├── client.go    # OAuth2 client implementation
│   ├── tokens.go    # Token management
│   └── errors.go    # Error types
├── jwt/             # JWT validation library
│   ├── validator.go # JWT validation logic
│   ├── claims.go    # Claims processing
│   └── keys.go       # Key management
└── oidc/            # OIDC discovery and validation
    ├── discovery.go # Well-known endpoint
    └── validation.go # OIDC token validation
```

Supporting Directories

Additional directories support development, deployment, and documentation workflows while keeping the core codebase focused on business logic.

```
api/
├── oauth2/          # OAuth2 API definitions
│   ├── openapi.yaml # OpenAPI specification
│   └── schemas/     # JSON schemas for validation
└── oidc/            # OIDC API definitions
    └── discovery.json # Well-known configuration

web/
├── static/          # Static assets for consent UI
│   ├── css/          # Stylesheets
│   ├── js/           # JavaScript
│   └── images/        # Images and icons
└── templates/        # HTML templates
    ├── consent.html # User consent page
    ├── error.html   # Error page template
    └── login.html   # Login page template

deployments/
├── docker/          # Docker configurations
│   ├── Dockerfile    # Application container
│   └── docker-compose.yml # Development environment
└── kubernetes/      # Kubernetes manifests
    ├── deployment.yaml # Application deployment
    ├── service.yaml   # Service definition
    └── configmap.yaml # Configuration
└── terraform/        # Infrastructure as code
    └── main.tf         # Cloud resources
```

File Naming Conventions

Consistent file naming makes the codebase navigable and predictable. Files are named to clearly indicate their purpose and scope within each component.

File Type	Naming Pattern	Purpose	Examples
HTTP Handlers	<code>handlers.go</code>	HTTP endpoint implementations	<code>auth/handlers.go</code> , <code>token/handlers.go</code>
Data Models	<code>models.go</code>	Struct definitions and validation	<code>clients/models.go</code> , <code>token/models.go</code>
Storage Interfaces	<code>interfaces.go</code>	Abstract storage contracts	<code>storage/interfaces.go</code>
Storage Implementations	<code>{technology}.go</code>	Concrete storage implementations	<code>storage/postgres/postgres.go</code>
Business Logic	<code>{feature}.go</code>	Core business functionality	<code>auth/pkce.go</code> , <code>token/jwt.go</code>
Tests	<code>{filename}_test.go</code>	Unit and integration tests	<code>handlers_test.go</code> , <code>pkce_test.go</code>
Integration Tests	<code>{component}_integration_test.go</code>	Cross-component testing	<code>auth_integration_test.go</code>

This file organization supports several important development practices:

Clear Dependencies: The structure makes component dependencies visible through import statements, preventing circular dependencies and ensuring clean architecture boundaries.

Testability: Each component can be unit tested independently, and the separation of interfaces from implementations enables effective mocking and integration testing.

Team Scalability: Different teams can own different components without stepping on each other, as component boundaries are clearly defined in the file system.

Security Auditing: Security-critical code is concentrated in specific files (`crypto/`, authentication handlers, token validation), making security reviews more focused and effective.

Key Insight: The file structure mirrors the component architecture, making the system's design visible in the codebase organization. This reduces cognitive load for developers and makes architectural violations obvious through inappropriate cross-directory imports.

Technology Stack: Recommended Libraries and Tools for Implementation

The technology stack for an OAuth2 provider must balance security, performance, standards compliance, and developer productivity. The choices made here will impact both the development experience and the operational characteristics of the system in production environments.

Core Technology Decisions

Go serves as the primary implementation language due to its excellent HTTP server capabilities, strong cryptographic standard library, built-in concurrency support, and static binary deployment model. Go's explicit error handling reduces the risk of security bugs common in OAuth2 implementations, while its performance characteristics support the high-throughput token operations required by authorization servers.

PostgreSQL provides the primary data store for persistent entities like clients, users, tokens, and consent records.

PostgreSQL offers ACID transactions required for consistent token operations, JSON column support for flexible client metadata storage, and excellent performance for the read-heavy workloads common in OAuth2 systems. The robust ecosystem and operational tooling make it suitable for production deployments.

Redis serves as the caching layer and session store for temporary data like authorization codes, rate limiting counters, and revoked token tracking. Redis provides the sub-second latency required for authorization flows and supports automatic expiration for time-bounded entities like authorization codes.

Technology Category	Simple Option	Advanced Option	Recommendation for Learning
HTTP Framework	<code>net/http</code> standard library	<code>gin-gonic/gin</code> or <code>gorilla/mux</code>	Start with <code>net/http</code> for clarity
Database	SQLite for development	PostgreSQL for production	SQLite initially, PostgreSQL for Milestone 3+
Caching	In-memory maps	Redis with persistence	In-memory for Milestones 1-2, Redis for production
JWT Library	<code>golang-jwt/jwt/v4</code>	Custom implementation with <code>crypto/</code>	Use <code>golang-jwt/jwt/v4</code> for reliability
Password Hashing	<code>golang.org/x/crypto/bcrypt</code>	<code>golang.org/x/crypto/argon2</code>	<code>bcrypt</code> for simplicity
Configuration	Environment variables	<code>spf13/viper</code> with multiple sources	Environment variables initially
Logging	<code>log/slog</code> standard library	<code>sirupsen/logrus</code> or <code>uber-go/zap</code>	<code>log/slog</code> for structured logging
Testing	<code>testing</code> standard library	<code>stretchr/testify</code> for assertions	Standard library with <code>testify</code> for convenience

HTTP Server and Routing

The HTTP server foundation handles OAuth2 protocol endpoints, serves consent UI, and provides administrative interfaces. The choice between standard library and frameworks involves trade-offs between simplicity and convenience features.

Standard Library Approach (`net/http`): Using Go's built-in HTTP server provides complete control over request handling, middleware chains, and error responses. This approach requires more boilerplate but offers better understanding of HTTP fundamentals and fewer dependencies. For learning OAuth2 concepts, this transparency is valuable.

Framework Approach (`gin-gonic/gin`): Web frameworks provide convenient request binding, middleware systems, and response helpers that reduce boilerplate for OAuth2 endpoint implementations. However, they can obscure important details about HTTP handling that are crucial for security-sensitive OAuth2 implementations.

```
Recommended HTTP Stack:  
├── net/http (server)          # Core HTTP server  
├── gorilla/mux (routing)     # URL routing and path variables  
├── gorilla/handlers (middleware) # CORS, logging, compression  
└── html/template (templating)  # Consent screen rendering
```

Cryptographic Libraries

OAuth2 providers require multiple cryptographic operations including JWT signing, client secret hashing, secure random generation, and PKCE challenge verification. Go's `crypto/` package provides excellent foundations, but additional libraries handle higher-level operations.

JWT Operations: The `golang-jwt/jwt/v4` library provides robust JWT creation, parsing, and validation with support for multiple signing algorithms. It handles edge cases like algorithm confusion attacks and provides secure defaults for token operations.

Password Hashing: `golang.org/x/crypto/bcrypt` offers appropriate password hashing for client secrets with configurable work factors. The library handles salt generation and provides constant-time comparison functions to prevent timing attacks.

Random Generation: Go's `crypto/rand` provides cryptographically secure random number generation for authorization codes, client secrets, and token generation. This is critical for OAuth2 security properties.

Cryptographic Operation	Library	Key Functions	Security Considerations
JWT Signing/Verification	<code>golang-jwt/jwt/v4</code>	<code>jwt.NewWithClaims()</code> , <code>jwt.Parse()</code>	Use RS256 or ES256, validate all claims
Client Secret Hashing	<code>golang.org/x/crypto/bcrypt</code>	<code>bcrypt.GenerateFromPassword()</code> , <code>bcrypt.CompareHashAndPassword()</code>	Cost factor 12+, constant-time comparison
Secure Random Generation	<code>crypto/rand</code>	<code>rand.Read()</code> , <code>rand.Int()</code>	Always check errors, use appropriate entropy
PKCE Challenge Verification	<code>crypto/sha256</code>	<code>sha256.Sum256()</code>	Base64URL encoding, constant-time comparison
TLS Certificate Management	<code>crypto/tls</code>	<code>tls.LoadX509KeyPair()</code>	Regular rotation, proper certificate validation

Database and Storage

The storage layer must handle both persistent entities (clients, users, consent) and temporary entities (authorization codes, sessions) with different consistency and performance requirements.

PostgreSQL Configuration: The database schema uses separate tables for each entity type with appropriate indexes for OAuth2 query patterns. JSON columns store flexible client metadata and user profiles while maintaining queryable structure.

Connection Management: Database connections use `lib/pq` or `pgx` drivers with connection pooling configured for OAuth2 workload characteristics. Token operations are read-heavy with occasional writes, requiring optimized connection pool settings.

Migration Strategy: Database schema changes use versioned migrations managed by `golang-migrate/migrate` or similar tools. OAuth2 schemas evolve as new features are added, requiring careful migration planning to avoid service disruptions.

```

Database Architecture:
├── PostgreSQL (primary)          # Persistent entities
|   ├── oauth_clients             # Client registrations
|   ├── oauth_codes               # Authorization codes
|   ├── oauth_tokens              # Issued tokens
|   ├── users                     # User accounts
|   └── user_consent              # Consent decisions
├── Redis (cache)                # Temporary data
|   ├── sessions                  # User sessions
|   ├── rate_limits               # Rate limiting counters
|   └── revoked_tokens            # Revocation tracking
└── Local files (keys)           # JWT signing keys
    ├── private.pem               # Token signing key
    └── public.pem                 # Token verification key

```

Development and Testing Tools

OAuth2 providers require specialized testing approaches to verify security properties, protocol compliance, and integration behavior with real OAuth2 clients.

Unit Testing: Standard Go testing with `testify` assertions provides component-level verification. Mock implementations of storage interfaces enable isolated testing of business logic without database dependencies.

Integration Testing: `testcontainers-go` provides real database instances for integration tests, ensuring that storage implementations work correctly with actual database systems. This catches SQL compatibility issues and transaction behavior differences.

OAuth2 Protocol Testing: Custom test clients simulate various OAuth2 flows including error conditions, malformed requests, and security attacks. These tests verify that the implementation correctly handles edge cases defined in OAuth2 specifications.

Security Testing: Specialized tools verify cryptographic operations, timing attack resistance, and input validation completeness. Static analysis tools like `gosec` identify common security patterns in OAuth2 implementations.

Testing Category	Primary Tool	Secondary Tools	Purpose
Unit Tests	<code>testing</code> + <code>testify/assert</code>	<code>testify/mock</code> for interfaces	Component isolation and logic verification
Integration Tests	<code>testcontainers-go</code>	<code>dockertest</code> alternative	Real database and service integration
HTTP Endpoint Tests	<code>net/http/httpptest</code>	<code>gavv/httpexpect</code> for fluent APIs	OAuth2 protocol compliance
Security Tests	Custom tooling	<code>gosec</code> , <code>nancy</code> for vulnerabilities	Cryptographic and input validation
Load Tests	<code>k6</code> or <code>bombardier</code>	<code>pprof</code> for profiling	Performance under OAuth2 workloads
Browser Tests	<code>chromedp</code>	<code>selenium</code> for complex UI	User consent flow validation

Configuration and Deployment

OAuth2 providers require careful configuration management for security-sensitive settings like signing keys, database credentials, and endpoint URLs. The configuration system must support both development flexibility and production security requirements.

Configuration Sources: The system reads configuration from environment variables, configuration files, and command-line flags in that priority order. This supports both containerized deployments (environment variables) and traditional deployments (configuration files).

Secret Management: Sensitive values like JWT signing keys and database passwords are loaded from secure sources like HashiCorp Vault, Kubernetes secrets, or cloud provider secret managers rather than being stored in configuration files.

Environment-Specific Settings: Different deployment environments (development, staging, production) require different token lifetimes, security policies, and integration endpoints. The configuration system supports environment-specific overrides while maintaining consistent base settings.

Decision: JWT Signing Algorithm Selection

- **Context:** OAuth2 access tokens can use symmetric (HS256) or asymmetric (RS256/ES256) signing algorithms
- **Options Considered:** HS256 with shared secret, RS256 with RSA keypair, ES256 with ECDSA keypair
- **Decision:** Use RS256 for access tokens, HS256 for internal tokens
- **Rationale:** RS256 enables distributed token validation by resource servers without sharing secrets, while HS256 is sufficient for internal tokens like refresh tokens where the authorization server is the only validator
- **Consequences:** Requires RSA key management and slightly higher CPU costs for signing/verification, but enables proper OAuth2 architecture with independent resource servers

This technology stack provides a solid foundation for implementing a production-ready OAuth2 provider while maintaining code clarity for learning purposes. The choices prioritize security, standards compliance, and operational simplicity over cutting-edge features that might introduce complexity or security risks.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation
HTTP Server	<code>net/http</code> standard library	<code>gin-gonic/gin</code> framework	Start with <code>net/http</code> for transparency
Database	SQLite with <code>mattn/go-sqlite3</code>	PostgreSQL with <code>lib/pq</code>	SQLite for learning, PostgreSQL for production
Caching	<code>sync.Map</code> in-memory	Redis with <code>go-redis/redis/v8</code>	In-memory initially, Redis for scale
JWT Library	<code>golang-jwt/jwt/v4</code>	Custom with <code>crypto/rsa</code>	Use <code>golang-jwt/jwt/v4</code> for reliability
Configuration	Environment variables	<code>spf13/viper</code> multi-source	Environment variables for simplicity
Logging	<code>log/slog</code> structured logging	<code>uber-go/zap</code> high-performance	<code>log/slog</code> for built-in structured logging
Testing	<code>testing</code> + <code>testify/assert</code>	<code>testify/suite</code> + <code>testcontainers</code>	Standard testing with <code>testify</code> assertions
Password Hashing	<code>golang.org/x/crypto/bcrypt</code>	<code>golang.org/x/crypto/argon2</code>	<code>bcrypt</code> with cost factor 12

Recommended File Structure

This structure organizes the OAuth2 provider for maximum maintainability while supporting incremental milestone development:

```
oauth2-provider/
├── cmd/
│   └── server/
│       ├── main.go
│       └── config.yaml
├── internal/
│   ├── config/
│   │   ├── config.go
│   │   └── validation.go
│   ├── storage/
│   │   ├── interfaces.go
│   │   ├── memory/
│   │   │   ├── clients.go
│   │   │   ├── users.go
│   │   │   └── tokens.go
│   │   └── sqlite/
│   │       ├── clients.go
│   │       ├── users.go
│   │       ├── tokens.go
│   │       └── migrations.sql
│   ├── crypto/
│   │   ├── random.go
│   │   ├── jwt.go
│   │   └── keys.go
│   ├── clients/
│   │   ├── registration.go
│   │   ├── validation.go
│   │   ├── handlers.go
│   │   └── models.go
│   ├── auth/
│   │   ├── handlers.go
│   │   ├── pkce.go
│   │   ├── consent.go
│   │   ├── codes.go
│   │   └── models.go
│   ├── token/
│   │   ├── handlers.go
│   │   ├── jwt.go
│   │   ├── refresh.go
│   │   ├── grants.go
│   │   └── models.go
│   ├── introspection/
│   │   ├── handlers.go
│   │   ├── validation.go
│   │   └── models.go
│   ├── userinfo/
│   │   ├── handlers.go
│   │   ├── claims.go
│   │   ├── consent.go
│   │   └── models.go
│   ├── middleware/
│   │   ├── auth.go
│   │   ├── logging.go
│   │   └── errors.go
│   └── server/
│       ├── server.go
│       ├── routes.go
│       └── handlers.go
└── pkg/
```

```
|   └── oauth2client/           # OAuth2 client library for testing
|       ├── client.go          # Client implementation
|       └── flows.go           # Flow implementations
├── web/                         # Static web assets
|   ├── templates/              # User consent page
|   |   ├── consent.html       # User login page
|   |   └── login.html         # Error page template
|   └── static/                 # Basic styling
|       ├── css/style.css      # Consent form JavaScript
|       └── js/consent.js        # Database migrations
|   └── migrations/             # Initial database schema
|       ├── 001_initial_schema.sql # Consent management schema
|       └── 002_add_consent_table.sql
├── docs/                         # Additional documentation
├── scripts/                      # Development and build scripts
|   ├── setup.sh                  # Development environment setup
|   └── test.sh                   # Test execution script
└── deployments/                 # Deployment configurations
    ├── docker-compose.yml       # Local development environment
    └── Dockerfile                # Container image definition
```

Infrastructure Starter Code

Configuration Management (`internal/config/config.go`):

```
package config
```

GO

```
import (
```

```
    "os"
```

```
    "strconv"
```

```
    "time"
```

```
)
```

```
type Config struct {
```

```
    Server ServerConfig
```

```
    Database DatabaseConfig
```

```
    JWT JWTConfig
```

```
    OAuth OAuth2Config
```

```
}
```

```
type ServerConfig struct {
```

```
    Port          string
```

```
    ReadTimeout   time.Duration
```

```
    WriteTimeout  time.Duration
```

```
}
```

```
type DatabaseConfig struct {
```

```
    Driver      string
```

```
    URL        string
```

```
    MaxConns   int
```

```
}
```

```
type JWTConfig struct {
```

```
    PrivateKeyPath string
```

```
    PublicKeyPath  string
```

```
    Algorithm     string
```

```
    AccessTTL     time.Duration
```

```
    RefreshTTL      time.Duration
}

}

type OAuth2Config struct {
    AuthorizationCodeTTL time.Duration
    RequirePKCE         bool
    AllowedScopes       []string
}

// Load reads configuration from environment variables with sensible defaults

func Load() (*Config, error) {
    cfg := &Config{
        Server: ServerConfig{
            Port:           getEnv("PORT", "8080"),
            ReadTimeout:   parseDuration(getEnv("READ_TIMEOUT", "30s")),
            WriteTimeout:  parseDuration(getEnv("WRITE_TIMEOUT", "30s")),
        },
        Database: DatabaseConfig{
            Driver:     getEnv("DB_DRIVER", "sqlite3"),
            URL:       getEnv("DB_URL", "./oauth2.db"),
            MaxConns:  parseInt(getEnv("DB_MAX_CONNS", "10")),
        },
        JWT:  JWTConfig{
            PrivateKeyPath: getEnv("JWT_PRIVATE_KEY", "./keys/private.pem"),
            PublicKeyPath:  getEnv("JWT_PUBLIC_KEY", "./keys/public.pem"),
            Algorithm:     getEnv("JWT_ALGORITHM", "RS256"),
            AccessTTL:     parseDuration(getEnv("ACCESS_TOKEN_TTL", "15m")),
            RefreshTTL:    parseDuration(getEnv("REFRESH_TOKEN_TTL", "720h")),
        },
        OAuth: OAuth2Config{
    }
}
```

```
        AuthorizationCodeTTL: parseDuration(getEnv("AUTH_CODE_TTL", "10m")),

        RequirePKCE: parseBool(getEnv("REQUIRE_PKCE", "true")),

        AllowedScopes: []string{"openid", "profile", "email"},

    },
}

return cfg, cfg.validate()
}

func getEnv(key, defaultValue string) string {

    if value := os.Getenv(key); value != "" {

        return value
    }

    return defaultValue
}

func parseInt(s string) int {

    i, _ := strconv.Atoi(s)

    return i
}

func parseDuration(s string) time.Duration {

    d, _ := time.ParseDuration(s)

    return d
}

func parseBool(s string) bool {

    b, _ := strconv.ParseBool(s)

    return b
}

func (c *Config) validate() error {
```

```
// TODO: Add configuration validation logic  
  
// Check file paths exist, durations are reasonable, etc.  
  
return nil  
  
}
```

Secure Random Generation (`internal/crypto/random.go`):

```
package crypto

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
)

// TokenGenerator provides cryptographically secure random token generation

type TokenGenerator struct{}


// NewTokenGenerator creates a new token generator

func NewTokenGenerator() *TokenGenerator {
    return &TokenGenerator{}
}

// GenerateAuthorizationCode creates a cryptographically secure authorization code

func (t *TokenGenerator) GenerateAuthorizationCode() (string, error) {
    // Generate 32 bytes of random data for authorization code
    bytes := make([]byte, 32)

    if _, err := rand.Read(bytes); err != nil {
        return "", fmt.Errorf("failed to generate random bytes: %w", err)
    }

    // Return base64url encoded string (URL-safe, no padding)
    return base64.RawURLEncoding.EncodeToString(bytes), nil
}

// GenerateClientSecret creates a client secret and its bcrypt hash

func (t *TokenGenerator) GenerateClientSecret() (secret, hash string, err error) {
    // TODO: Implement client secret generation using crypto/rand
    // TODO: Hash the secret using bcrypt with appropriate cost factor
}
```

GO

```

// TODO: Return both the plain secret (for client) and hash (for storage)

return "", "", fmt.Errorf("not implemented")

}

// GenerateRefreshToken creates a cryptographically secure refresh token

func (t *TokenGenerator) GenerateRefreshToken() (string, error) {

    // Generate 32 bytes of random data for refresh token

    bytes := make([]byte, 32)

    if _, err := rand.Read(bytes); err != nil {

        return "", fmt.Errorf("failed to generate random bytes: %w", err)

    }

    return base64.RawURLEncoding.EncodeToString(bytes), nil

}

// GenerateState creates a CSRF-protection state parameter

func (t *TokenGenerator) GenerateState() (string, error) {

    // Generate 16 bytes for state parameter

    bytes := make([]byte, 16)

    if _, err := rand.Read(bytes); err != nil {

        return "", fmt.Errorf("failed to generate random bytes: %w", err)

    }

    return base64.RawURLEncoding.EncodeToString(bytes), nil

}

```

Storage Interface Definitions (`internal/storage/interfaces.go`):

```
package storage

import (
    "context"
    "time"
)

// Client represents an OAuth2 client application

type Client struct {

    ClientID      string      `json:"client_id"`
    ClientSecret   string      `json:"-"`          // Never serialize
    SecretHash     string      `json:"-"`          // Store bcrypt hash
    Name          string      `json:"name"`
    RedirectURIs []string    `json:"redirect_uris"`
    GrantTypes    []string    `json:"grant_types"`
    Scopes        []string    `json:"scopes"`
    CreatedAt     time.Time   `json:"created_at"`

}

// AuthorizationCode represents a temporary authorization code

type AuthorizationCode struct {

    Code          string      `json:"code"`
    ClientID      string      `json:"client_id"`
    UserID        string      `json:"user_id"`
    RedirectURI   string      `json:"redirect_uri"`
    Scope         string      `json:"scope"`
    CodeChallenge string      `json:"code_challenge"`
    CodeChallengeMethod string   `json:"code_challenge_method"`
    ExpiresAt     time.Time   `json:"expires_at"`
    Used          bool        `json:"used"`
    CreatedAt     time.Time   `json:"created_at"`

}
```

GO

```
}

// ClientStore defines the interface for client persistence

type ClientStore interface {

    // CreateClient stores a new OAuth2 client

    CreateClient(ctx context.Context, client *Client) error


    // GetClient retrieves a client by client_id

    GetClient(ctx context.Context, clientID string) (*Client, error)


    // ValidateClientCredentials checks client_id and client_secret

    ValidateClientCredentials(ctx context.Context, clientID, clientSecret string) error


    // TODO: Add methods for client management (update, delete, list)

}

// AuthorizationCodeStore defines the interface for authorization code persistence

type AuthorizationCodeStore interface {

    // StoreAuthorizationCode saves an authorization code

    StoreAuthorizationCode(ctx context.Context, code *AuthorizationCode) error


    // GetAuthorizationCode retrieves a code and marks it as used

    GetAuthorizationCode(ctx context.Context, code string) (*AuthorizationCode, error)


    // TODO: Add cleanup method for expired codes

}

// TokenStore defines the interface for token persistence

type TokenStore interface {

    // TODO: Define methods for storing and retrieving access/refresh tokens

    // TODO: Add methods for token revocation tracking
```

```
// TODO: Add methods for token introspection data

}

// UserStore defines the interface for user data persistence

type UserStore interface {

    // TODO: Define methods for user authentication and profile data

    // TODO: Add methods for consent management

    // TODO: Add methods for scope-to-claims mapping

}
```

Core Logic Skeleton Code

Client Registration Handler (`internal/clients/handlers.go`):

```
package clients
```

GO

```
import (
    "encoding/json"
    "net/http"

    "oauth2-provider/internal/storage"
    "oauth2-provider/internal/crypto"
)
```

```
type RegistrationHandler struct {
```

```
    clientStore storage.ClientStore
    tokenGen     *crypto.TokenGenerator
}
```

```
func NewRegistrationHandler(store storage.ClientStore, tokenGen *crypto.TokenGenerator) *RegistrationHandler {
```

```
    return &RegistrationHandler{
        clientStore: store,
        tokenGen:    tokenGen,
    }
}
```

```
// RegisterClient handles OAuth2 client registration requests
```

```
func (h *RegistrationHandler) RegisterClient(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Parse client registration request from JSON body
    // TODO 2: Validate redirect URIs are proper HTTPS URLs
    // TODO 3: Generate unique client_id using crypto.TokenGenerator
    // TODO 4: Generate client_secret and hash using GenerateClientSecret
    // TODO 5: Create Client struct with generated credentials
    // TODO 6: Store client using ClientStore.CreateClient
    // TODO 7: Return client_id and client_secret in JSON response
}
```

```

// TODO 8: Handle all error cases with proper HTTP status codes

w.Header().Set("Content-Type", "application/json")

w.WriteHeader(http.StatusNotImplemented)

json.NewEncoder(w).Encode(map[string]string{"error": "not_implemented"})

}

// ValidateClient middleware for authenticating client requests

func (h *RegistrationHandler) ValidateClient(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        // TODO 1: Extract client credentials from request (Basic auth or form)

        // TODO 2: Validate client_id format and presence

        // TODO 3: Use ClientStore.ValidateClientCredentials to check auth

        // TODO 4: Set client information in request context

        // TODO 5: Call next handler if validation succeeds

        // TODO 6: Return 401 Unauthorized for invalid credentials

        // TODO 7: Use constant-time comparison to prevent timing attacks

        next.ServeHTTP(w, r)
    })
}

```

Authorization Endpoint Handler (`internal/auth/handlers.go`):

```
package auth                                     GO

import (
    "net/http"
    "net/url"

    "oauth2-provider/internal/storage"
    "oauth2-provider/internal/crypto"
)

type AuthorizationHandler struct {
    clientStore storage.ClientStore
    codeStore   storage.AuthorizationCodeStore
    tokenGen    *crypto.TokenGenerator
}

func NewAuthorizationHandler(
    clientStore storage.ClientStore,
    codeStore   storage.AuthorizationCodeStore,
    tokenGen    *crypto.TokenGenerator,
) *AuthorizationHandler {
    return &AuthorizationHandler{
        clientStore: clientStore,
        codeStore:   codeStore,
        tokenGen:    tokenGen,
    }
}

// HandleAuthorizationRequest processes OAuth2 authorization requests

func (h *AuthorizationHandler) HandleAuthorizationRequest(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Parse query parameters (client_id, redirect_uri, scope, state, etc.)
    // TODO 2: Validate client_id exists using ClientStore.GetClient
```

```

    // TODO 3: Validate redirect_uri matches registered URI exactly

    // TODO 4: Validate response_type is "code"

    // TODO 5: Validate PKCE parameters (code_challenge, code_challenge_method)

    // TODO 6: Check if user is authenticated (session/cookie)

    // TODO 7: If not authenticated, redirect to login page

    // TODO 8: Display consent screen with requested scopes

    // TODO 9: Handle user consent decision (allow/deny)

    // TODO 10: Generate authorization code using TokenGenerator

    // TODO 11: Store code with client/user binding using AuthorizationCodeStore

    // TODO 12: Redirect back to client with code and state parameters

    // TODO 13: Handle all error cases with proper OAuth2 error responses

    http.Error(w, "Authorization endpoint not implemented", http.StatusNotImplemented)

}

// HandleConsentDecision processes user consent form submissions

func (h *AuthorizationHandler) HandleConsentDecision(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Parse form data for user consent decision (allow/deny)

    // TODO 2: Retrieve authorization request from session

    // TODO 3: If denied, redirect with access_denied error

    // TODO 4: If approved, generate and store authorization code

    // TODO 5: Redirect to client with authorization code

    // TODO 6: Store consent decision for future requests

    http.Error(w, "Consent handling not implemented", http.StatusNotImplemented)

}

```

Language-Specific Implementation Hints

Go-Specific Security Practices:

- Use `crypto/rand` instead of `math/rand` for all security-sensitive random generation
- Always check errors from cryptographic operations - they indicate serious problems

- Use `crypto/subtle.ConstantTimeCompare()` for token and password comparisons to prevent timing attacks
- Prefer `golang.org/x/crypto/bcrypt` over custom password hashing implementations
- Use `context.Context` with timeouts for all database and HTTP operations

HTTP Server Configuration:

```
server := &http.Server{  
  
    Addr:      ":8080",  
  
    Handler:   router,  
  
    ReadTimeout: 30 * time.Second, // Prevent slow-read attacks  
  
    WriteTimeout: 30 * time.Second, // Prevent slow-write attacks  
  
    IdleTimeout: 60 * time.Second, // Close idle connections  
  
    TLSConfig: &tls.Config{  
  
        MinVersion: tls.VersionTLS12, // Require TLS 1.2+  
  
    },  
  
}
```

GO

Database Connection Pooling:

```
db, err := sql.Open("sqlite3", "oauth2.db?_foreign_keys=on&_journal_mode=WAL")  
  
if err != nil {  
  
    return err  
  
}  
  
db.SetMaxOpenConns(10)          // Limit concurrent connections  
  
db.SetMaxIdleConns(5)          // Keep some connections open  
  
db.SetConnMaxLifetime(time.Hour) // Rotate connections
```

GO

JWT Configuration Best Practices:

- Use RS256 (RSA-SHA256) for access tokens to enable distributed validation
- Set appropriate token lifetimes: 15-60 minutes for access tokens, 30 days for refresh tokens
- Include only necessary claims in JWTs - they're not encrypted, only signed
- Use the `jti` (JWT ID) claim for revocation tracking

Milestone Checkpoint: Architecture Setup

After implementing the basic project structure and infrastructure components, verify the following:

File Structure Verification:

```
# Check that all required directories exist

find . -type d -name "internal" -o -name "cmd" -o -name "pkg" | wc -l

# Should return 3

# Verify component directories are created

ls internal/

# Should show: auth, clients, config, crypto, storage, token, etc.
```

BASH

Configuration Loading Test:

```
# Set environment variables and test config loading

export PORT=8080

export DB_URL=./test.db

go run cmd/server/main.go --dry-run

# Should start without errors and log configuration values
```

BASH

Cryptographic Operations Test:

```
# Test random token generation

go test ./internal/crypto/ -v

# Should pass all tests for secure random generation
```

BASH

Storage Interface Compilation:

```
# Verify all interfaces compile correctly

go build ./internal/storage/

# Should compile without errors
```

BASH

Basic HTTP Server Test:

```
# Start server and test basic connectivity

go run cmd/server/main.go &

curl -i http://localhost:8080/health

# Should return 200 OK with basic health check response
```

BASH

This architectural foundation provides the structure and infrastructure needed to implement the OAuth2 provider components in the subsequent milestones, with clear separation of concerns and proper security foundations.

Data Model

Milestone(s): This section establishes the foundational data structures for all milestones (1-4), providing the schema that will be implemented incrementally





Mental Model: Library Card System

Think of the OAuth2 data model as a sophisticated library card system. When you register for a library card, the library creates a record with your personal information and what sections you're allowed to access. The library issues you a physical card (client credentials) that proves you're registered. When you want to check out a book, you present your card to authorize the transaction, and the librarian gives you a temporary checkout slip (authorization code) that you can exchange at the front desk for an actual book loan record (access token). The library keeps track of all your current loans (active tokens) and your borrowing history (consent records). If you lose your card or want to return all your books early, the library can revoke your access immediately.

In OAuth2, registered client applications are like library cardholders, authorization codes are like temporary checkout slips, access tokens are like active loan records, and the consent system tracks what data access permissions users have granted to each application. Just as a library needs to track cards, loans, and patron permissions, our OAuth2 provider needs robust data structures to manage clients, tokens, users, and consent decisions securely.

The data model serves as the foundation for all OAuth2 operations. Every authorization request, token exchange, and user consent decision depends on these core structures. Getting the data model right is critical because security vulnerabilities often stem from incomplete data validation, missing relationships between entities, or improper lifecycle management of sensitive credentials.

OAuth Client Registration

OAuth2 **client registration** establishes the foundation of trust between the authorization server and third-party applications. When an application wants to use OAuth2, it must first register with the authorization server to obtain credentials and declare its intended behavior. This registration process creates a **client record** that defines the application's identity, capabilities, and security constraints.

The client registration data model captures both the public metadata that identifies the application and the confidential credentials used for authentication. This dual nature reflects OAuth2's support for different client types - public clients like mobile apps that cannot securely store secrets, and confidential clients like server-side web applications that can protect their credentials.

Client Entity Structure

The `Client` entity represents a registered OAuth2 application with all the metadata necessary for secure authorization flows:

Field Name	Type	Description
ClientID	string	Unique public identifier for the OAuth2 application, generated as a cryptographically random 32-character string
ClientSecret	string	Original plaintext secret returned only during registration, used for client authentication
SecretHash	string	Bcrypt hash of the client secret stored permanently, used for credential verification
Name	string	Human-readable application name displayed on consent screens and administrative interfaces
RedirectURIs	<code>[]string</code>	Whitelist of exact URIs where authorization codes can be delivered, prevents authorization code interception
GrantTypes	<code>[]string</code>	OAuth2 grant types this client is authorized to use (<code>authorization_code</code> , <code>client_credentials</code> , <code>refresh_token</code>)
Scopes	<code>[]string</code>	Maximum scopes this client can request, acts as an upper bound for authorization requests
CreatedAt	<code>time.Time</code>	Timestamp when the client was registered, used for auditing and lifecycle management

The `ClientID` serves as the public identifier for the application and is included in all OAuth2 flows. It must be cryptographically random and globally unique to prevent client impersonation attacks. The 32-character length provides sufficient entropy while remaining manageable for developers who need to configure it in their applications.

The separation between `ClientSecret` and `SecretHash` follows security best practices for credential storage. The plaintext `ClientSecret` is returned only once during registration and should be stored securely by the client application.

The authorization server stores only the bcrypt `SecretHash`, which provides one-way verification without exposing the original secret even to system administrators.

The `RedirectURIs` array enforces strict validation of where authorization codes can be delivered. Each URI must match exactly - no wildcards, partial matches, or subdomain matching is allowed. This prevents authorization code interception attacks where malicious actors register similar domains to capture codes intended for legitimate applications.

Client Types and Security Profiles

OAuth2 distinguishes between **confidential clients** that can securely store credentials and **public clients** that cannot. This distinction affects how the authorization server validates client authentication and enforces security measures:

Client Type	Authentication Method	PKCE Requirement	Secret Storage
Confidential	Client ID + Secret	Recommended	Server-side secure storage
Public	Client ID only	Mandatory	No secret issued
Native Mobile	Client ID only	Mandatory	App cannot store secrets securely
Single Page App	Client ID only	Mandatory	Browser cannot store secrets securely

Confidential clients receive both a `ClientID` and `ClientSecret` during registration. They must authenticate using both credentials when exchanging authorization codes for tokens. Examples include traditional web applications with server-side code that can store the secret in environment variables or secure configuration files.

Public clients receive only a `ClientID` and cannot authenticate with a secret. To maintain security, they must use **PKCE (Proof Key for Code Exchange)** for all authorization requests. The `CodeChallenge` and `CodeChallengeMethod` fields in authorization flows provide cryptographic proof that the same client that initiated the authorization flow is the one exchanging the code for tokens.

Security Principle: Defense in Depth

Client registration implements multiple layers of security validation. Even if one check fails (like redirect URI validation), other mechanisms (like PKCE verification for public clients) prevent successful attacks. This layered approach ensures that single vulnerabilities don't compromise the entire system.

Architecture Decision Records

Decision: Bcrypt for Client Secret Hashing

- Context:** Client secrets need secure storage that prevents recovery even by system administrators while allowing verification during authentication
- Options Considered:**
 - Store plaintext secrets for simple comparison
 - Use SHA-256 hash for fast verification
 - Use bcrypt with configurable work factor
- Decision:** Bcrypt with work factor 12
- Rationale:** Bcrypt provides adaptive cost that can increase over time as hardware improves. The work factor of 12 balances security (resists brute force) with performance (sub-100ms verification). Unlike SHA-256, bcrypt includes salt automatically and resists rainbow table attacks.
- Consequences:** Slightly slower client authentication (acceptable for OAuth2 flows), but dramatically improved security against credential database breaches. Work factor can be increased in future deployments without changing the verification logic.

Option	Pros	Cons
Plaintext	Simple implementation, fast verification	Catastrophic security risk if database compromised
SHA-256	Fast hashing and verification	Vulnerable to rainbow tables, no adaptive cost
Bcrypt	Salt included, adaptive cost, industry standard	Slower than plain hashing, requires bcrypt library

Decision: Exact String Matching for Redirect URIs

- Context:** Redirect URI validation must prevent authorization code interception while allowing legitimate client deployments
- Options Considered:**
 - Exact string matching only
 - Domain-based matching (allow subdomains)
 - Regex pattern matching for flexibility
- Decision:** Exact string matching with no wildcards or patterns
- Rationale:** OAuth2 security depends critically on preventing redirect URI manipulation. Even sophisticated pattern matching has edge cases that attackers exploit. The principle of least privilege suggests allowing only exactly what's needed. Exact matching eliminates entire classes of bypass attacks.
- Consequences:** Clients must register separate URIs for different environments (dev, staging, production), but this actually improves security by making each deployment explicit. Slightly more complex client configuration but eliminates redirect URI attack surface.

Option	Pros	Cons
Exact matching	No bypass vulnerabilities, clear security boundary	Requires separate URIs per environment
Domain matching	Flexible for subdomains	Complex parsing, potential bypass via subdomain takeover
Regex patterns	Maximum flexibility	High complexity, regex injection risks

Token Data Structures

OAuth2 tokens represent different stages of the authorization process, each with specific security properties and lifecycle requirements. The token data model encompasses **authorization codes** (temporary credentials for token exchange), **access tokens** (credentials for accessing protected resources), and **refresh tokens** (credentials for obtaining new access tokens). Each token type has distinct expiration policies, validation requirements, and security constraints.

The token lifecycle follows a progression from broad permissions to specific capabilities. Authorization codes represent a user's consent to grant access but cannot directly access resources. Access tokens represent specific, time-limited permissions to access particular resources. Refresh tokens represent long-term authorization to obtain new access tokens without re-prompting the user for consent.

Authorization Code Structure

Authorization codes are **temporary credentials** issued after successful user authentication and consent. They serve as proof that a user has authorized a specific client to access specific scopes. The code must be exchanged quickly for tokens and can only be used once to prevent replay attacks:

Field Name	Type	Description
Code	string	Cryptographically random 43-character string using URL-safe base64 encoding
ClientID	string	Client that initiated the authorization request, must match during token exchange
UserID	string	User who granted authorization, becomes the 'sub' claim in issued access tokens
RedirectURI	string	Exact redirect URI from the authorization request, must match during token exchange
Scope	string	Space-delimited scopes the user authorized, may be subset of requested scopes
CodeChallenge	string	Base64URL-encoded SHA256 hash of the client's PKCE code verifier
CodeChallengeMethod	string	Method used to generate challenge, always "S256" for SHA256 hashing
ExpiresAt	time.Time	Absolute expiration time, typically 10 minutes from creation
Used	bool	Prevents replay attacks - set to true after successful token exchange
CreatedAt	time.Time	Creation timestamp for auditing and cleanup of expired codes

The 43-character `Code` provides 256 bits of entropy when base64-encoded, making brute force attacks computationally infeasible. The code binds together all the authorization parameters (`ClientID`, `RedirectURI`, `Scope`) so that the token endpoint can verify that the exchange request matches exactly what the user authorized.

The PKCE fields (`CodeChallenge`, `CodeChallengeMethod`) provide cryptographic proof that the same client that initiated the authorization flow is the one exchanging the code. This prevents authorization code interception attacks, particularly important for mobile applications where redirect URIs might be intercepted by malicious apps.

Critical Security Property: Single Use Enforcement

Once an authorization code is exchanged for tokens, it must never be usable again. The `Used` flag provides this guarantee. If an already-used code is presented again, it indicates either a replay attack or that the code was intercepted. In either case, all tokens issued from that authorization code should be revoked immediately.

Access Token Structure (JWT Format)

Access tokens are implemented as **JSON Web Tokens (JWTs)** containing claims about the authorized access. JWTs provide stateless verification - resource servers can validate tokens cryptographically without calling back to the authorization server:

Claim Name	Type	Description
iss (Issuer)	string	Authorization server identifier, typically the server's base URL
sub (Subject)	string	User identifier from the authorization, uniquely identifies the resource owner
aud (Audience)	string	Intended recipient of the token, typically the resource server's identifier
exp (Expiration)	int64	Unix timestamp when token expires, typically 15-60 minutes from issuance
iat (Issued At)	int64	Unix timestamp when token was issued, prevents token pre-dating attacks
jti (JWT ID)	string	Unique token identifier, enables revocation tracking and prevents replay
scope	string	Space-delimited permissions granted to the client for this user
client_id	string	Client identifier, allows resource servers to log which app accessed data

The JWT structure follows the standard format: `header.payload.signature`. The header specifies the signing algorithm (typically RS256 or ES256 for asymmetric signatures). The payload contains the claims listed above as base64-encoded JSON. The signature provides cryptographic proof that the token was issued by the authorization server and hasn't been tampered with.

The `exp` claim enforces token expiration, limiting the blast radius if tokens are compromised. Short expiration times (15-60 minutes) balance security with user experience - users don't need to re-authenticate frequently, but compromised tokens become useless quickly.

The `jti` claim enables token revocation even though JWTs are stateless. When a token is revoked, the authorization server adds the `jti` to a revocation list. Resource servers can check this list during token validation to reject revoked tokens immediately.

Refresh Token Structure

Refresh tokens enable **long-term access** without requiring users to re-authenticate repeatedly. They're stored on the authorization server and used to obtain new access tokens when the current ones expire:

Field Name	Type	Description
Token	string	Cryptographically random 43-character string using URL-safe base64 encoding
ClientID	string	Client authorized to use this refresh token, validated during refresh requests
UserID	string	User who originally granted authorization, maintains consistent identity
Scope	string	Authorized scopes that can be refreshed, may be subset of original grant
ExpiresAt	time.Time	Long-term expiration, typically 30-90 days from creation
TokenFamily	string	Refresh token family identifier for rotation tracking and breach detection
ParentToken	string	Previous refresh token in the rotation chain, null for initial token
Revoked	bool	Revocation status - once true, token and entire family become invalid
CreatedAt	time.Time	Creation timestamp for lifecycle management and auditing
LastUsedAt	time.Time	Most recent usage for detecting suspicious patterns and inactive cleanup

Refresh tokens use **token rotation** for enhanced security. Each time a refresh token is used to obtain new access tokens, a new refresh token is issued and the old one is invalidated. This creates a "token family" chain that enables breach detection - if an old refresh token is used after a new one has been issued, it indicates the token family has been compromised and all tokens in the family should be revoked.

The `TokenFamily` identifier groups related refresh tokens together. When token theft is detected (old token reused), the entire family can be revoked instantly, preventing further abuse. The `ParentToken` field creates the chain linkage that enables tracing the rotation history.

Security Insight: Token Family Breach Detection

Token rotation creates a security property where legitimate usage forms a linear chain (each token used exactly once), but theft creates a branch (old token reused after new token issued). Detecting these branches enables automatic revocation of compromised token families before significant damage occurs.

Token Lifecycle State Machine

Tokens progress through defined states that determine their validity and usability:

Current State	Event	Next State	Actions Taken
Issued	Normal usage	Active	Update LastUsedAt timestamp
Active	Expiration time reached	Expired	Token becomes invalid
Active	Explicit revocation	Revoked	Add to revocation list
Active	Refresh (for refresh tokens)	Rotated	Issue new token, invalidate current
Issued/Active	System breach detection	Revoked	Revoke token family
Expired	Cleanup process	Deleted	Remove from storage
Revoked	Cleanup process (after expiry)	Deleted	Remove from revocation list

This state machine ensures that tokens have clear validity rules and that state transitions maintain security invariants. The cleanup transitions prevent the database from growing indefinitely with expired tokens while maintaining revocation lists long enough to catch late-arriving requests with revoked tokens.

User and Consent Models

The user and consent data models capture the **resource owner** information and track their authorization decisions over time. In OAuth2, the resource owner (typically a human user) controls access to their protected data and makes consent decisions about which applications can access what information. The data model must support both identity management for authentication and fine-grained consent tracking for authorization.

User consent represents the cornerstone of OAuth2's security model. Users must explicitly authorize each application's access to their data, and they retain control over these permissions throughout the application lifecycle. The consent system must track not just current permissions but also the history of authorization decisions to support revocation and auditing.

User Profile Structure

The user profile stores identity information and authentication credentials for resource owners:

Field Name	Type	Description
UserID	string	Unique user identifier, becomes 'sub' claim in tokens
Username	string	Human-readable username for authentication
Email	string	Email address for contact and identity verification
EmailVerified	bool	Whether email address has been verified through confirmation process
PasswordHash	string	Bcrypt hash of user password for authentication
GivenName	string	User's first name for OIDC standard claims
FamilyName	string	User's last name for OIDC standard claims
Picture	string	URL to user's profile picture for OIDC claims
Locale	string	User's preferred language/locale for OIDC claims
CreatedAt	time.Time	Account creation timestamp
UpdatedAt	time.Time	Last profile modification timestamp
Active	bool	Whether account is enabled and can authenticate

The user profile follows **OpenID Connect standard claims** to ensure compatibility with OIDC clients. The `UserID` serves as the stable identifier that appears in the `sub` claim of access tokens and ID tokens. It should never change even if other profile fields are updated.

The email verification status is critical for security flows like password reset and account recovery. Unverified emails should not be used for sensitive operations, and the verification status should be included in OIDC claims when email scope is granted.

Consent Record Structure

Consent records track user authorization decisions for each client application:

Field Name	Type	Description
UserID	string	User who granted the consent
ClientID	string	Client application that received the consent
Scope	string	Space-delimited scopes the user authorized
GrantedAt	time.Time	When the user originally granted consent
LastUsedAt	time.Time	Most recent authorization using this consent
ExpiresAt	time.Time	Optional consent expiration for time-limited grants
Revoked	bool	Whether user has revoked this consent
RevokedAt	time.Time	When consent was revoked (null if still active)

The consent record enables **consent reuse** - if a user has previously authorized a client for specific scopes, future authorization requests for the same or subset of scopes can skip the consent screen. This improves user experience while maintaining security through explicit authorization tracking.

Scope handling in consent records follows the **principle of least privilege**. If a client requests scopes beyond what the user previously consented to, a new consent screen must be presented for the additional scopes. The system stores the exact scopes granted, not just the fact that consent was given.

Scope to Claims Mapping

The scope-to-claims mapping defines which user profile fields are accessible for each OAuth2 scope:

Scope	Claims Included	Description
openid	sub	Required for OIDC, provides user identifier only
profile	sub, given_name, family_name, picture, locale	Basic profile information
email	email, email_verified	Email address and verification status
phone	phone_number, phone_number_verified	Phone contact information
address	address	Formatted postal address information

This mapping ensures that **scope-based access control** is enforced consistently. The UserInfo endpoint uses this mapping to filter the response based on the scopes granted in the access token. Only claims corresponding to granted scopes are included in the response.

The mapping supports **incremental authorization** where applications can request additional scopes over time. Each new scope request triggers a consent screen showing only the new permissions being requested, with previously granted scopes noted as already authorized.

Privacy Principle: Minimal Disclosure

Users should understand exactly what information they're sharing and with whom. The scope-to-claims mapping provides granular control so users can authorize access to their email without also sharing their profile picture, or share basic profile information without revealing their phone number.

Architecture Decision Records

Decision: Separate User Authentication from OAuth Authorization

- **Context:** Users need to authenticate to the OAuth provider before they can authorize third-party applications
- **Options Considered:**
 - Combine user authentication with OAuth flows in single endpoint
 - Separate authentication system with session management
 - Delegate authentication to external identity provider
- **Decision:** Separate authentication system with secure session cookies
- **Rationale:** OAuth2 specification assumes user is already authenticated when they reach the authorization endpoint. Separating concerns allows different authentication methods (password, MFA, SSO) without affecting OAuth flows. Session-based authentication provides better UX than re-authenticating for each OAuth request.
- **Consequences:** Additional complexity in session management, but cleaner separation of authentication and authorization concerns. Enables future integration with enterprise SSO without changing OAuth implementation.

Option	Pros	Cons
Combined flow	Simpler implementation	Tight coupling, harder to extend authentication methods
Separate with sessions	Clean separation, flexible auth methods	Session management complexity
External delegation	Leverage existing systems	External dependency, integration complexity

Decision: Consent Expiration and Revocation Model

- **Context:** User consent should not last forever, and users need ability to revoke previously granted permissions
- **Options Considered:**
 - Permanent consent until explicitly revoked
 - Time-limited consent with configurable expiration
 - Consent tied to token lifetime (expires with refresh token)
- **Decision:** Optional time-limited consent with user-controlled revocation
- **Rationale:** Different use cases need different consent durability. High-security applications might want consent to expire annually, while trusted applications might receive long-term consent. User revocation must always be available regardless of expiration policy.
- **Consequences:** More complex consent management, but provides flexibility for different security requirements. Requires user interface for managing active consents and clear communication about consent duration.

Option	Pros	Cons
Permanent consent	Simple implementation	No way to expire stale permissions
Time-limited consent	Automatic permission cleanup	May interrupt user workflows
Token-tied expiration	Aligned with token lifecycle	Complex interaction with refresh token rotation

Common Pitfalls

⚠ Pitfall: Storing Plaintext Client Secrets

Many implementations store client secrets in plaintext for simplicity, but this creates catastrophic security risks if the database is compromised. Even system administrators shouldn't be able to recover client secrets.

Why it's wrong: If an attacker gains database access, they can impersonate any registered client and potentially access all user data. Regulatory compliance frameworks often require credential hashing as a basic security control.

How to fix: Always hash client secrets using bcrypt with a work factor of at least 10. Store only the hash and compare submitted secrets against the hash during authentication. Return the plaintext secret only once during registration.

⚠ Pitfall: Flexible Redirect URI Matching

Implementing substring matching, wildcard matching, or regex patterns for redirect URI validation seems developer-friendly but creates security vulnerabilities that attackers regularly exploit.

Why it's wrong: Even sophisticated pattern matching has edge cases. Substring matching allows `evil.com/callback` to match `good.com/callback`. Wildcard matching can be bypassed through subdomain takeover attacks. These bypasses enable authorization code interception.

How to fix: Implement exact string matching only. Clients must register separate URIs for each environment (development, staging, production). This improves security by making each deployment explicit and eliminates entire classes of redirect URI manipulation attacks.

⚠ Pitfall: Long-Lived Authorization Codes

Setting authorization code expiration to 30 minutes or longer seems reasonable for user experience, but significantly increases the attack window for code interception and replay attacks.

Why it's wrong: Authorization codes can be intercepted through HTTP referrer headers, browser history, server logs, or man-in-the-middle attacks. The longer the code remains valid, the more time attackers have to exploit it.

How to fix: Set authorization code expiration to 10 minutes maximum, preferably 5 minutes. Modern applications can complete the code exchange quickly. Use PKCE for all clients to provide additional protection against code interception.

⚠ Pitfall: Missing Token Family Revocation

Implementing refresh token rotation without family tracking allows compromised tokens to continue working even after legitimate users obtain new tokens.

Why it's wrong: If a refresh token is stolen but the user continues using the application, both the legitimate user and the attacker will obtain valid refresh tokens. Without family tracking, there's no way to detect this compromise and revoke the stolen tokens.

How to fix: Implement token families with breach detection. When an old refresh token is used after a new one has been issued, revoke the entire token family immediately. This automatically stops attacks when they're detected.

⚠ Pitfall: JWT Payload Security Misconceptions

Storing sensitive information like passwords, social security numbers, or private keys in JWT payload because "JWTs are secure tokens."

Why it's wrong: JWT payload is base64-encoded, not encrypted. Anyone who obtains the token can decode the payload and read all claims. JWTs provide integrity (through signatures) but not confidentiality.

How to fix: Include only non-sensitive claims in JWTs: user ID, scopes, expiration times, issuer information. Store sensitive user data in your database and access it using the user ID from the token's `sub` claim.

Pitfall: Overly Broad Scope Grants

Granting all requested scopes to applications without considering the principle of least privilege or allowing scope escalation in refresh flows.

Why it's wrong: Applications might request more permissions than they actually need, either due to over-engineering or malicious intent. Once granted, these excessive permissions increase the impact of potential security breaches.

How to fix: Implement scope validation that checks requested scopes against the client's registered maximum scopes. Display clear descriptions of what each scope allows on consent screens. Never allow refresh token flows to grant scopes beyond what was originally authorized.

Implementation Guidance

This subsection provides concrete implementation patterns and starter code to translate the data model design into working Go code. The focus is on secure defaults, proper validation, and maintainable data structures that can evolve with your OAuth2 provider's needs.

Technology Recommendations

Component	Simple Option	Advanced Option
Database	SQLite with GORM	PostgreSQL with pgx driver
Password Hashing	bcrypt from golang.org/x/crypto	Argon2id for new implementations
ID Generation	crypto/rand with base64 encoding	UUID v4 with github.com/google/uuid
Time Handling	time.Time with UTC normalization	Database timestamps with timezone
Validation	Manual validation functions	github.com/go-playground/validator
Configuration	Environment variables	Structured config with github.com/spf13/viper

Recommended File Structure

```
project-root/
├── cmd/
│   └── oauth-server/
│       └── main.go           ← Server entry point
├── internal/
│   ├── models/              ← Data model definitions (this section)
│   │   ├── client.go        ← Client registration structures
│   │   ├── token.go         ← Token and authorization code structures
│   │   ├── user.go          ← User and consent structures
│   │   └── validation.go    ← Data validation functions
│   ├── storage/             ← Database interfaces and implementations
│   │   ├── interfaces.go    ← Storage interface definitions
│   │   ├── sqlite.go        ← SQLite implementation
│   │   └── migrations/      ← Database schema migrations
│   ├── auth/                ← Authentication and authorization logic
│   │   ├── client.go        ← Client credential validation
│   │   ├── password.go      ← Password hashing utilities
│   │   └── tokens.go        ← Token generation and validation
│   └── config/              ← Configuration management
       └── config.go
└── go.mod
```

Core Data Structure Implementation

```
package models

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
    "time"
    "golang.org/x/crypto/bcrypt"
)

// Client represents a registered OAuth2 application

type Client struct {

    ClientID      string      `json:"client_id" gorm:"primaryKey"`
    ClientSecret  string      `json:"client_secret,omitempty" gorm:"-"` // Never persisted
    SecretHash    string      `json:"-" gorm:"column:secret_hash"`      // Bcrypt hash
    Name          string      `json:"name" gorm:"not null"`
    RedirectURIs []string    `json:"redirect_uris" gorm:"serializer:json"`
    GrantTypes    []string    `json:"grant_types" gorm:"serializer:json"`
    Scopes        []string    `json:"scopes" gorm:"serializer:json"`
    CreatedAt     time.Time   `json:"created_at" gorm:"autoCreateTime"`
}

// AuthorizationCode represents a temporary code for token exchange

type AuthorizationCode struct {

    Code          string      `json:"code" gorm:"primaryKey"`
    ClientID     string      `json:"client_id" gorm:"not null;index"`
    UserID        string      `json:"user_id" gorm:"not null;index"`
    RedirectURI   string      `json:"redirect_uri" gorm:"not null"`
    Scope         string      `json:"scope"`
    CodeChallenge string      `json:"code_challenge"`
}
```

GO

```

CodeChallengeMethod string     `json:"code_challenge_method"`

ExpiresAt           time.Time `json:"expires_at" gorm:"not null;index"`

Used                bool      `json:"used" gorm:"default:false"`

CreatedAt           time.Time `json:"created_at" gorm:"autoCreateTime"`

}

// RefreshToken represents a long-lived token for obtaining new access tokens

type RefreshToken struct {

    Token        string      `json:"token" gorm:"primaryKey"`

    ClientID    string      `json:"client_id" gorm:"not null;index"`

    UserID       string      `json:"user_id" gorm:"not null;index"`

    Scope        string      `json:"scope"`

    ExpiresAt   time.Time   `json:"expires_at" gorm:"not null;index"`

    TokenFamily string      `json:"token_family" gorm:"not null;index"`

    ParentToken *string    `json:"parent_token,omitempty"`

    Revoked      bool       `json:"revoked" gorm:"default:false"`

    CreatedAt   time.Time   `json:"created_at" gorm:"autoCreateTime"`

    LastUsedAt  *time.Time  `json:"last_used_at,omitempty"`

}

// User represents a resource owner in the OAuth2 system

type User struct {

    UserID        string      `json:"user_id" gorm:"primaryKey"`

    Username     string      `json:"username" gorm:"uniqueIndex;not null"`

    Email         string      `json:"email" gorm:"uniqueIndex;not null"`

    EmailVerified bool      `json:"email_verified" gorm:"default:false"`

    PasswordHash string     `json:"-" gorm:"not null"`

    GivenName    string      `json:"given_name"`

    FamilyName   string      `json:"family_name"`

    Picture      string      `json:"picture"`
}

```

```
Locale      string     `json:"locale" gorm:"default:'en'"`  
  
CreatedAt   time.Time `json:"created_at" gorm:"autoCreateTime"`  
  
UpdatedAt   time.Time `json:"updated_at" gorm:"autoUpdateTime"`  
  
Active      bool       `json:"active" gorm:"default:true"`  
  
}  
  
// ConsentRecord tracks user authorization decisions for clients  
  
type ConsentRecord struct {  
  
    UserID      string     `json:"user_id" gorm:"primaryKey"`  
  
    ClientID   string     `json:"client_id" gorm:"primaryKey"`  
  
    Scope       string     `json:"scope" gorm:"not null"`  
  
    GrantedAt  time.Time `json:"granted_at" gorm:"autoCreateTime"`  
  
    LastUsedAt *time.Time `json:"last_used_at,omitempty"`  
  
    ExpiresAt  *time.Time `json:"expires_at,omitempty"`  
  
    Revoked    bool       `json:"revoked" gorm:"default:false"`  
  
    RevokedAt  *time.Time `json:"revoked_at,omitempty"`  
  
}
```

Token Generation Utilities

```
package auth

import (
    "crypto/rand"
    "crypto/sha256"
    "encoding/base64"
    "fmt"
    "time"
    "golang.org/x/crypto/bcrypt"
)

// TokenGenerator provides secure random generation methods

type TokenGenerator struct {
    // Configuration could be added here for different token formats
}

// GenerateClientSecret creates a client secret and its bcrypt hash

func GenerateClientSecret() (secret, hash string, err error) {
    // TODO: Generate 32 bytes of cryptographically random data
    // TODO: Base64 encode the random bytes to create the secret
    // TODO: Hash the secret using bcrypt with cost 12
    // TODO: Return both the plaintext secret and the hash
    // Hint: Use crypto/rand.Read() and bcrypt.GenerateFromPassword()
}

// GenerateAuthorizationCode creates a 32-byte random authorization code

func GenerateAuthorizationCode() (string, error) {
    // TODO: Generate 32 bytes of cryptographically random data
    // TODO: Encode using base64 URL encoding (no padding)
    // TODO: Return the encoded string
    // Hint: Use base64.RawURLEncoding.EncodeToString()
```

GO

```
}

// GenerateRefreshToken creates a 32-byte random refresh token

func GenerateRefreshToken() (string, error) {

    // TODO: Generate 32 bytes of cryptographically random data

    // TODO: Encode using base64 URL encoding (no padding)

    // TODO: Return the encoded string

    // Implementation should be identical to GenerateAuthorizationCode()

}

// GenerateTokenFamily creates a unique identifier for refresh token families

func GenerateTokenFamily() (string, error) {

    // TODO: Generate 16 bytes of cryptographically random data

    // TODO: Encode using base64 URL encoding (no padding)

    // TODO: Return the encoded string

    // Hint: Shorter than tokens since it's just an identifier

}

// VerifyClientSecret compares a plaintext secret against a bcrypt hash

func VerifyClientSecret(secret, hash string) bool {

    // TODO: Use bcrypt.CompareHashAndPassword to verify the secret

    // TODO: Return true if the secret matches, false otherwise

    // Hint: bcrypt.CompareHashAndPassword returns nil for success

}

// VerifyCodeChallenge validates a PKCE code verifier against its challenge

func VerifyCodeChallenge(verifier, challenge string) bool {

    // TODO: SHA256 hash the verifier string

    // TODO: Base64 URL encode the hash (no padding)

    // TODO: Compare the result with the stored challenge

    // Hint: Use crypto/sha256 and base64.RawURLEncoding
```

}

Data Validation Functions

```
package models

import (
    "fmt"
    "net/url"
    "strings"
    "time"
)

// ValidateClient checks client registration data for security requirements

func (c *Client) Validate() error {

    // TODO: Verify ClientID is not empty and has sufficient length

    // TODO: Verify Name is not empty and reasonable length

    // TODO: Validate each RedirectURI is a valid absolute URL

    // TODO: Ensure at least one redirect URI is provided

    // TODO: Validate GrantTypes contains only supported values

    // TODO: Validate Scopes contains only supported scope values

    // Hint: Use net/url.Parse() for URI validation

}

// ValidateAuthorizationRequest checks OAuth2 authorization parameters

func ValidateAuthorizationRequest(req *AuthorizationRequest, client *Client) error {

    // TODO: Verify ResponseType is "code" (only supported type)

    // TODO: Verify RedirectURI exactly matches one of client's registered URIs

    // TODO: Verify Scope is subset of client's allowed scopes

    // TODO: Verify State parameter is present (CSRF protection)

    // TODO: For public clients, verify PKCE parameters are present

    // Hint: Use strings.Fields() to split space-delimited scopes

}

// IsExpired checks if an authorization code has expired
```

GO

```
func (ac *AuthorizationCode) IsExpired() bool {
    return time.Now().UTC().After(ac.ExpiresAt)
}

// IsExpired checks if a refresh token has expired

func (rt *RefreshToken) IsExpired() bool {
    return time.Now().UTC().After(rt.ExpiresAt)
}

// HasScope checks if a scope string contains a specific scope

func HasScope(scopeString, targetScope string) bool {
    // TODO: Split scopeString by spaces
    // TODO: Check if targetScope exists in the split scopes
    // TODO: Return true if found, false otherwise
    // Hint: Space-delimited scope checking is case-sensitive
}
```

Database Interface Definition

```
package storage

import (
    "context"
    "your-project/internal/models"
)

// ClientStore interface for client persistence

type ClientStore interface {

    CreateClient(ctx context.Context, client *models.Client) error
    GetClient(ctx context.Context, clientID string) (*models.Client, error)
    UpdateClient(ctx context.Context, client *models.Client) error
    DeleteClient(ctx context.Context, clientID string) error
}

// AuthCodeStore interface for authorization code persistence

type AuthCodeStore interface {

    CreateAuthCode(ctx context.Context, code *models.AuthorizationCode) error
    GetAuthCode(ctx context.Context, code string) (*models.AuthorizationCode, error)
    MarkAuthCodeUsed(ctx context.Context, code string) error
    CleanupExpiredAuthCodes(ctx context.Context) error
}

// RefreshTokenStore interface for refresh token persistence

type RefreshTokenStore interface {

    CreateRefreshToken(ctx context.Context, token *models.RefreshToken) error
    GetRefreshToken(ctx context.Context, token string) (*models.RefreshToken, error)
    RevokeRefreshToken(ctx context.Context, token string) error
    RevokeTokenFamily(ctx context.Context, family string) error
    CleanupExpiredTokens(ctx context.Context) error
}
```

GO

```
// UserStore interface for user persistence

type UserStore interface {

    CreateUser(ctx context.Context, user *models.User) error

    GetUser(ctx context.Context, userID string) (*models.User, error)

    GetUserByUsername(ctx context.Context, username string) (*models.User, error)

    UpdateUser(ctx context.Context, user *models.User) error

}

// ConsentStore interface for consent persistence

type ConsentStore interface {

    CreateConsent(ctx context.Context, consent *models.ConsentRecord) error

    GetConsent(ctx context.Context, userID, clientID string) (*models.ConsentRecord, error)

    UpdateConsent(ctx context.Context, consent *models.ConsentRecord) error

    RevokeConsent(ctx context.Context, userID, clientID string) error

    ListUserConsents(ctx context.Context, userID string) ([]*models.ConsentRecord, error)

}
```

Configuration Management

```
package config

import (
    "fmt"
    "os"
    "strconv"
    "time"
)

// Config holds all configuration for the OAuth2 provider

type Config struct {

    Server    ServerConfig    `json:"server"`
    Database DatabaseConfig   `json:"database"`
    JWT       JWTConfig       `json:"jwt"`
    OAuth     OAuth2Config    `json:"oauth"`
}

type ServerConfig struct {

    Port int    `json:"port"`
    Host string `json:"host"`
}

type DatabaseConfig struct {

    Driver    string `json:"driver"`
    Connection string `json:"connection"`
}

type JWTConfig struct {

    Issuer    string    `json:"issuer"`
    PrivateKey string    `json:"private_key"`
    PublicKey  string    `json:"public_key"`
}
```

GO

```

    Algorithm string      `json:"algorithm"`

    ExpiryTime time.Duration `json:"expiry_time"`

}

type OAuth2Config struct {

    AuthCodeExpiry     time.Duration `json:"auth_code_expiry"`

    RefreshTokenExpiry time.Duration `json:"refresh_token_expiry"`

    RequirePKCE        bool         `json:"require_pkce"`

}

// Load reads configuration from environment variables

func Load() (*Config, error) {

    // TODO: Read SERVER_PORT environment variable, default to 8080

    // TODO: Read DATABASE_URL environment variable, default to SQLite

    // TODO: Read JWT_ISSUER environment variable, required

    // TODO: Read JWT_PRIVATE_KEY_PATH environment variable, required

    // TODO: Read AUTH_CODE_EXPIRY environment variable, default to 10 minutes

    // TODO: Read REFRESH_TOKEN_EXPIRY environment variable, default to 30 days

    // TODO: Validate that all required configuration is present

    // Hint: Use strconv.Atoi() for integer conversion, time.ParseDuration() for durations

}

```

Milestone Checkpoints

After implementing the data model structures:

1. **Compile Check:** Run `go build ./internal/models/...` - all structures should compile without errors
2. **Validation Test:** Create a simple test that validates client registration data
3. **Generation Test:** Test the token generation functions produce different outputs on each call
4. **Hash Verification:** Test that client secret hashing and verification work correctly

Expected behavior:

- `GenerateClientSecret()` should return different secrets each time
- `VerifyClientSecret()` should return true for correct secrets, false for incorrect ones
- `GenerateAuthorizationCode()` should produce 43-character URL-safe strings
- Client validation should reject invalid redirect URIs and empty required fields

Signs something is wrong:

- Token generation functions return the same value multiple times (entropy issue)
- Client secret verification always returns false (hashing mismatch)
- Validation functions accept clearly invalid data (validation logic incomplete)

Client Registration Component

Milestone(s): Milestone 1 - Client Registration & Authorization Endpoint

Mental Model: Business License System

Think of OAuth2 client registration like a business licensing system at your local city hall. When a new restaurant wants to operate, they can't just start serving customers - they must first register with the city, provide their business details, and receive official credentials that prove their legitimacy. The city issues them a business license number (like a `client_id`) and a confidential tax ID (like a `client_secret`) that they'll use for all official transactions.

Just as the city maintains strict records of where each restaurant is allowed to operate (their permitted addresses), OAuth2 client registration maintains strict records of where each application is allowed to redirect users after authorization (their registered `redirect_uri` values). A restaurant with a downtown license can't suddenly start operating in the suburbs without updating their registration - similarly, an OAuth2 client can't redirect to unregistered URIs without updating their client record.

The registration authority (whether city hall or OAuth2 provider) acts as the trusted source of truth. When other parties need to verify a business's legitimacy (like suppliers checking credentials), they contact the registration authority. Similarly, when your OAuth2 provider receives an authorization request, it validates the client's identity and permissions against the authoritative registration records.

This mental model helps explain why client registration is foundational - without proper registration and credential management, there's no way to distinguish legitimate applications from malicious ones attempting to steal user data or impersonate trusted services.

Registration Interface

The client registration interface provides the foundational API for OAuth2 applications to establish their identity with the authorization server. This interface serves as the entry point for all client applications, whether they're confidential clients (backend web applications with secure secret storage) or public clients (mobile apps and single-page applications that cannot securely store secrets).

The registration process follows a structured workflow that ensures security while remaining accessible to developers. Applications provide essential metadata about their identity, intended usage patterns, and technical capabilities. The OAuth2 provider validates this information, generates cryptographically secure credentials, and establishes the trust relationship that enables all subsequent OAuth2 flows.

Method	Parameters	Returns	Description
RegisterClient	store ClientStore , name string , redirectURIs []string	*Client , error	Creates new OAuth2 client with generated credentials and validates redirect URIs
CreateClient	ctx context.Context , client *Client	error	Persists client record to storage with proper indexing
GetClient	ctx context.Context , clientID string	*Client , error	Retrieves client by client_id for validation during flows
UpdateClient	ctx context.Context , clientID string , updates *ClientUpdate	error	Modifies existing client configuration (name, redirectURIs, scopes)
DeleteClient	ctx context.Context , clientID string	error	Soft deletes client and revokes all associated tokens

The `Client` data structure captures all essential information about a registered OAuth2 application:

Field	Type	Description
ClientID	string	Unique public identifier, 32-byte cryptographically random string
ClientSecret	string	Confidential credential for client authentication (empty for public clients)
SecretHash	string	bcrypt hash of client secret for secure storage and validation
Name	string	Human-readable application name displayed on consent screens
RedirectURIs	[]string	Exact URIs where authorization codes can be delivered
GrantTypes	[]string	Permitted OAuth2 flows (authorization_code, client_credentials, refresh_token)
Scopes	[]string	Maximum scopes this client can request (subset enforcement)
CreatedAt	time.Time	Registration timestamp for auditing and lifecycle management

The registration workflow follows these detailed steps:

- 1. Request Validation:** The registration handler validates that required fields are present and properly formatted. Application names must be non-empty and reasonable length (1-100 characters). Redirect URIs must be absolute HTTPS URLs (HTTP allowed only for localhost development). The handler rejects registrations with obviously malicious or misconfigured parameters.
- 2. Credential Generation:** The system generates a cryptographically secure `client_id` using 32 bytes of random data, encoded as a URL-safe base64 string. For confidential clients, it generates an equally strong `client_secret` and immediately hashes it using bcrypt with a work factor of 12. Public clients (mobile and single-page applications) receive no secret, as they cannot store credentials securely.
- 3. URI Validation:** Each provided redirect URI undergoes strict validation. The system requires HTTPS schemes for production (with localhost HTTP allowed for development). It rejects URIs with fragments, as OAuth2 specifications

prohibit fragment-based redirects. The validation ensures URIs are absolute, properly formatted, and don't contain suspicious patterns that might enable redirect attacks.

4. **Default Configuration:** The registration process applies sensible defaults for grant types and scopes based on client type. Confidential clients receive `authorization_code`, `refresh_token`, and optionally `client_credentials` grants. Public clients receive only `authorization_code` with PKCE enforcement. Default scopes include basic profile access, with additional scopes requiring explicit approval.
5. **Storage and Indexing:** The system persists the complete client record with proper database indexing on `ClientID` for fast lookups during authorization flows. It stores the bcrypt hash rather than plaintext secrets, ensuring credential security even if storage is compromised. The storage layer supports atomic operations to prevent partial registrations.
6. **Response Generation:** Successful registrations return the complete client record including the generated `client_id` and `client_secret` (for confidential clients). This is the only time the plaintext secret is revealed - clients must store it securely as it cannot be retrieved later, only regenerated through a separate rotation process.

The registration interface includes comprehensive error handling for common failure scenarios:

Error Condition	Response	Description
Missing application name	<code>invalid_request</code>	Application name is required for consent screen display
Invalid redirect URI	<code>invalid_redirect_uri</code>	URI must be absolute HTTPS (or localhost HTTP)
Duplicate client name	<code>invalid_request</code>	Application names must be unique within the provider
Storage failure	<code>server_error</code>	Database unavailable or constraint violation
Rate limit exceeded	<code>temporarily_unavailable</code>	Too many registrations from same source

Credential Storage and Hashing

Credential storage represents one of the most security-critical aspects of the OAuth2 provider, as compromised client credentials can enable widespread attacks against user data. The storage system must protect client secrets with the same rigor applied to user passwords, while supporting the high-throughput access patterns required during OAuth2 flows.

The credential storage architecture employs multiple layers of protection to ensure client secret security. At the core lies a cryptographically sound hashing strategy that renders stored secrets useless to attackers even in the event of database compromise. The system combines this with secure random generation, constant-time validation, and comprehensive audit logging.

Secret Generation and Hashing Process:

The `GenerateClientSecret` function implements a secure two-phase process that creates both the plaintext secret returned to the client and the hash stored in the database:

1. **Random Generation:** The system generates 32 bytes of cryptographically random data using the operating system's secure random number generator. This provides approximately 256 bits of entropy, far exceeding the security requirements for client authentication. The raw bytes are encoded using URL-safe base64 to create a string suitable for HTTP transmission and storage in configuration files.

- bcrypt Hashing:** The plaintext secret immediately undergoes bcrypt hashing with a work factor of 12. bcrypt provides several critical security properties: it's computationally expensive (requiring ~250ms on modern hardware), includes automatic salt generation to prevent rainbow table attacks, and uses a proven algorithm resistant to both brute-force and cryptanalytic attacks. The work factor of 12 represents a careful balance between security and performance - high enough to significantly slow brute-force attempts while remaining fast enough for interactive authentication flows.
- Secure Cleanup:** After hashing, the system immediately clears the plaintext secret from memory and returns both the original secret (for client storage) and the hash (for database storage). This minimizes the window during which plaintext secrets exist in memory.

Storage Schema and Indexing:

The client credentials storage employs a carefully designed schema that supports both security and performance requirements:

Field	Type	Index	Description
<code>client_id</code>	VARCHAR(64)	PRIMARY KEY	Unique identifier, base64-encoded random bytes
<code>secret_hash</code>	VARCHAR(72)	None	bcrypt hash of client secret (60 chars + NULL)
<code>name</code>	VARCHAR(255)	UNIQUE	Application name for consent screens
<code>redirect_uris</code>	JSON	None	Array of permitted redirect destinations
<code>grant_types</code>	JSON	None	Array of permitted OAuth2 flows
<code>scopes</code>	JSON	None	Array of maximum permitted scopes
<code>created_at</code>	TIMESTAMP	INDEX	Registration time for auditing
<code>last_used_at</code>	TIMESTAMP	INDEX	Most recent authentication for cleanup
<code>active</code>	BOOLEAN	INDEX	Soft deletion flag

The indexing strategy optimizes for the OAuth2 flow access patterns. The primary key on `client_id` enables O(1) lookups during authorization and token requests. The unique index on `name` prevents duplicate application names that could confuse users. Indexes on timestamp fields support efficient cleanup of unused clients and audit queries.

Secret Validation Process:

The `VerifyClientSecret` function implements constant-time secret validation that prevents timing attacks while maintaining compatibility with standard bcrypt libraries:

- Hash Retrieval:** The system retrieves the stored bcrypt hash for the provided `client_id`. If no client exists, it performs a dummy bcrypt operation against a fake hash to maintain constant timing behavior.
- bcrypt Comparison:** The bcrypt library's constant-time compare function validates the provided secret against the stored hash. This comparison takes consistent time regardless of how much of the secret matches, preventing attackers from using timing differences to guess secrets character by character.
- Result Handling:** The function returns a boolean result without leaking information about why validation failed. Whether the client doesn't exist, the secret is wrong, or the client is disabled, the result appears identical to potential attackers.

Audit Logging and Monitoring:

The credential storage system maintains comprehensive audit logs that enable security monitoring and forensic analysis:

Event Type	Logged Data	Purpose
Client Registration	<code>client_id</code> , <code>name</code> , source IP, timestamp	Track new client creation
Authentication Success	<code>client_id</code> , source IP, timestamp	Monitor normal usage
Authentication Failure	attempted <code>client_id</code> , source IP, timestamp	Detect brute-force attacks
Secret Rotation	<code>client_id</code> , old/new secret hashes, timestamp	Track credential changes
Client Deletion	<code>client_id</code> , <code>name</code> , deletion reason, timestamp	Audit client lifecycle

The monitoring system analyzes these logs to detect suspicious patterns such as repeated authentication failures from the same IP address, authentication attempts for non-existent clients, or unusual geographic patterns in client usage.

Backup and Recovery Considerations:

Client credential storage requires special handling in backup and recovery scenarios. Database backups containing client records must be encrypted at rest and protected with the same security measures as production systems. The recovery process must validate that restored client hashes remain valid and haven't been corrupted during backup or restoration.

For disaster recovery scenarios, the system supports client secret rotation workflows that allow legitimate clients to re-establish their credentials without requiring manual intervention from every application developer. This involves secure out-of-band communication channels and cryptographic proof of client ownership.

Architecture Decision Records

The client registration component requires several critical design decisions that significantly impact the security, performance, and usability of the entire OAuth2 provider. Each decision involves careful trade-offs between competing concerns.

Decision: Client Secret Hashing Algorithm

- Context:** Client secrets must be stored securely to prevent credential theft in case of database compromise, while supporting high-throughput authentication during OAuth2 flows. The hashing algorithm must balance computational cost, security strength, and implementation complexity.
- Options Considered:** Plain storage (no hashing), SHA-256 with salt, bcrypt, scrypt, Argon2id
- Decision:** Use bcrypt with work factor 12 for client secret hashing
- Rationale:** bcrypt provides proven security with widespread library support and excellent resistance to both rainbow table and brute-force attacks. Work factor 12 requires ~250ms per hash on modern hardware, adding meaningful cost for attackers while remaining acceptable for interactive OAuth2 flows (which typically involve 1-2 client authentications per user session). Unlike memory-hard functions (scrypt/Argon2id), bcrypt's CPU-only approach suits deployment environments with constrained memory.
- Consequences:** Enables secure credential storage with acceptable performance overhead. Requires careful work factor tuning for different deployment environments. Provides upgrade path to stronger algorithms in the future through algorithm versioning.

Option	Pros	Cons
Plain storage	Zero computational overhead	Catastrophic security failure if database compromised
SHA-256 + salt	Fast, simple implementation	Vulnerable to GPU-accelerated brute force attacks
bcrypt (chosen)	Proven security, tunable cost, wide support	Fixed memory usage, slower than simple hashes
scrypt	Memory-hard, resistant to ASIC attacks	Complex tuning, higher memory requirements
Argon2id	Latest standard, excellent security properties	Limited library support, complex parameterization

Decision: Client ID Generation Strategy

- **Context:** Client IDs must be unique, unpredictable, and suitable for use in URLs and HTTP headers. They serve as public identifiers that appear in authorization URLs and audit logs, so they must not leak sensitive information about the client or provider.
- **Options Considered:** Sequential integers, UUIDs (v4), cryptographically random strings, hash of client metadata
- **Decision:** Use 32-byte cryptographically random strings, base64url-encoded
- **Rationale:** Cryptographic randomness provides maximum security against enumeration and prediction attacks. 32 bytes offers 256 bits of entropy, making collisions mathematically impossible at any realistic scale. Base64url encoding creates URL-safe strings without padding characters. This approach avoids UUID overhead while providing stronger security guarantees than standard UUID v4 (which uses only 122 bits of randomness).
- **Consequences:** Eliminates enumeration attacks and ensures global uniqueness without coordination. Creates slightly longer identifiers than UUIDs (43 vs 36 characters) but provides superior security. Requires cryptographically secure random number generator.

Option	Pros	Cons
Sequential integers	Short, simple, predictable ordering	Trivial enumeration, leaks registration volume
UUIDs v4	Standard format, good uniqueness	Only 122 bits entropy, longer than needed
Random strings (chosen)	Maximum entropy, customizable length	Non-standard format, requires careful encoding
Hashed metadata	Deterministic, prevents duplicates	Vulnerable to rainbow tables, complex collision handling

Decision: Redirect URI Validation Strictness

- **Context:** OAuth2 redirect URI validation prevents authorization code interception attacks, but overly strict validation can break legitimate use cases. The validation must balance security against developer usability, particularly for development and testing scenarios.
- **Options Considered:** Exact string matching only, prefix matching, pattern matching with wildcards, runtime URI validation
- **Decision:** Exact string matching with localhost HTTP exception for development
- **Rationale:** Exact matching provides the strongest security guarantee by preventing any form of redirect manipulation. The OAuth2 specification strongly recommends exact matching to prevent subtle attack vectors. Allowing HTTP for localhost URIs supports local development without compromising production security. This approach eliminates entire classes of redirect-based attacks while remaining practical for legitimate development workflows.
- **Consequences:** Maximizes security against redirect attacks but requires clients to register all redirect URIs upfront. Development environments can use HTTP localhost without security compromise. Clients needing dynamic redirects must implement application-level routing after the OAuth2 redirect.

Option	Pros	Cons
Exact matching (chosen)	Maximum security, spec-compliant	Requires registering all URIs, less flexible
Prefix matching	More flexible for path-based routing	Vulnerable to path traversal attacks
Wildcard patterns	Very flexible, supports complex routing	Complex implementation, many attack vectors
Runtime validation	Ultimate flexibility	Cannot validate at registration time, complex security model

Decision: Public Client Secret Handling

- **Context:** Public clients (mobile apps, single-page applications) cannot securely store client secrets, but the OAuth2 specification allows both secret-based and secret-less authentication. The provider must handle both client types while maintaining security.
- **Options Considered:** Require secrets for all clients, allow empty secrets for public clients, separate registration endpoints, client type declaration
- **Decision:** Allow empty client secrets with mandatory PKCE for public clients
- **Rationale:** Following OAuth2 Security Best Current Practice, public clients should not receive secrets and must use PKCE for security. Empty secrets clearly indicate public client status and eliminate any temptation to embed secrets in client-side code. Mandatory PKCE provides equivalent security to client secrets for public clients. This approach aligns with current OAuth2 security guidance and modern application architectures.
- **Consequences:** Enables secure mobile and SPA integration without compromising security. Requires PKCE implementation and validation. Creates two distinct client authentication paths that must be properly handled throughout the system.

Option	Pros	Cons
Require all secrets	Simple, uniform authentication model	Impossible for public clients to implement securely
Empty secrets (chosen)	Clear public client indication, spec-compliant	Requires dual authentication logic
Separate endpoints	Clear separation of client types	Duplicated registration logic, more complex API
Client type declaration	Explicit client capability declaration	Additional complexity, potential misconfiguration

Decision: Client Name Uniqueness Enforcement

- **Context:** Client names appear on consent screens to help users identify applications requesting access. Duplicate names could enable phishing attacks where malicious applications impersonate legitimate ones. However, legitimate applications might have naming conflicts.
- **Options Considered:** No uniqueness requirement, global uniqueness enforcement, namespace-based uniqueness, user confirmation for duplicates
- **Decision:** Enforce global uniqueness of client names with case-insensitive comparison
- **Rationale:** Global uniqueness prevents impersonation attacks where malicious clients use names identical to popular applications. Case-insensitive comparison prevents near-duplicate names that could confuse users. This approach prioritizes user security over developer convenience, as confused users might grant inappropriate permissions to impersonator applications. The burden on developers to choose unique names is manageable and encourages distinctive branding.
- **Consequences:** Prevents client impersonation attacks through name confusion. May require developers to modify desired application names if conflicts exist. Requires careful handling of internationalization and Unicode normalization in name comparison.

Option	Pros	Cons
No uniqueness	No naming restrictions for developers	Enables impersonation attacks
Global uniqueness (chosen)	Prevents impersonation, clear user identification	May inconvenience legitimate developers
Namespaced uniqueness	Reduces conflicts while preventing impersonation	Complex namespace management
User confirmation	Flexible with explicit user warning	Users may ignore warnings, complex UX

Common Pitfalls

OAuth2 client registration appears straightforward but contains numerous subtle security and implementation traps that can compromise the entire authorization system. These pitfalls often arise from misunderstanding the threat model or making seemingly reasonable optimizations that introduce vulnerabilities.

⚠ Pitfall: Storing Plaintext Client Secrets

Many implementations store client secrets in plaintext or with reversible encryption, reasoning that the secrets need to be compared during authentication. This creates catastrophic vulnerability if the database is compromised, as attackers gain immediate access to all client credentials.

Why it's wrong: Client secrets function like passwords and should never be recoverable. Unlike user passwords, client secrets don't need account recovery workflows - they can be regenerated if lost. Storing plaintext secrets means a single database breach exposes every registered application's credentials.

How to fix it: Always hash client secrets using bcrypt, scrypt, or Argon2id before storage. Store only the hash and compare provided secrets against the hash using the library's constant-time comparison function. Never implement custom hashing or comparison logic.

Pitfall: Weak Client ID Generation

Using predictable client ID generation (sequential numbers, timestamps, or weak random generators) enables enumeration attacks where adversaries can guess valid client IDs and attempt to exploit them in authorization flows.

Why it's wrong: Client IDs appear in authorization URLs and logs, making them observable to attackers. Predictable IDs allow attackers to discover all registered clients and potentially launch targeted attacks against specific applications. Even UUID v4 provides insufficient randomness (122 bits) for high-security environments.

How to fix it: Generate client IDs using cryptographically secure random number generators with at least 256 bits of entropy. Use base64url encoding to create URL-safe strings. Never use sequential numbers, timestamps, or application names as client ID components.

Pitfall: Loose Redirect URI Validation

Implementing flexible redirect URI validation (prefix matching, wildcard support, or runtime validation) seems developer-friendly but introduces authorization code interception vulnerabilities where attackers can redirect authorization codes to attacker-controlled endpoints.

Why it's wrong: The redirect URI is the primary security boundary preventing authorization code theft. Any flexibility in validation creates attack opportunities. Even seemingly safe approaches like prefix matching can be exploited through path traversal or subdomain attacks.

How to fix it: Implement exact string matching for redirect URIs. Allow HTTP only for localhost addresses during development. Require applications to register all possible redirect URIs at registration time. Document this limitation clearly and provide guidance on application-level routing after OAuth2 redirects.

Pitfall: Inconsistent Client Type Handling

Mixing confidential and public client authentication logic without clear separation leads to security vulnerabilities where public clients might authenticate using missing or weak credentials, or confidential clients might bypass proper authentication.

Why it's wrong: Confidential and public clients have fundamentally different security models. Confidential clients rely on secret-based authentication, while public clients depend on PKCE. Inconsistent handling can allow authentication bypasses or force insecure authentication methods.

How to fix it: Clearly distinguish client types at registration time. Store client type explicitly in the client record. Implement separate authentication paths that enforce appropriate security measures for each client type. Never allow public clients to authenticate with secrets, and never allow confidential clients to skip secret authentication.

Pitfall: Missing Client Secret Rotation Support

Failing to implement client secret rotation capabilities creates operational security problems when secrets are compromised, accidentally exposed, or need regular rotation for compliance requirements.

Why it's wrong: Client secrets will eventually need rotation due to security breaches, compliance requirements, or operational accidents. Without rotation support, the only option is deleting and re-creating clients, which breaks all existing deployments and requires coordination across multiple teams.

How to fix it: Implement client secret rotation endpoints that generate new secrets while temporarily accepting both old and new secrets during a transition period. Provide secure out-of-band notification mechanisms for secret rotation. Include secret version tracking to support gradual rollout of new secrets.

Pitfall: Inadequate Rate Limiting on Registration

Allowing unlimited client registration enables abuse where attackers create large numbers of clients to exhaust storage, create confusion in audit logs, or launch distributed attacks using many different client IDs.

Why it's wrong: Client registration should be a relatively rare operation, but unlimited registration enables various abuse scenarios. Mass client creation can exhaust storage quotas, make audit logs unusable, and provide attackers with numerous client IDs for distributed attack campaigns.

How to fix it: Implement rate limiting on client registration endpoints based on source IP, user account, and global registration rate. Consider requiring manual approval for client registration in high-security environments. Monitor registration patterns for abuse and implement automated detection of suspicious registration activity.

Pitfall: Exposing Internal Client Information

Returning internal database fields, sensitive configuration details, or other clients' information in registration responses creates information disclosure vulnerabilities that can aid attackers in understanding the system or launching targeted attacks.

Why it's wrong: Client registration responses should contain only the information needed by the registering application. Exposing internal IDs, configuration details, or information about other clients provides intelligence that attackers can use for reconnaissance or privilege escalation attacks.

How to fix it: Carefully design registration response objects to include only public information needed by clients. Never include database internal IDs, other clients' information, or sensitive configuration. Implement response filtering that explicitly allowlists returned fields rather than blocklisting sensitive ones.

Pitfall: Insufficient Redirect URI Format Validation

Accepting malformed URIs, URIs with fragments, or non-HTTP(S) schemes in redirect URI registration creates potential security vulnerabilities and protocol violations that can be exploited in authorization flows.

Why it's wrong: OAuth2 specifications explicitly prohibit certain URI formats (particularly those with fragments) and require absolute URIs. Accepting invalid formats can lead to protocol violations, security vulnerabilities, or interoperability problems with client applications.

How to fix it: Implement comprehensive URI validation that checks scheme (HTTPS required, HTTP only for localhost), absolute format, absence of fragments, and proper encoding. Reject registrations with invalid URIs rather than attempting to correct them. Provide clear error messages explaining validation requirements.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Framework	net/http with chi router	gin-gonic/gin or fiber
Database	SQLite with migrations	PostgreSQL with pgx driver
Password Hashing	golang.org/x/crypto/bcrypt	Custom bcrypt with configurable cost
Random Generation	crypto/rand	Custom entropy pool management
JSON Handling	encoding/json	json-iterator/go for performance
Configuration	Environment variables	Viper with multiple config sources
Logging	log/slog (Go 1.21+)	zerolog or zap for structured logging
Testing	testing package	testify for assertions

Recommended File Structure

```
internal/client/
├── client.go          ← Core client data structures and business logic
├── store.go           ← ClientStore interface and implementations
├── postgres_store.go  ← PostgreSQL storage implementation
├── sqlite_store.go    ← SQLite storage implementation (development)
├── registration.go   ← Client registration service logic
├── validation.go     ← Redirect URI and client data validation
├── handlers.go        ← HTTP handlers for registration endpoints
├── client_test.go    ← Unit tests for core logic
├── store_test.go      ← Storage implementation tests
└── integration_test.go ← End-to-end registration flow tests

cmd/server/
├── main.go            ← Application entry point
└── config.go          ← Configuration loading and validation

internal/crypto/
├── tokens.go          ← Token generation utilities (shared across components)
└── random.go          ← Cryptographic random number generation

migrations/
├── 001_create_clients.sql ← Database schema for client storage
└── 002_add_client_indexes.sql ← Performance indexes
```

Infrastructure Starter Code

Complete Database Migration (`migrations/001_create_clients.sql`):

SQL

```
-- OAuth2 client registration table

CREATE TABLE oauth_clients (
    client_id VARCHAR(64) PRIMARY KEY,
    secret_hash VARCHAR(72), -- bcrypt hash (60 chars) + safety margin
    name VARCHAR(255) UNIQUE NOT NULL,
    redirect_uris JSON NOT NULL,
    grant_types JSON NOT NULL DEFAULT '[ "authorization_code" ]',
    scopes JSON NOT NULL DEFAULT '[ "openid", "profile" ]',
    client_type VARCHAR(20) NOT NULL DEFAULT 'confidential', -- confidential or public
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_used_at TIMESTAMP,
    active BOOLEAN DEFAULT true
);

-- Indexes for performance

CREATE INDEX idx_clients_created_at ON oauth_clients(created_at);
CREATE INDEX idx_clients_last_used ON oauth_clients(last_used_at);
CREATE INDEX idx_clients_active ON oauth_clients(active);
CREATE UNIQUE INDEX idx_clients_name_case_insensitive ON oauth_clients(LOWER(name));
```

Complete Random Token Generator (internal/crypto/random.go):

```
package crypto
```

```
import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
)
```

```
// TokenGenerator provides cryptographically secure random token generation
```

```
type TokenGenerator struct{}
```

```
// GenerateClientID creates a cryptographically random client identifier
```

```
// Returns 32 bytes of randomness encoded as base64url (43 characters)
```

```
func (tg *TokenGenerator) GenerateClientID() (string, error) {
```

```
    bytes := make([]byte, 32)
```

```
    if _, err := rand.Read(bytes); err != nil {
```

```
        return "", fmt.Errorf("failed to generate random client ID: %w", err)
```

```
    }
```

```
    return base64.RawURLEncoding.EncodeToString(bytes), nil
```

```
}
```

```
// GenerateClientSecret creates a cryptographically random client secret
```

```
// Returns 32 bytes of randomness encoded as base64url (43 characters)
```

```
func (tg *TokenGenerator) GenerateClientSecret() (string, error) {
```

```
    bytes := make([]byte, 32)
```

```
    if _, err := rand.Read(bytes); err != nil {
```

```
        return "", fmt.Errorf("failed to generate random client secret: %w", err)
```

```
    }
```

```
    return base64.RawURLEncoding.EncodeToString(bytes), nil
```

```
}
```

```
// GenerateAuthorizationCode creates a cryptographically random authorization code
```

GO

```
// Returns 32 bytes of randomness encoded as base64url (43 characters)

func (tg *TokenGenerator) GenerateAuthorizationCode() (string, error) {

    bytes := make([]byte, 32)

    if _, err := rand.Read(bytes); err != nil {

        return "", fmt.Errorf("failed to generate random authorization code: %w", err)
    }

    return base64.RawURLEncoding.EncodeToString(bytes), nil
}
```

Complete Configuration Management (cmd/server/config.go):

```
package main GO

import (
    "fmt"
    "os"
    "strconv"
    "time"
)

type Config struct {
    Server    ServerConfig
    Database DatabaseConfig
    JWT       JWTConfig
    OAuth     OAuth2Config
}

type ServerConfig struct {
    Port          int
    ReadTimeout   time.Duration
    WriteTimeout  time.Duration
    IdleTimeout   time.Duration
}

type DatabaseConfig struct {
    Driver string
    DSN    string
    MaxOpenConns int
    MaxIdleConns int
}

type JWTConfig struct {
    PrivateKeyPath string
}
```

```

    PublicKeyPath  string

    Issuer         string

    Audience       string

}

type OAuth2Config struct {

    AuthorizationCodeTTL time.Duration

    AccessTokenTTL       time.Duration

    RefreshTokenTTL      time.Duration

    BCryptCost           int

}

// Load reads configuration from environment variables with sensible defaults

func Load() (*Config, error) {

    cfg := &Config{

        Server: ServerConfig{

            Port:          getEnvInt("PORT", 8080),

            ReadTimeout:  getEnvDuration("READ_TIMEOUT", 30*time.Second),

            WriteTimeout: getEnvDuration("WRITE_TIMEOUT", 30*time.Second),

            IdleTimeout:  getEnvDuration("IDLE_TIMEOUT", 60*time.Second),

        },

        Database: DatabaseConfig{

            Driver:        getEnvString("DB_DRIVER", "postgres"),

            DSN:          getEnvString("DATABASE_URL", ""),

            MaxOpenConns: getEnvInt("DB_MAX_OPEN_CONNS", 25),

            MaxIdleConns: getEnvInt("DB_MAX_IDLE_CONNS", 5),

        },

        JWT: JWTConfig{

            PrivateKeyPath: getEnvString("JWT_PRIVATE_KEY_PATH", "./keys/private.pem"),

            PublicKeyPath:  getEnvString("JWT_PUBLIC_KEY_PATH", "./keys/public.pem"),

        }

    }

    return cfg, nil
}

```

```
        Issuer:           getEnvString("JWT_ISSUER", "https://oauth.example.com"),
        Audience:         getEnvString("JWT_AUDIENCE", "https://api.example.com"),
    },
OAuth: OAuth2Config{
    AuthorizationCodeTTL: getEnvDuration("AUTHORIZATION_CODE_TTL", 10*time.Minute),
    AccessTokenTTL:       getEnvDuration("ACCESS_TOKEN_TTL", 1*time.Hour),
    RefreshTokenTTL:      getEnvDuration("REFRESH_TOKEN_TTL", 30*24*time.Hour), // 30 days
    BCryptCost:          getEnvInt("BCRYPT_COST", 12),
},
}

if cfg.Database.DSN == "" {
    return nil, fmt.Errorf("DATABASE_URL environment variable is required")
}

return cfg, nil
}

func getEnvString(key, defaultValue string) string {
    if value := os.Getenv(key); value != "" {
        return value
    }
    return defaultValue
}

func getEnvInt(key string, defaultValue int) int {
    if value := os.Getenv(key); value != "" {
        if parsed, err := strconv.Atoi(value); err == nil {
            return parsed
        }
    }
}
```

```
    return defaultValue

}

func getEnvDuration(key string, defaultValue time.Duration) time.Duration {
    if value := os.Getenv(key); value != "" {
        if parsed, err := time.ParseDuration(value); err == nil {
            return parsed
        }
    }
    return defaultValue
}
```

Core Logic Skeleton Code

Client Registration Service (internal/client/registration.go):

```
package client

import (
    "context"
    "fmt"
    "time"
    "golang.org/x/crypto/bcrypt"
    "your-project/internal/crypto"
)

type RegistrationService struct {

    store      ClientStore

    generator *crypto.TokenGenerator

    bcryptCost int
}

func NewRegistrationService(store ClientStore, bcryptCost int) *RegistrationService {
    return &RegistrationService{
        store:      store,
        generator: &crypto.TokenGenerator{},
        bcryptCost: bcryptCost,
    }
}

// RegisterClient creates a new OAuth2 client with generated credentials

func (rs *RegistrationService) RegisterClient(ctx context.Context, name string, redirectURIs []string) (*Client, error) {
    // TODO 1: Validate input parameters (name length, redirect URI format)

    // TODO 2: Check if client name already exists (case-insensitive)

    // TODO 3: Generate cryptographically random client_id

    // TODO 4: Generate client_secret and bcrypt hash (for confidential clients)

    // TODO 5: Create Client struct with generated credentials and metadata
}
```

GO

```

// TODO 6: Store client record in database with proper error handling

// TODO 7: Return complete client record including plaintext secret

// Hint: Only return plaintext secret on registration - never store it

}

// GenerateClientSecret creates a random secret and its bcrypt hash

func (rs *RegistrationService) GenerateClientSecret() (secret, hash string, err error) {

    // TODO 1: Generate cryptographically random secret using TokenGenerator

    // TODO 2: Create bcrypt hash using configured cost factor

    // TODO 3: Return both plaintext secret and hash

    // TODO 4: Handle any errors from random generation or hashing

    // Hint: Use constant-time operations and clear plaintext from memory quickly

}

// VerifyClientSecret validates a provided secret against stored hash

func (rs *RegistrationService) VerifyClientSecret(secret, hash string) bool {

    // TODO 1: Use bcrypt.CompareHashAndPassword for constant-time comparison

    // TODO 2: Return true only if secret matches hash exactly

    // TODO 3: Handle empty hashes (public clients) appropriately

    // Hint: bcrypt.CompareHashAndPassword returns error on mismatch

}

```

Client Storage Interface (internal/client/store.go):

```
package client

import (
    "context"
    "time"
)

// Client represents a registered OAuth2 application

type Client struct {

    ClientID      string      `json:"client_id" db:"client_id"`
    ClientSecret  string      `json:"client_secret,omitempty" // Only included in registration response`
    SecretHash    string      `json:"-" db:"secret_hash" // Never expose in JSON`
    Name          string      `json:"name" db:"name"`
    RedirectURIs []string    `json:"redirect_uris" db:"redirect_uris"`
    GrantTypes    []string    `json:"grant_types" db:"grant_types"`
    Scopes        []string    `json:"scopes" db:"scopes"`
    ClientType    string      `json:"client_type" db:"client_type" // "confidential" or "public"`
    CreatedAt     time.Time   `json:"created_at" db:"created_at"`
    LastUsedAt   *time.Time  `json:"last_used_at,omitempty" db:"last_used_at"`
    Active        bool        `json:"active" db:"active"`
}

// ClientStore defines the interface for client persistence

type ClientStore interface {

    // CreateClient stores a new OAuth2 client record
    CreateClient(ctx context.Context, client *Client) error

    // GetClient retrieves a client by client_id
    GetClient(ctx context.Context, clientID string) (*Client, error)

    // UpdateClient modifies an existing client's configuration
}
```

```
UpdateClient(ctx context.Context, clientID string, client *Client) error

// DeleteClient soft-deletes a client (sets active=false)

DeleteClient(ctx context.Context, clientID string) error

// ListClients returns all active clients (for admin interfaces)

ListClients(ctx context.Context, limit, offset int) ([]*Client, error)

// CheckNameExists validates that client name is unique (case-insensitive)

CheckNameExists(ctx context.Context, name string) (bool, error)

}
```

Client Validation Logic (internal/client/validation.go):

```
package client

import (
    "fmt"
    "net/url"
    "strings"
)

// ValidateClientRegistration checks all registration parameters

func ValidateClientRegistration(name string, redirectURIs []string) error {

    // TODO 1: Validate client name (length, characters, not empty)

    // TODO 2: Validate redirect URIs (format, scheme, no fragments)

    // TODO 3: Check that at least one redirect URI is provided

    // TODO 4: Ensure redirect URIs are unique within the list

    // TODO 5: Return descriptive error for any validation failure

    // Hint: Use net/url.Parse for URI validation

}

// ValidateRedirectURI ensures a redirect URI meets OAuth2 security requirements

func ValidateRedirectURI(uri string) error {

    // TODO 1: Parse URI using net/url.Parse

    // TODO 2: Validate scheme (HTTPS required, HTTP only for localhost)

    // TODO 3: Ensure URI is absolute (has host)

    // TODO 4: Check that fragment is empty (OAuth2 prohibition)

    // TODO 5: Validate host is not empty and properly formatted

    // Hint: Localhost detection: host == "localhost" or host == "127.0.0.1"

}

// IslocalhostURI determines if a URI is a localhost address for development

func IslocalhostURI(uri string) (bool, error) {

    // TODO 1: Parse URI and extract host

    // TODO 2: Check for localhost, 127.0.0.1, ::1, or [::1]
```

GO

```
// TODO 3: Handle port numbers correctly  
  
// TODO 4: Return true only for actual localhost addresses  
  
}
```

HTTP Handlers (internal/client/handlers.go):

```
package client
```

GO

```
import (  
    "encoding/json"  
    "net/http"  
    "log/slog"  
)
```

```
type ClientHandlers struct {
```

```
    registrationService *RegistrationService  
    logger             *slog.Logger  
}
```

```
func NewClientHandlers(service *RegistrationService, logger *slog.Logger) *ClientHandlers {
```

```
    return &ClientHandlers{  
        registrationService: service,  
        logger:             logger,  
    }
```

```
}
```

```
type RegisterClientRequest struct {
```

```
    Name          string `json:"name"  
    RedirectURIs []string `json:"redirect_uris"  
    GrantTypes   []string `json:"grant_types,omitempty"  
    Scopes       []string `json:"scopes,omitempty"  
}
```

```
// RegisterClient handles POST /oauth2/register requests
```

```
func (ch *ClientHandlers) RegisterClient(w http.ResponseWriter, r *http.Request) {  
    // TODO 1: Parse and validate JSON request body  
    // TODO 2: Apply default grant types and scopes if not provided  
    // TODO 3: Call RegistrationService.RegisterClient with validated data
```

```

    // TODO 4: Handle registration errors appropriately (400 vs 500)

    // TODO 5: Return JSON response with client credentials

    // TODO 6: Log registration event for auditing

    // Hint: This is the only time client_secret is returned in plaintext

}

// GetClient handles GET /oauth2/clients/{client_id} requests (for admin)

func (ch *ClientHandlers) GetClient(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract client_id from URL path

    // TODO 2: Retrieve client from store

    // TODO 3: Remove sensitive fields (secret_hash) from response

    // TODO 4: Return JSON response or 404 if not found

    // TODO 5: Log access for auditing

}

```

Language-Specific Hints

Go-Specific Implementation Tips:

- JSON Handling:** Use struct tags to control JSON serialization. The `json:"-"` tag prevents sensitive fields from appearing in responses. The `omitempty` tag excludes nil pointers and empty values.
- Database Integration:** Use the `db` struct tags with libraries like `sqlx` for automatic scanning. Store JSON arrays as PostgreSQL JSON columns or comma-separated strings for SQLite.
- Error Handling:** Wrap errors with `fmt.Errorf("context: %w", err)` to maintain error chains. Use custom error types for OAuth2-specific errors that need special HTTP status codes.
- Context Usage:** Always pass `context.Context` to database operations for timeout and cancellation support. Use `context.Background()` only in tests or main functions.
- Security Libraries:** Use `golang.org/x/crypto/bcrypt` for password hashing, never implement custom crypto. Use `crypto/rand` for random generation, never `math/rand`.
- Testing:** Use table-driven tests for validation functions. Use `testify/mock` for mocking the ClientStore interface. Use `httptest` for testing HTTP handlers.

Milestone Checkpoint

After implementing the client registration component, verify the following functionality:

Registration API Testing:

```
# Test client registration

curl -X POST http://localhost:8080/oauth2/register \
-H "Content-Type: application/json" \
-d '{
    "name": "Test Application",
    "redirect_uris": ["https://app.example.com/callback", "http://localhost:3000/callback"]
}'

# Expected response (save the client_secret!):

{
    "client_id": "abc123...",
    "client_secret": "def456...",
    "name": "Test Application",
    "redirect_uris": ["https://app.example.com/callback", "http://localhost:3000/callback"],
    "grant_types": ["authorization_code"],
    "scopes": ["openid", "profile"],
    "client_type": "confidential",
    "created_at": "2024-01-01T12:00:00Z",
    "active": true
}
```

BASH

Database Verification:

```
-- Verify client record was created correctly

SELECT client_id, name, redirect_uris, secret_hash IS NOT NULL as has_secret
FROM oauth_clients
WHERE name = 'Test Application';

-- Verify secret is hashed, not plaintext

SELECT LENGTH(secret_hash), LEFT(secret_hash, 7) = '$2a$12$' as is_bcrypt
FROM oauth_clients
WHERE name = 'Test Application';
```

SQL

Validation Testing:

```
# Test duplicate name rejection                                BASH

curl -X POST http://localhost:8080/oauth2/register \
      -H "Content-Type: application/json" \
      -d '{"name": "Test Application", "redirect_uris": ["https://other.example.com/callback"]}' \
# Should return 400 Bad Request

# Test invalid redirect URI

curl -X POST http://localhost:8080/oauth2/register \
      -H "Content-Type: application/json" \
      -d '{"name": "Invalid App", "redirect_uris": ["http://example.com/callback"]}' \
# Should return 400 Bad Request (HTTP not allowed except localhost)
```

Security Verification:

1. Confirm client secrets are never logged in plaintext
2. Verify bcrypt hashes start with `$2a$12$` (correct algorithm and cost)
3. Test that `client_id` values are truly random (no sequential patterns)
4. Ensure redirect URI validation rejects malicious URIs with fragments
5. Verify case-insensitive name uniqueness works correctly

Signs Something Is Wrong:

- Client secrets appear in logs or error messages (security leak)
- Client registration succeeds with invalid redirect URIs (validation bypass)
- Database contains plaintext secrets (hashing failure)
- Sequential or predictable `client_id` values (weak random generation)
- Duplicate client names accepted (uniqueness constraint failure)

Authorization Endpoint Component

Milestone(s): Milestone 1 - Client Registration & Authorization Endpoint

Mental Model: Bank Authorization Process

Think of the OAuth2 authorization endpoint like a bank's signature card verification process. When you want to give someone permission to access your bank account (like setting up automatic bill payments), you don't just hand over your debit card PIN. Instead, you visit the bank with proper identification, review exactly what permissions the company is requesting, and then sign an authorization form that grants specific, limited access to your account.

The authorization endpoint serves as this secure "bank branch" where the real account owner (resource owner) can review what a third-party application is requesting and make an informed decision. Just as the bank verifies your identity before processing any authorization, the OAuth2 provider must authenticate the user before showing them the consent screen. The bank also validates that the company requesting access is legitimate (registered business) and that their request form (authorization request) contains all required information and matches their registered details.

The authorization code that gets generated is like a signed authorization form with a unique reference number. This form can only be used once, expires quickly (typically within 10 minutes), and must be exchanged by the legitimate business at the bank's separate "account setup" window (token endpoint) to receive the actual account access credentials. The authorization code itself provides no account access - it's merely proof that the account owner approved the request under specific terms.

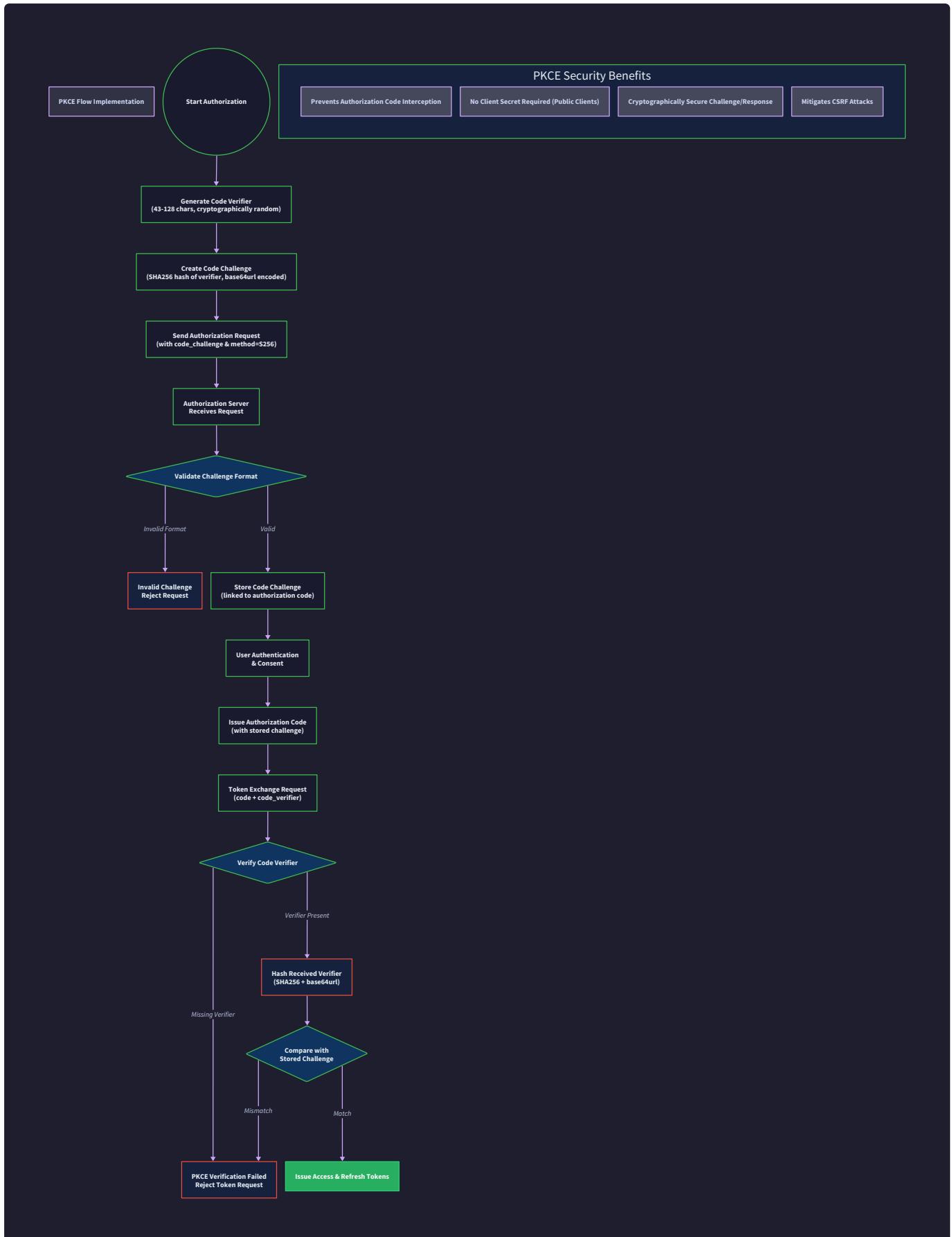
This mental model highlights several critical security properties: the authorization decision is always made by the authenticated account owner, the authorization is specific to one requesting party and their stated purpose, and the authorization proof (code) cannot be directly used to access protected resources.

PKCE Implementation

Proof Key for Code Exchange (PKCE) addresses a fundamental security weakness in the original OAuth2 authorization code flow. When mobile applications or single-page applications initiate an authorization request, they cannot securely store a client secret (since anyone can inspect the app binary or JavaScript source). Without PKCE, an attacker who intercepts an authorization code could potentially exchange it for tokens without being the legitimate client.

PKCE solves this by creating a cryptographic challenge-response mechanism that proves the client exchanging the authorization code is the same client that initiated the authorization request. The process works like a secure handshake where the client demonstrates knowledge of a secret without ever transmitting that secret over potentially insecure channels.

The **code challenge generation** begins when the client creates a cryptographically random string called a `code_verifier`. This verifier must be between 43 and 128 characters long and use unreserved characters (A-Z, a-z, 0-9, '-', '.', '_', '~'). The client then applies a one-way transformation to create a `code_challenge`. While the OAuth2 specification allows "plain" transformation (where challenge equals verifier), security best practices mandate using "S256" method where the challenge is the base64url-encoded SHA256 hash of the verifier.



During the **authorization request phase**, the client includes the `code_challenge` and `code_challenge_method` parameters in the authorization URL. The authorization server stores these values alongside the generated authorization

code. An attacker intercepting this request only sees the hashed challenge, not the original verifier, making it computationally infeasible to generate a valid verifier.

The **verification process** occurs during token exchange when the client presents the original `code_verifier` to the token endpoint. The authorization server retrieves the stored `code_challenge` and `code_challenge_method` for that authorization code, recomputes the challenge by applying the same transformation to the provided verifier, and compares the results. Only if the computed challenge matches the stored challenge does the server proceed with token issuance.

PKCE Parameter	Source	Purpose	Security Property
<code>code_verifier</code>	Client-generated	Proves client identity during token exchange	High entropy secret never transmitted during authorization
<code>code_challenge</code>	SHA256 hash of verifier	Binds authorization code to client	One-way transformation prevents verifier derivation
<code>code_challenge_method</code>	Always "S256"	Specifies hash algorithm	Ensures cryptographic binding, not plaintext

PKCE enforcement policies must distinguish between different client types. Public clients (mobile apps, SPAs) should be required to use PKCE for all authorization requests, as they cannot securely store client secrets. Confidential clients (server-side web applications) that can authenticate with client secrets may optionally use PKCE as defense-in-depth, but the authorization server should not reject requests lacking PKCE parameters from properly authenticated confidential clients.

The **code challenge validation** implementation must use constant-time comparison operations to prevent timing attacks that could leak information about the expected challenge value. Additionally, the authorization server must store the PKCE parameters securely alongside authorization codes and ensure they are properly cleaned up when codes expire or are consumed.

Critical Security Insight: PKCE transforms OAuth2 from relying solely on client authentication (which public clients cannot do securely) to cryptographic proof of request initiation, making the authorization code flow secure even for clients that cannot keep secrets.

Architecture Decision Record: PKCE Enforcement Strategy

Decision: Mandatory PKCE for Public Clients with Graceful Confidential Client Support

- **Context:** Different OAuth2 client types have different security capabilities. Public clients (mobile, SPA) cannot securely store secrets while confidential clients can authenticate via client secrets.
- **Options Considered:**
 1. Mandatory PKCE for all clients
 2. Optional PKCE for all clients
 3. Mandatory for public clients, optional for confidential clients
- **Decision:** Require PKCE for clients without client secrets, allow optional PKCE for confidential clients
- **Rationale:** Maximizes security for vulnerable public clients while maintaining compatibility with existing confidential clients that rely on client secret authentication
- **Consequences:** Requires client type detection logic and different validation paths, but provides appropriate security for each client category

Option	Pros	Cons	Security Level
Mandatory for all	Uniform security model, future-proof	Breaks existing confidential clients	Highest
Optional for all	Maximum compatibility	Public clients remain vulnerable	Lowest
Type-specific (chosen)	Appropriate security per client type	Complex validation logic	Optimal balance

User Consent Screen

The **consent screen interface** serves as the critical trust boundary where users make informed decisions about granting access to their protected resources. This interface must clearly communicate what the requesting application wants to access, who is making the request, and what the consequences of approval or denial will be. The design must balance security transparency with usability to avoid consent fatigue that leads users to blindly approve all requests.

Essential consent screen elements include the requesting application's verified name and description, a clear list of requested scopes translated into human-readable permissions, visual indicators of data sensitivity levels, and explicit approval/denial buttons that make the user's choice unambiguous. The screen must also display the user's current authentication context and provide a secure way to change accounts if needed.

The **scope presentation strategy** transforms technical OAuth2 scope strings into meaningful descriptions that non-technical users can understand. Rather than showing "user:email repo:public_repo", the screen should display "Access your email address" and "Read your public repositories". Each permission description should include specific examples of what the application will be able to do and, importantly, what it will NOT be able to do with the granted access.

Scope	Technical Name	User-Friendly Description	Data Sensitivity
Profile basic info	profile	View your name and profile picture	Low
Email address	email	Access your email address for notifications	Medium
Repository access	repo	Read and modify your code repositories	High
Admin privileges	admin:org	Manage organization settings and members	Critical

Consent persistence logic determines when to show the consent screen versus when to rely on previously granted permissions. The system should store consent decisions with sufficient granularity to detect when new permissions are requested or when significant time has passed since the last consent. Users should never be surprised by applications having access they don't remember granting.

A robust consent system maintains **consent records** that track not just the granted scopes, but also when consent was given, when it was last used, and under what circumstances. This enables features like consent expiration, suspicious activity detection, and detailed access logs that users can review to maintain control over their data.

The **consent flow state machine** manages the user journey from initial authorization request through consent decision to final redirect. The system must handle edge cases like users navigating away mid-flow, session timeouts during consent review, and attempts to manipulate consent parameters through browser developer tools.

State	User Action	System Response	Next State
Request Received	-	Validate parameters, authenticate user	Show Consent
Show Consent	Approve	Generate authorization code, redirect	Flow Complete
Show Consent	Deny	Send access_denied error, redirect	Flow Complete
Show Consent	Navigate away	Clean up pending state	Request Expired
Session Timeout	-	Require re-authentication	Authenticate User

Cross-site request forgery (CSRF) protection in the consent flow relies heavily on the OAuth2 `state` parameter, but the consent screen itself may require additional protection if implemented as an HTML form. The authorization server must validate that consent decisions originate from legitimate user interactions, not from malicious scripts or cross-site attacks.

Authorization Code Generation

Cryptographically secure random generation forms the foundation of authorization code security. Authorization codes must be indistinguishable from random data to prevent attackers from predicting or brute-forcing valid codes. The system must use a cryptographically secure pseudo-random number generator (CSPRNG) that draws entropy from the operating system's secure random source.

The **authorization code format** should prioritize security over human readability. A 32-byte random value encoded as base64url (resulting in a 43-character string) provides approximately 256 bits of entropy, making brute-force attacks computationally infeasible. The encoding must use base64url rather than standard base64 to ensure the codes are URL-safe without requiring additional percent-encoding.

Code binding mechanisms associate each authorization code with the specific authorization request parameters that generated it. This binding prevents authorization code injection attacks where an attacker attempts to use a legitimate code with different parameters than originally authorized. The system must store and validate the client ID, redirect URI, requested scopes, and PKCE challenge alongside each code.

Bound Parameter	Storage Requirement	Validation Point	Attack Prevention
Client ID	Required	Token exchange	Prevents cross-client code reuse
Redirect URI	Exact match	Token exchange	Prevents code interception
Scope	Original requested scopes	Token exchange	Prevents privilege escalation
PKCE Challenge	SHA256 hash + method	Token exchange	Proves client identity
User ID	Internal user identifier	Token exchange	Prevents user confusion

Authorization code lifecycle management enforces strict temporal and usage constraints to minimize the window of vulnerability. Codes must expire within 10 minutes of generation (RFC recommendation) and become invalid immediately after one successful exchange attempt. The system should also implement cleanup processes to purge expired codes and prevent storage exhaustion.

The **single-use enforcement** mechanism must be atomic to prevent race conditions where the same code could be exchanged multiple times simultaneously. This typically requires database-level constraints or atomic compare-and-swap operations that mark codes as used in the same transaction that validates their availability.

Authorization code storage security requires protecting codes both at rest and in transit. While codes are temporary and cannot directly access resources, they represent valuable authentication artifacts that attackers might attempt to steal. Storage should use appropriate database security measures, and codes should never appear in application logs or error messages.

Architecture Decision Records

Architecture Decision Record: State Parameter Validation Strategy

Decision: Cryptographically Secure State Generation with Round-Trip Validation

- **Context:** The OAuth2 state parameter provides CSRF protection but implementation approaches vary from simple timestamps to cryptographic tokens
- **Options Considered:**
 1. Client-provided state with no server validation
 2. Server-generated random state with session storage
 3. Hybrid approach allowing client state with server validation
- **Decision:** Accept client-provided state parameters while strongly recommending cryptographically random values and providing validation guidance
- **Rationale:** Maintains OAuth2 specification compliance while enabling flexible client implementations, with clear security guidance preventing weak state implementations
- **Consequences:** Requires client education about state security but provides maximum compatibility with diverse client architectures

Approach	CSRF Protection	Implementation Complexity	Client Flexibility	Chosen
Client-only state	Variable (depends on client)	Low	High	/
Server-generated state	High	Medium	Low	
Hybrid validation	High	High	Medium	

Architecture Decision Record: Authorization Code Expiration Strategy

Decision: Short-Lived Codes with Aggressive Cleanup

- Context:** Authorization codes represent a security risk while valid, but overly short expiration can cause usability issues with network latency
- Options Considered:**
 - 10-minute expiration (RFC 6749 recommendation)
 - 5-minute expiration for enhanced security
 - Adaptive expiration based on client risk profile
- Decision:** 10-minute expiration with 5-minute option for high-security deployments, plus immediate cleanup on first use
- Rationale:** Balances security with practical network delays and mobile application backgrounding scenarios while maintaining RFC compliance
- Consequences:** Requires robust cleanup mechanisms and clear client guidance on timely code exchange

Architecture Decision Record: Redirect URI Validation Strictness

Decision: Exact String Matching with No Wildcards or Partial Matches

- Context:** Redirect URI validation prevents authorization code interception but strict validation can cause deployment difficulties
- Options Considered:**
 - Exact string matching only
 - Allow subdomain wildcards (*.example.com)
 - Allow path prefix matching
- Decision:** Enforce exact string matching with multiple allowed URIs per client
- Rationale:** Exact matching eliminates entire classes of redirect URI manipulation attacks while multiple URI registration provides deployment flexibility
- Consequences:** Requires careful client configuration but provides strongest security guarantees against redirect attacks

Validation Level	Security Strength	Configuration Flexibility	Attack Surface	Chosen
Exact matching	Highest	Medium	Minimal	✓
Subdomain wildcards	Medium	High	Moderate	
Path prefixes	Low	Highest	Significant	

Common Pitfalls

⚠️ Pitfall: Using Predictable Authorization Code Generation

Many implementations attempt to create "readable" authorization codes by using timestamps, sequential numbers, or weak random number generators. This creates a critical vulnerability where attackers can predict or enumerate valid codes.

Why it's wrong: Authorization codes with insufficient entropy can be brute-forced or predicted, allowing attackers to obtain tokens for arbitrary users without their consent.

How to fix: Use a cryptographically secure random number generator to create codes with at least 128 bits of entropy. In Go, use `crypto/rand` rather than `math/rand`. Encode the result as base64url for URL safety.

⚠️ Pitfall: Insufficient PKCE Challenge Validation

Implementations sometimes skip PKCE validation for "trusted" clients or use string equality comparison that's vulnerable to timing attacks.

Why it's wrong: Weak PKCE validation defeats the entire purpose of the security mechanism, leaving public clients vulnerable to authorization code interception attacks.

How to fix: Always validate PKCE parameters when present, use constant-time comparison operations, and require PKCE for all public clients (those without client secrets).

⚠️ Pitfall: Storing User Passwords in Consent Context

Some implementations attempt to "streamline" the consent flow by storing user credentials or authentication state insecurely to avoid re-authentication.

Why it's wrong: Storing authentication credentials beyond the immediate authentication session creates unnecessary credential exposure risks and violates security best practices.

How to fix: Use secure session management with time-limited authentication tokens. Require re-authentication for sensitive operations rather than caching credentials.

⚠️ Pitfall: Ignoring Authorization Code Binding

Implementations might generate authorization codes without properly binding them to the original request parameters, allowing codes to be used with different clients or redirect URIs than intended.

Why it's wrong: Unbound authorization codes can be stolen and used by different clients, leading to token theft and unauthorized access.

How to fix: Store all critical request parameters (client ID, redirect URI, scope, PKCE challenge) alongside each authorization code and validate these parameters during token exchange.

⚠️ Pitfall: Exposing Authorization Codes in Logs or Error Messages

Authorization codes might inadvertently appear in application logs, error messages, or debugging output, creating an information disclosure vulnerability.

Why it's wrong: Logged authorization codes can be extracted by attackers with log access and used to obtain unauthorized tokens.

How to fix: Implement careful logging practices that redact or hash sensitive parameters. Never include authorization codes in error messages returned to clients.

⚠ Pitfall: Weak State Parameter Validation

Clients might use predictable state values (like "state=1234") or servers might not properly validate state round-trips, weakening CSRF protection.

Why it's wrong: Predictable or unvalidated state parameters allow cross-site request forgery attacks that can trick users into authorizing malicious applications.

How to fix: Generate cryptographically random state values with sufficient entropy and validate that the returned state exactly matches the sent state. Consider binding state to user sessions for additional protection.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Routing	net/http with custom mux	Gin or Echo framework
Template Engine	html/template (stdlib)	Templ or custom components
Cryptography	crypto/rand for CSPRNG	Additional JOSE libraries
Session Management	Signed JWT cookies	Redis-backed sessions
PKCE Validation	Manual SHA256 computation	Dedicated OAuth2 library

Recommended File Structure

```
internal/authorization/
  endpoint.go           ← HTTP handlers for /authorize
  consent.go            ← Consent screen logic
  pkce.go               ← PKCE challenge validation
  code_generator.go     ← Authorization code creation
  models.go              ← AuthorizationRequest, AuthorizationCode types
  templates/
    consent.html        ← User consent screen template
    error.html          ← OAuth2 error display template
  endpoint_test.go      ← Authorization endpoint tests
  pkce_test.go          ← PKCE validation tests
```

Authorization Request Handler (Core Logic Skeleton)

```
// HandleAuthorizationRequest processes OAuth2 authorization requests and redirects  
// to consent screen or returns authorization code based on existing consent.  
  
// Implements RFC 6749 Section 4.1.1 with PKCE support per RFC 7636.  
  
func (s *AuthorizationService) HandleAuthorizationRequest(w http.ResponseWriter, r *http.Request) {  
  
    // TODO 1: Parse and validate query parameters into AuthorizationRequest struct  
  
    // Required: client_id, redirect_uri, response_type, scope  
  
    // Optional: state, code_challenge, code_challenge_method  
  
  
    // TODO 2: Retrieve client configuration from ClientStore using client_id  
  
    // Return invalid_client error if client not found  
  
  
    // TODO 3: Validate redirect_uri against client's registered redirect URIs  
  
    // Must be exact string match - no wildcards or partial matches  
  
  
    // TODO 4: Validate response_type parameter - must be "code" for authorization code flow  
  
    // Return unsupported_response_type error for other values  
  
  
    // TODO 5: Validate requested scopes against client's allowed scopes  
  
    // Remove any unauthorized scopes from the request  
  
  
    // TODO 6: Validate PKCE parameters if present  
  
    // code_challenge_method must be "S256", challenge must be valid base64url  
  
    // Require PKCE for public clients (no client_secret)  
  
  
    // TODO 7: Authenticate user - redirect to login if not authenticated  
  
    // Store authorization request in session for post-login continuation  
  
  
    // TODO 8: Check for existing consent for this user, client, and scope combination  
  
    // If valid consent exists and covers all requested scopes, skip consent screen
```

```
// TODO 9: If no consent or insufficient consent, redirect to consent screen  
//           Pass authorization request context to consent handler  
  
// TODO 10: If consent already exists, generate authorization code immediately  
//           Bind code to client_id, user_id, redirect_uri, scope, and PKCE challenge  
  
// TODO 11: Redirect back to client with authorization code and state parameter  
//           URL format: redirect_uri?code=AUTH_CODE&state=ORIGINAL_STATE  
  
// Hint: Store authorization codes with 10-minute expiration  
// Hint: Use constant-time comparison for all string validations  
// Hint: Log security events but never log authorization codes or user credentials  
}
```

PKCE Challenge Validator (Complete Implementation)

```
package authorization

import (
    "crypto/sha256"
    "encoding/base64"
    "errors"
    "strings"
)

// PKCEValidator handles Proof Key for Code Exchange challenge verification

type PKCEValidator struct{}


// ValidateChallenge verifies a PKCE code_verifier against the stored code_challenge.
// Implements RFC 7636 Section 4.6 verification requirements.

func (p *PKCEValidator) ValidateChallenge(verifier, challenge, method string) error {
    if method != "S256" {
        return errors.New("unsupported code_challenge_method, must be S256")
    }

    // Verify verifier format - 43-128 characters, unreserved chars only
    if len(verifier) < 43 || len(verifier) > 128 {
        return errors.New("code_verifier length must be 43-128 characters")
    }

    for _, c := range verifier {
        if !isUnreservedChar(c) {
            return errors.New("code_verifier contains invalid characters")
        }
    }
}
```

GO

```

// Compute SHA256 of verifier

hash := sha256.Sum256([]byte(verifier))

computedChallenge := base64.RawURLEncoding.EncodeToString(hash[:])



// Constant-time comparison to prevent timing attacks

if !constantTimeEqual(computedChallenge, challenge) {

    return errors.New("code_verifier does not match code_challenge")

}

return nil
}

// GenerateChallenge creates a PKCE code_challenge from a code_verifier

func (p *PKCEValidator) GenerateChallenge(verifier string) (string, error) {

    if len(verifier) < 43 || len(verifier) > 128 {

        return "", errors.New("verifier length must be 43-128 characters")

    }

    hash := sha256.Sum256([]byte(verifier))

    return base64.RawURLEncoding.EncodeToString(hash[:]), nil
}

// isUnreservedChar checks if character is allowed in PKCE code_verifier

func isUnreservedChar(c rune) bool {

    return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') ||
           (c >= '0' && c <= '9') || c == '-' || c == '.' || c == '_' || c == '~'
}

// constantTimeEqual performs constant-time string comparison

func constantTimeEqual(a, b string) bool {

    if len(a) != len(b) {

```

```
    return false

}

result := byte(0)

for i := 0; i < len(a); i++ {

    result |= a[i] ^ b[i]

}

return result == 0
```

Consent Screen Handler (Core Logic Skeleton)

```
// HandleConsentScreen displays the OAuth2 consent interface and processes user decisions. GO

// Renders scope permissions in human-readable format and stores consent decisions.

func (s *AuthorizationService) HandleConsentScreen(w http.ResponseWriter, r *http.Request) {

    if r.Method == "GET" {

        // TODO 1: Retrieve authorization request from secure session

        //           Redirect to error page if session expired or invalid


        // TODO 2: Load client information for display (name, description, icon)

        //           Show generic "Unknown Application" if client data unavailable


        // TODO 3: Transform technical scopes into user-friendly descriptions

        //           Use scope mapping table to generate readable permission list


        // TODO 4: Render consent template with client info and permission descriptions

        //           Include CSRF token and original authorization request parameters


        // TODO 5: Provide clear approve/deny buttons with explicit consequences

        //           Show "Allow" and "Deny" with descriptions of each action's result

    }

    if r.Method == "POST" {

        // TODO 6: Validate CSRF token to prevent cross-site request forgery

        //           Return error if token missing or invalid


        // TODO 7: Parse user decision from form data (approve/deny)

        //           Handle case where user closes browser without deciding


        // TODO 8: If user denied, redirect back with access_denied error

        //           Include original state parameter in error redirect
    }
}
```

```
// TODO 9: If user approved, store consent decision in persistent storage
//           Record user_id, client_id, granted_scopes, and timestamp

// TODO 10: Generate authorization code bound to the consent decision
//           Include all original request parameters in code binding

// TODO 11: Redirect back to client with authorization code and state
//           Clean up session data after successful redirect

// Hint: Consent decisions should have configurable expiration
// Hint: Store granular consent (per-scope) for future partial consent checks
// Hint: Log consent decisions for audit and user access review

}

}
```

Authorization Code Generator (Complete Implementation)

```
package authorization

import (
    "crypto/rand"
    "encoding/base64"
    "time"
)

// CodeGenerator creates cryptographically secure authorization codes

type CodeGenerator struct {
    codeLength int // Number of random bytes (32 recommended)
    expiration time.Duration // Code lifetime (10 minutes recommended)
}

// NewCodeGenerator creates a generator with secure default settings

func NewCodeGenerator() *CodeGenerator {
    return &CodeGenerator{
        codeLength: 32, // 256 bits of entropy
        expiration: 10 * time.Minute,
    }
}

// GenerateAuthorizationCode creates a cryptographically random authorization code.

// Returns base64url-encoded code suitable for use in URLs without escaping.

func (g *CodeGenerator) GenerateAuthorizationCode() (string, error) {
    randomBytes := make([]byte, g.codeLength)

    // Use crypto/rand for cryptographically secure random generation
    _, err := rand.Read(randomBytes)
    if err != nil {
        return "", err
    }

    // Base64 encode the random bytes
    var encoded string
    if err = base64.URLEncoding.EncodeToString(randomBytes, &encoded); err != nil {
        return "", err
    }

    return encoded, nil
}
```

GO

```

}

// Base64url encoding for URL safety without percent-encoding

code := base64.RawURLEncoding.EncodeToString(randomBytes)

return code, nil
}

// CreateAuthorizationCode generates code and creates bound AuthorizationCode record

func (g *CodeGenerator) CreateAuthorizationCode(req *AuthorizationRequest, userID string)
(*AuthorizationCode, error) {

code, err := g.GenerateAuthorizationCode()

if err != nil {

return nil, err
}

return &AuthorizationCode{

Code:           code,
ClientID:       req.ClientID,
UserID:         userID,
RedirectURI:    req.RedirectURI,
Scope:          req.Scope,
CodeChallenge:  req.CodeChallenge,
CodeChallengeMethod: req.CodeChallengeMethod,
ExpiresAt:      time.Now().Add(g.expiration),
Used:           false,
CreatedAt:      time.Now(),
}, nil
}

```

Language-Specific Hints

Go-Specific Implementation Notes:

- Use `crypto/rand.Read()` for all random generation - never use `math/rand` for security-critical values
- Implement constant-time string comparison using bitwise XOR to prevent timing attacks
- Use `html/template` with auto-escaping enabled for consent screen rendering to prevent XSS
- Store authorization codes with `time.Time` expiration and use database-level TTL where possible
- Implement atomic code consumption using SQL `UPDATE ... WHERE used = false` with affected row counting
- Use structured logging with `slog` but implement custom redaction for sensitive OAuth2 parameters

Error Handling Patterns:

- Return structured OAuth2 errors using the `OAuth2Error` type with proper error codes
- Implement redirect-based error responses that preserve the original `state` parameter
- Log security events (failed validations, expired codes) but never log sensitive values
- Use HTTP status codes appropriately: 400 for client errors, 302 for OAuth2 redirects, 500 for server errors

Milestone Checkpoint

Verification Steps for Authorization Endpoint Implementation:

- 1. Client Registration Verification:** Register a test client and verify generated `client_id` format and `redirect_uri` storage
- 2. Authorization Request Processing:** Send GET request to `/authorize` with valid parameters, confirm consent screen display
- 3. PKCE Challenge Handling:** Test authorization requests with and without PKCE, verify public client enforcement
- 4. Consent Flow Testing:** Submit consent form approval/denial, verify authorization code generation and redirect behavior
- 5. Error Response Validation:** Test invalid `client_id`, mismatched `redirect_uri`, and expired authorization codes
- 6. Security Boundary Testing:** Attempt CSRF attacks on consent form, test state parameter validation, verify redirect URI exact matching

Expected Behavior After Implementation:

- Authorization requests with valid parameters show branded consent screen with readable scope descriptions
- User consent approval generates single-use authorization code that expires within 10 minutes
- PKCE-enabled requests properly validate code challenges using SHA256 verification
- Invalid requests return appropriate OAuth2 error responses with correct redirect behavior
- Security attacks (CSRF, redirect URI manipulation, code reuse) are properly rejected

Common Implementation Issues and Diagnostics:

- **Consent screen not displaying:** Check user authentication and session management
- **Authorization codes not working:** Verify code binding and single-use enforcement
- **PKCE validation failing:** Check SHA256 computation and base64url encoding
- **Redirect loops:** Verify redirect URI exact matching and state parameter handling
- **Security test failures:** Review constant-time comparisons and CSRF token validation

Token Endpoint Component

Milestone(s): Milestone 2 - Token Endpoint & JWT Generation

Mental Model: Ticket Exchange System

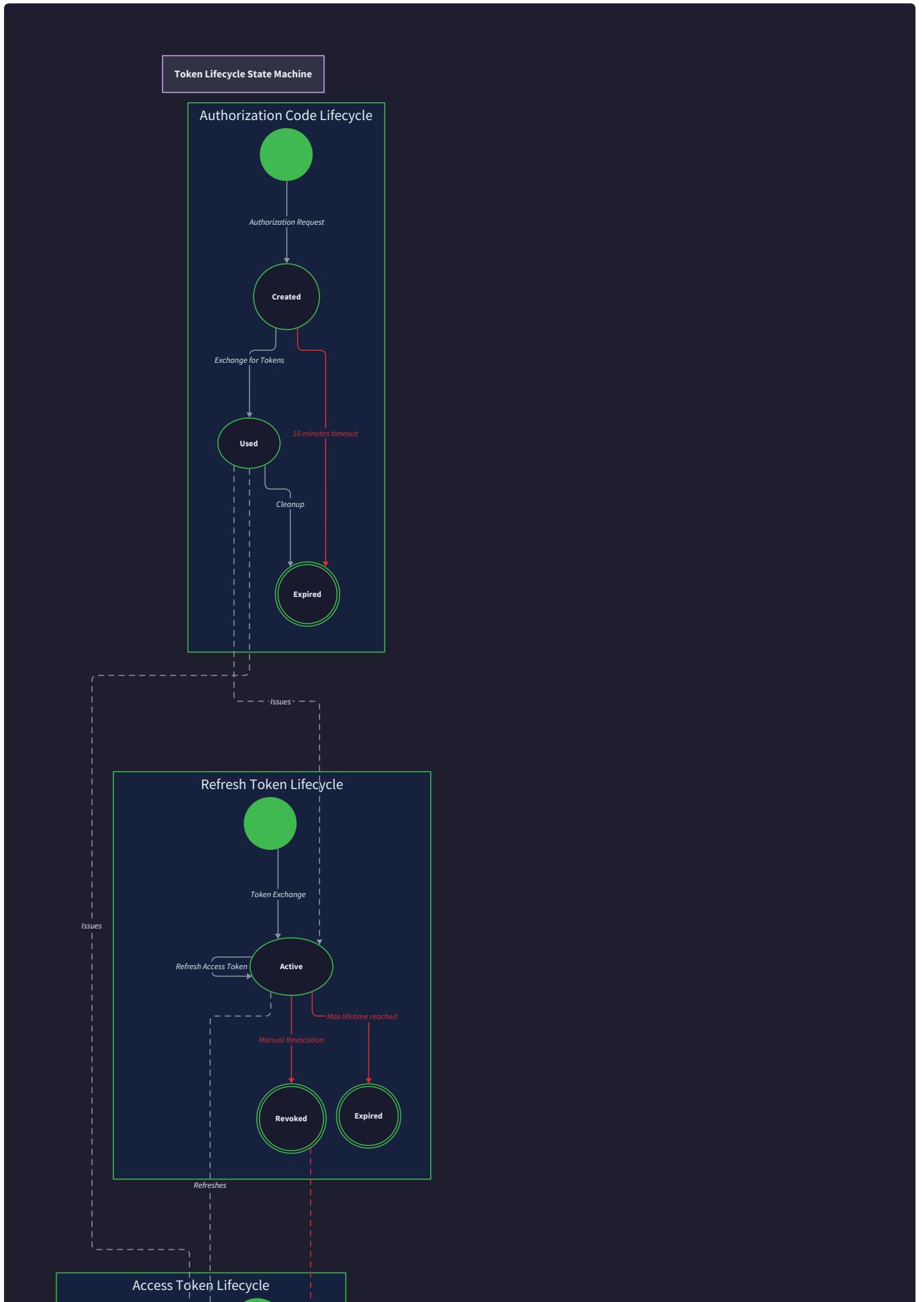
Think of the token endpoint as a sophisticated ticket exchange system at a major entertainment venue. When you arrive at a concert hall with a voucher (authorization code), you don't just walk into the show - you must first visit the box office to exchange that voucher for actual tickets that grant you specific access privileges.

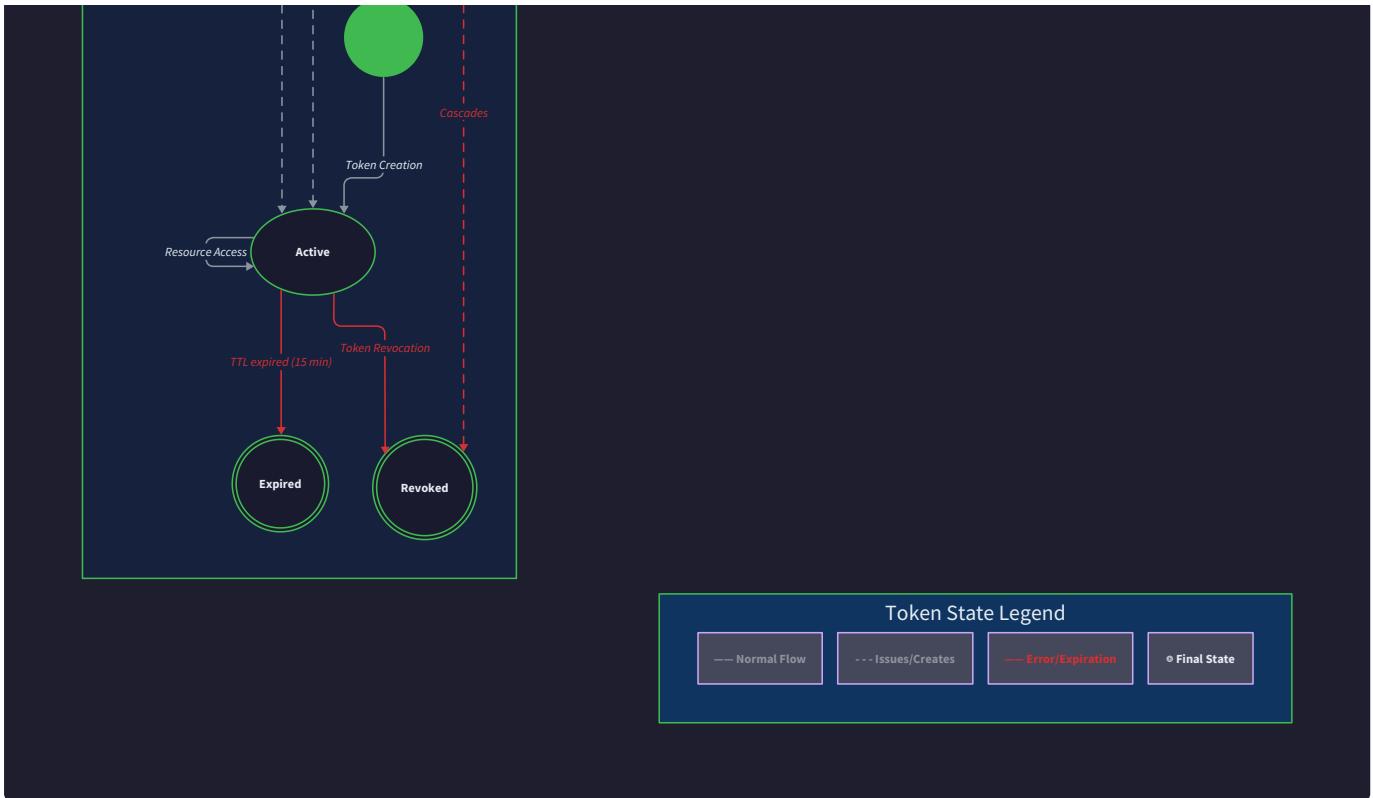
The authorization code is like a **time-sensitive voucher** that proves you've already gone through the proper authorization process (getting approval from the venue's security desk). However, this voucher itself can't get you into the concert - it's just proof that you're authorized to receive the real tickets. The voucher has an expiration time (10 minutes maximum) and can only be used once to prevent fraud.

When you present your voucher at the box office (token endpoint), the clerk (token service) performs several verification steps: they check that the voucher is genuine, hasn't expired, hasn't been used before, and matches the name on your ID (client authentication). They also verify that you're the same person who originally requested the voucher by checking your PKCE code verifier - think of this as a unique password you created when you first requested access.

Once all verifications pass, the box office issues you two types of tickets: an **access token** (like a concert ticket with specific seat assignments and time validity) and a **refresh token** (like a season pass that allows you to get new concert tickets without going through the full authorization process again). The access token is short-lived but gives you immediate access to the specific resources (concert sections) you're authorized for. The refresh token is long-lived but can only be used to obtain new access tokens, not for direct resource access.

For corporate clients (machine-to-machine scenarios), there's a special VIP box office where organizations with proper business credentials can directly purchase tickets without individual user vouchers - this is the client credentials grant flow.





JWT Access Token Generation

The heart of our token endpoint is the generation of **JSON Web Tokens (JWTs)** that serve as cryptographically signed access tokens. Unlike opaque token formats, JWTs are self-contained credentials that carry all necessary information for resource servers to validate access without consulting the authorization server.

A JWT consists of three base64url-encoded components separated by dots: header.payload.signature. The **header** specifies the signing algorithm and token type. The **payload** contains claims about the token's subject, issuer, audience, expiration, and granted scopes. The **signature** provides cryptographic proof that the token hasn't been tampered with and was issued by the legitimate authorization server.

Our JWT generation process follows a carefully designed sequence to ensure both security and interoperability. When an authorization code is successfully exchanged, we construct the JWT payload with mandatory claims required by the OAuth2 and OIDC specifications, plus any additional claims specific to our implementation.

JWT Claims Structure:

Claim	Type	Required	Description
<code>sub</code>	string	Yes	Subject identifier - the user ID who authorized the token
<code>iss</code>	string	Yes	Issuer - our authorization server's identifier URL
<code>aud</code>	string/array	Yes	Audience - intended recipients (resource servers) for this token
<code>exp</code>	number	Yes	Expiration time as Unix timestamp - when this token expires
<code>iat</code>	number	Yes	Issued at time as Unix timestamp - when this token was created
<code>jti</code>	string	Yes	JWT ID - unique identifier for this specific token instance
<code>scope</code>	string	Yes	Space-separated list of scopes granted to this token
<code>client_id</code>	string	No	Client identifier that requested this token
<code>token_use</code>	string	No	Always "access" to distinguish from ID tokens

The token generation algorithm follows these precise steps:

- 1. Claim Assembly:** We start by constructing the base claims object with the subject (`sub`) field set to the user ID from the authorization code, the issuer (`iss`) set to our server's base URL, and the audience (`aud`) configured based on the intended resource servers.
- 2. Timestamp Calculation:** We set the issued-at time (`iat`) to the current Unix timestamp and calculate the expiration time (`exp`) by adding our configured token lifetime (typically 15-60 minutes for access tokens).
- 3. Unique Identifier Generation:** We generate a cryptographically secure random JWT ID (`jti`) that serves as a unique identifier for this token instance, enabling precise revocation tracking.
- 4. Scope Inclusion:** We copy the exact scope string from the authorization code into the `scope` claim, ensuring the token carries the precise permissions that were authorized by the user.
- 5. Header Construction:** We build the JWT header specifying our chosen signing algorithm (RS256 for RSA signatures or ES256 for ECDSA) and the standard "JWT" token type.
- 6. Signature Generation:** We serialize the header and payload, base64url-encode them, concatenate with a dot separator, and generate a cryptographic signature using our private key.
- 7. Token Assembly:** Finally, we concatenate the encoded header, payload, and signature with dot separators to produce the final JWT access token.

JWT Generation Service Interface:

Method	Parameters	Returns	Description
GenerateAccessToken	<code>userID string, clientID string, scope string, audience []string</code>	<code>string, error</code>	Creates a signed JWT access token with specified claims
GenerateRefreshToken	<code>userID string, clientID string, scope string, tokenFamily string</code>	<code>*RefreshToken, error</code>	Creates a new refresh token with family tracking
ValidateJWT	<code>tokenString string</code>	<code>*jwt.Claims, error</code>	Verifies JWT signature and extracts validated claims
RevokeToken	<code>jti string</code>	<code>error</code>	Adds JWT ID to revocation list until expiry

Critical Security Insight: The JWT payload is only base64-encoded, not encrypted. Never include sensitive information like passwords, social security numbers, or private user data in JWT claims. The signature prevents tampering but doesn't provide confidentiality.

Token Signing Algorithm Selection:

Our implementation supports both RSA and ECDSA signing algorithms, each with distinct security and performance characteristics. RSA with SHA-256 (RS256) provides wide compatibility across OAuth2 clients and resource servers, making it the safer default choice for interoperability. ECDSA with P-256 (ES256) offers equivalent security with smaller key sizes and signatures, resulting in better performance and smaller tokens.

For token generation, we maintain both a private key for signing and a corresponding public key that resource servers use for signature verification. The public key must be made available through a JSON Web Key Set (JWKS) endpoint that clients can discover and fetch.

Access Token Lifetime Strategy:

Access tokens should be short-lived to limit the window of exposure if compromised. Industry best practices recommend 15 minutes to 1 hour maximum lifetime for access tokens. Shorter lifetimes provide better security by reducing the risk from stolen tokens, but increase the frequency of refresh token usage and potential performance impact.

Our default configuration uses 30-minute access tokens as a balance between security and usability. Resource-intensive applications or high-security environments may prefer 15-minute tokens, while internal applications with lower risk profiles might use 1-hour tokens.

Refresh Token Rotation

Refresh tokens enable applications to obtain new access tokens without requiring user re-authentication, but they represent a significant security risk if compromised since they typically have much longer lifetimes (days to months). To mitigate this risk, we implement **refresh token rotation** with family tracking - a security mechanism that detects and responds to token theft attempts.

Refresh Token Family Concept:

Every initial authorization generates a new **token family** identified by a unique family ID. When a refresh token is used to obtain new access tokens, we generate a completely new refresh token and invalidate the old one. Critically, the new refresh token belongs to the same family as its parent, creating a chain of related tokens.

This family relationship enables powerful security detection: if an application ever attempts to use an old (already rotated) refresh token, it indicates either a programming error or token theft. In response to this violation, we immediately revoke the entire token family, forcing the user to re-authenticate and preventing further abuse of any stolen tokens.

Refresh Token Data Model:

Field	Type	Description
Token	string	Cryptographically random refresh token string
ClientID	string	Client that this refresh token was issued to
UserID	string	User who authorized this token
Scope	string	Scopes that can be granted when refreshing
ExpiresAt	time.Time	When this refresh token expires
TokenFamily	string	Unique identifier linking related refresh tokens
ParentToken	*string	Previous refresh token in this family (null for initial)
Revoked	bool	Whether this token has been revoked
CreatedAt	time.Time	When this refresh token was created
LastUsedAt	*time.Time	When this token was last used for refresh

Refresh Token Rotation Algorithm:

- 1. Token Validation:** When a refresh request arrives, we first validate that the presented refresh token exists in our database, hasn't expired, and hasn't been marked as revoked.
- 2. Family Breach Detection:** We check if the token has already been used (has a non-null `LastUsedAt` field). If so, this indicates a replay attack or programming error, triggering immediate family revocation.
- 3. New Token Generation:** For valid refresh requests, we generate a new cryptographically random refresh token string and create a new `RefreshToken` record with the same family ID as the current token.
- 4. Parent Linking:** We set the new token's `ParentToken` field to point to the current token being exchanged, maintaining the family chain for audit purposes.
- 5. Current Token Invalidation:** We mark the current refresh token's `LastUsedAt` field and set a flag indicating it has been consumed, preventing reuse.
- 6. Access Token Issuance:** We generate a new JWT access token with the same scope and user information as the refresh token being exchanged.
- 7. Atomic Persistence:** All database changes (new refresh token creation, old token invalidation, access token logging) must be performed atomically to prevent inconsistent state.

Family Revocation Process:

When we detect refresh token reuse, indicating potential compromise, we execute a family-wide revocation:

1. **Family Discovery:** Starting with the compromised token's family ID, we query all refresh tokens belonging to that family.
2. **Batch Revocation:** We mark all tokens in the family as revoked and record the revocation timestamp for audit logging.
3. **Active Token Invalidation:** Any access tokens currently issued from this family should be added to a revocation list until their natural expiry.
4. **Client Notification:** We return an `invalid_grant` error to the client attempting the refresh, forcing them to restart the authorization flow.
5. **Security Logging:** We generate detailed audit logs including the compromised token, family size, and suspected breach time for security analysis.

Refresh Token Storage Strategy:

Refresh tokens require persistent storage with efficient lookup capabilities. We need to support queries by token string (for refresh operations), by family ID (for revocation), and by user ID (for administrative operations). A relational database with proper indexing provides the necessary query flexibility and transactional guarantees.

For high-throughput scenarios, consider implementing a write-through cache layer that stores recently active refresh tokens in memory while maintaining the database as the authoritative source. This reduces database load for frequent refresh operations while preserving the durability guarantees needed for security.

Client Credentials Grant

The **client credentials grant** provides a streamlined authentication flow for machine-to-machine scenarios where no human user is involved. This grant type allows confidential clients (servers, background services, APIs) to authenticate directly with the authorization server using their client credentials and receive access tokens for accessing protected resources.

Unlike the authorization code flow, the client credentials grant bypasses user authorization entirely since the client is acting on its own behalf rather than on behalf of a user. This makes it ideal for server-to-server API calls, scheduled background jobs, and microservice authentication.

Client Credentials Flow Sequence:

1. **Direct Authentication:** The client makes a POST request directly to the token endpoint with its client ID and client secret, along with the grant type set to "client_credentials".
2. **Client Authentication:** We verify the client credentials using the same authentication methods as the authorization code flow (HTTP Basic or POST body parameters).
3. **Scope Validation:** We check that the requested scopes are within the client's registered allowed scopes, rejecting any scope escalation attempts.
4. **Token Generation:** We generate a JWT access token with the client ID as the subject (no user ID since there's no user involved) and the requested scopes.
5. **Response:** We return the access token directly to the client with the standard OAuth2 token response format.

Client Credentials Token Claims:

For client credentials tokens, the JWT payload differs slightly from user-authorized tokens:

Claim	Value	Description
sub	client_id	The client itself is the subject
iss	server_url	Our authorization server identifier
aud	configured_audience	Resource servers that accept these tokens
exp	current_time + lifetime	Token expiration (typically longer than user tokens)
iat	current_time	Token issuance time
jti	random_uuid	Unique token identifier
scope	requested_scopes	Space-separated scope string
client_id	client_id	Explicit client identifier
token_use	"access"	Token type indicator
gtty	"client_credentials"	Grant type used to obtain this token

Client Credentials vs Authorization Code Tokens:

The key differences between client credentials tokens and user-authorized tokens reflect their different security contexts:

- **Subject:** Client credentials tokens have the client ID as the subject since there's no user involved
- **Lifetime:** These tokens can have longer lifetimes (1-24 hours) since they represent ongoing service authorization rather than temporary user delegation
- **Refresh Tokens:** Client credentials grants typically don't issue refresh tokens since clients can simply re-authenticate when needed
- **Scope Restrictions:** Scopes are limited to what the client is pre-authorized for during registration, with no user consent step

Client Credentials Security Considerations:

Client credentials represent powerful, long-lived authentication mechanisms that require careful security handling:

Security Principle: Client credentials tokens should be scoped to the minimum necessary permissions and should include rate limiting to prevent abuse if compromised.

Clients must securely store their client secrets using the same practices as database passwords or API keys. For containerized deployments, use secret management systems rather than environment variables or configuration files. Consider implementing client certificate authentication for high-security scenarios where mutual TLS provides stronger authentication than shared secrets.

Client Credentials Grant Implementation:

Method	Parameters	Returns	Description
HandleClientCredentials	clientID string, scope string	*TokenResponse, error	Processes client credentials grant requests
ValidateClientScope	client *Client, requestedScope string	error	Ensures requested scopes are within client's allowed scopes
GenerateClientToken	clientID string, scope string	string, error	Creates JWT access token for client credentials flow

Architecture Decision Records

Decision: JWT Signing Algorithm Selection

- Context:** We need to choose between HMAC (HS256), RSA (RS256), and ECDSA (ES256) for JWT signing. The choice affects security, performance, and key management complexity.
- Options Considered:**
 1. HMAC-SHA256 (HS256) - symmetric signing with shared secrets
 2. RSA-SHA256 (RS256) - asymmetric signing with RSA key pairs
 3. ECDSA-P256 (ES256) - asymmetric signing with elliptic curve keys
- Decision:** Use RS256 as the default with optional ES256 support
- Rationale:** HMAC requires sharing the same secret between authorization server and all resource servers, creating a security vulnerability if any resource server is compromised. RSA provides better key separation (private key stays on auth server, public keys distributed to resource servers) and has the widest compatibility across OAuth2 implementations. ES256 offers better performance and smaller signatures but has less universal support.
- Consequences:** Enables distributed token validation without secret sharing, requires RSA key pair management, slightly larger tokens than ECDSA but better client compatibility.

JWT Algorithm Comparison:

Algorithm	Key Type	Signature Size	Performance	Compatibility	Key Management
HS256	Symmetric	32 bytes	Fastest	Universal	Shared secrets (risky)
RS256	RSA (2048-bit)	256 bytes	Moderate	Excellent	Public/private key pairs
ES256	ECDSA P-256	64 bytes	Fast	Good	Public/private key pairs

Decision: Access Token Lifetime Strategy

- **Context:** Access token lifetime affects security (shorter is safer) versus performance (longer reduces refresh frequency). Industry practices range from 5 minutes to 24 hours.
- **Options Considered:**
 1. Short-lived tokens (5-15 minutes) - maximum security
 2. Medium-lived tokens (30-60 minutes) - balanced approach
 3. Long-lived tokens (2-24 hours) - maximum performance
- **Decision:** Default to 30-minute access tokens with configurable lifetimes per client
- **Rationale:** 30 minutes provides reasonable security (limits exposure window if tokens are compromised) while avoiding excessive refresh token usage. Per-client configuration allows high-security clients to use shorter lifetimes and internal services to use longer lifetimes based on risk assessment.
- **Consequences:** Balances security and performance, requires refresh token rotation every 30 minutes for long-running applications, enables risk-based lifetime adjustment per client.

Decision: Refresh Token Storage and Family Tracking

- **Context:** Refresh tokens need persistent storage and family tracking for security. Options include database storage, encrypted cookies, or hybrid approaches.
- **Options Considered:**
 1. Database storage with full family tracking - maximum security and auditability
 2. Encrypted stateless tokens - no database dependency but limited revocation
 3. Hybrid approach with database references and encrypted tokens
- **Decision:** Database storage with comprehensive family tracking and revocation support
- **Rationale:** Security requirements (immediate revocation, breach detection, audit logging) require persistent state that stateless approaches cannot provide. Database storage enables precise family tracking, immediate revocation, and comprehensive audit trails essential for production OAuth2 deployments.
- **Consequences:** Requires database operations for every refresh, enables immediate revocation and breach detection, supports comprehensive audit logging, scales with database performance characteristics.

Decision: Client Credentials Token Lifetime

- **Context:** Client credentials tokens represent service-to-service authentication and can have different lifetime requirements than user-authorized tokens.
- **Options Considered:**
 1. Same lifetime as user tokens (30 minutes) - consistent but potentially inefficient
 2. Longer lifetime (2-24 hours) - reduced authentication overhead
 3. Configurable per-client lifetime - maximum flexibility
- **Decision:** Default 2-hour lifetime with per-client configuration support
- **Rationale:** Service-to-service authentication typically involves lower risk than user delegation and benefits from reduced authentication overhead. 2 hours provides reasonable security while minimizing unnecessary token refresh operations. Per-client configuration allows high-security services to use shorter lifetimes.
- **Consequences:** Reduces authentication overhead for service clients, requires revocation list maintenance for longer periods, enables risk-appropriate lifetime configuration per service.

Common Pitfalls

⚠ Pitfall: Including Sensitive Data in JWT Payload

A common mistake is storing sensitive information like passwords, social security numbers, or private user data in JWT claims. Since JWTs are only base64-encoded (not encrypted), anyone who obtains the token can decode and read the payload.

Why it's wrong: JWT payloads are readable by anyone with access to the token. Base64 encoding provides no confidentiality - it's just a text encoding format, not encryption.

How to fix: Only include non-sensitive claims in JWTs. Use the subject (`sub`) field to identify users and let resource servers look up detailed user information from secure databases using the subject identifier.

⚠ Pitfall: Not Implementing Proper Refresh Token Rotation

Many implementations reuse the same refresh token multiple times instead of generating a new token on each refresh operation, missing critical security benefits of token rotation.

Why it's wrong: Reusing refresh tokens prevents detection of token theft. If a refresh token is compromised, the attacker can continue using it indefinitely without the legitimate client detecting the breach.

How to fix: Generate a new refresh token on every refresh operation and invalidate the old one. Implement family tracking to detect when old tokens are reused and revoke the entire family when this occurs.

⚠ Pitfall: Using Weak Random Number Generation

Using predictable random number generators or insufficient entropy for token generation creates security vulnerabilities where tokens can be guessed or predicted.

Why it's wrong: Predictable tokens enable attackers to forge valid tokens or guess other users' tokens, completely bypassing the authorization system.

How to fix: Use cryptographically secure random number generators (`crypto/rand` in Go) with sufficient entropy (at least 128 bits for authorization codes, 256 bits for refresh tokens). Never use `math/rand` or timestamp-based

generation for security tokens.

⚠ Pitfall: Not Validating JWT Expiration with Clock Skew Tolerance

Implementing strict expiration checking without accounting for clock differences between servers can cause valid tokens to be rejected due to minor time synchronization issues.

Why it's wrong: Distributed systems have inevitable clock skew, and rejecting tokens due to minor time differences (1-2 minutes) creates unnecessary authentication failures.

How to fix: Implement clock skew tolerance (typically 2-5 minutes) when validating JWT expiration times. Accept tokens that expired within the tolerance window while still enforcing meaningful expiration limits.

⚠ Pitfall: Missing Atomic Database Operations for Token Rotation

Failing to use database transactions when rotating refresh tokens can create inconsistent states where both old and new tokens remain valid, or neither token is valid.

Why it's wrong: Race conditions during token rotation can leave the database in an inconsistent state, causing authentication failures or security vulnerabilities.

How to fix: Use database transactions to atomically create new refresh tokens and invalidate old ones. Ensure that either both operations succeed or both fail, never leaving the system in a partially updated state.

⚠ Pitfall: Not Implementing Proper Client Scope Validation

Allowing clients to request any scope during client credentials grants without validating against registered scopes enables privilege escalation attacks.

Why it's wrong: Clients could request scopes they weren't authorized for during registration, gaining access to resources beyond their intended permissions.

How to fix: Always validate that requested scopes are within the client's registered allowed scopes. Reject requests that attempt to escalate privileges beyond what was granted during client registration.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
JWT Library	github.com/golang-jwt/jwt/v5	github.com/lestrrat-go/jwx/v2
Database	SQLite with github.com/mattn/go-sqlite3	PostgreSQL with github.com/lib/pq
Cryptographic Random	<code>crypto/rand</code> package	Hardware security module (HSM)
Key Management	File-based RSA keys	AWS KMS or HashiCorp Vault
Token Storage	In-memory map with mutex	Redis with TTL support
Time Handling	<code>time.Now()</code> with manual skew	NTP-synchronized time service

Recommended File Structure:

```
internal/token/
endpoint.go           ← HTTP handlers for token endpoint
generator.go          ← JWT generation and signing logic
refresh.go            ← Refresh token rotation and family tracking
client_credentials.go ← Client credentials grant implementation
validator.go          ← JWT validation and revocation checking
storage.go            ← Token persistence interfaces
crypto.go             ← Cryptographic utilities and key management
internal/storage/
refresh_token_store.go ← Refresh token database operations
revocation_store.go   ← JWT revocation list management
internal/crypto/
keys.go               ← RSA/ECDSA key loading and management
random.go             ← Secure random token generation
```

Infrastructure Starter Code:

Cryptographic Key Management (crypto/keys.go):

```
package crypto
```

GO

```
import (
    "crypto/rand"
    "crypto/rsa"
    "crypto/x509"
    "encoding/pem"
    "fmt"
    "os"
)

type KeyManager struct {
    privateKey *rsa.PrivateKey
    keyID      string
}

func NewKeyManager(keyPath, keyID string) (*KeyManager, error) {
    keyData, err := os.ReadFile(keyPath)
    if err != nil {
        return nil, fmt.Errorf("failed to read private key: %w", err)
    }

    block, _ := pem.Decode(keyData)
    if block == nil {
        return nil, fmt.Errorf("invalid PEM format")
    }

    privateKey, err := x509.ParsePKCS1PrivateKey(block.Bytes)
    if err != nil {
        return nil, fmt.Errorf("failed to parse private key: %w", err)
    }
}
```

```
return &KeyManager{  
  
    privateKey: privateKey,  
  
    keyID:      keyID,  
  
, nil  
}  
  
func (km *KeyManager) GetPrivateKey() *rsa.PrivateKey {  
  
    return km.privateKey  
}  
  
func (km *KeyManager) GetKeyID() string {  
  
    return km.keyID  
}  
  
func GenerateSecureToken(length int) (string, error) {  
  
    bytes := make([]byte, length)  
  
    if _, err := rand.Read(bytes); err != nil {  
  
        return "", fmt.Errorf("failed to generate random bytes: %w", err)  
    }  
  
    // Convert to URL-safe base64  
  
    return base64.RawURLEncoding.EncodeToString(bytes), nil  
}
```

Refresh Token Storage (`storage/refresh_token_store.go`):

```
package storage
```

GO

```
import (
    "context"
    "database/sql"
    "time"
)

type RefreshTokenStore struct {
    db *sql.DB
}

func NewRefreshTokenStore(db *sql.DB) *RefreshTokenStore {
    return &RefreshTokenStore{db: db}
}

func (store *RefreshTokenStore) CreateRefreshToken(ctx context.Context, token *RefreshToken) error {
    query := `

        INSERT INTO refresh_tokens
        (token, client_id, user_id, scope, expires_at, token_family, parent_token, created_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)

        `

    _, err := store.db.ExecContext(ctx, query,
        token.Token, token.ClientID, token.UserID, token.Scope,
        token.ExpiresAt, token.TokenFamily, token.ParentToken, token.CreatedAt)

    if err != nil {
        return fmt.Errorf("failed to create refresh token: %w", err)
    }

    return nil
}
```

```
}

func (store *RefreshTokenStore) GetRefreshToken(ctx context.Context, tokenStr string) (*RefreshToken, error) {

    query := `

        SELECT token, client_id, user_id, scope, expires_at, token_family,
               parent_token, revoked, created_at, last_used_at
        FROM refresh_tokens
        WHERE token = ? AND NOT revoked AND expires_at > ?

`


    var token RefreshToken
    var parentToken sql.NullString
    var lastUsedAt sql.NullTime

    err := store.db.QueryRowContext(ctx, query, tokenStr, time.Now()).Scan(
        &token.Token, &token.ClientID, &token.UserID, &token.Scope,
        &token.ExpiresAt, &token.TokenFamily, &parentToken, &token.Revoked,
        &token.CreatedAt, &lastUsedAt)

    if err == sql.ErrNoRows {
        return nil, ErrRefreshTokenNotFound
    } else if err != nil {
        return nil, fmt.Errorf("failed to get refresh token: %w", err)
    }

    if parentToken.Valid {
        token.ParentToken = &parentToken.String
    }

    if lastUsedAt.Valid {
```

```

        token.LastUsedAt = &lastUsedAt.Time

    }

    return &token, nil
}

func (store *RefreshTokenStore) RevokeTokenFamily(ctx context.Context, familyID string) error {
    query := `

        UPDATE refresh_tokens

        SET revoked = true, revoked_at = ?

        WHERE token_family = ?

        `

    _, err := store.db.ExecContext(ctx, query, time.Now(), familyID)

    if err != nil {
        return fmt.Errorf("failed to revoke token family: %w", err)
    }

    return nil
}

```

Core Logic Skeleton Code:

JWT Token Generator (token/generator.go):

```
package token
```

GO

```
import (  
    "crypto/rsa"  
    "time"
```

```
    "github.com/golang-jwt/jwt/v5"
```

```
)
```

```
type JWTGenerator struct {
```

```
    privateKey *rsa.PrivateKey
```

```
    keyID      string
```

```
    issuer     string
```

```
}
```

```
// GenerateAccessToken creates a signed JWT access token with specified claims
```

```
func (g *JWTGenerator) GenerateAccessToken(userID, clientID, scope string, audience []string)  
(string, error) {
```

```
    now := time.Now()
```

```
// TODO 1: Create JWT claims object with required standard claims
```

```
// - Set subject (sub) to userID
```

```
// - Set issuer (iss) to g.issuer
```

```
// - Set audience (aud) to audience parameter
```

```
// - Set issued at (iat) to current Unix timestamp
```

```
// - Set expiration (exp) to current time + 30 minutes
```

```
// Hint: Use jwt.MapClaims for flexible claim structure
```

```
// TODO 2: Add custom claims for OAuth2 functionality
```

```
// - Set "scope" claim to the scope parameter
```

```
// - Set "client_id" claim to clientID parameter
```

```
// - Set "token_use" claim to "access"
```

```
// - Generate and set "jti" (JWT ID) claim using GenerateSecureToken(16)

// TODO 3: Create JWT token with claims and RS256 algorithm

// - Use jwt.NewWithClaims(jwt.SigningMethodRS256, claims)

// - Set the "kid" (key ID) header to g.keyID

// Hint: Access token.Header["kid"] to set key ID

// TODO 4: Sign the token with private key and return signed string

// - Use token.SignedString(g.privateKey)

// - Handle any signing errors appropriately

return "", nil // Remove this line when implementing

}

// GenerateRefreshToken creates a new refresh token with family tracking

func (g *JWTGenerator) GenerateRefreshToken(userID, clientID, scope, tokenFamily string)
(*RefreshToken, error) {

now := time.Now()

// TODO 1: Generate cryptographically secure random token string

// - Use GenerateSecureToken(32) for 256-bit security

// - This will be the actual refresh token presented by clients

// TODO 2: Calculate appropriate expiration time for refresh token

// - Refresh tokens should be much longer-lived than access tokens

// - Typically 30 days to 6 months depending on security requirements

// - Use 30 days as default: now.Add(30 * 24 * time.Hour)

// TODO 3: Create RefreshToken struct with all required fields

// - Set Token to generated random string
```

```
// - Set ClientID, UserID, Scope from parameters

// - Set TokenFamily to provided family ID

// - Set ExpiresAt to calculated expiration

// - Set CreatedAt to current time

// - Leave ParentToken nil for new tokens (will be set during rotation)

// TODO 4: Return populated RefreshToken struct

// - Caller is responsible for persisting to database

return nil, nil // Remove this line when implementing

}
```

Token Endpoint Handler (token/endpoint.go):

```
package token

import (
    "encoding/json"
    "net/http"
    "strings"
)

type TokenEndpoint struct {
    generator      *JWTGenerator
    clientStore   ClientStore
    refreshStore  RefreshTokenStore
    validator     *PKCEValidator
}

// HandleTokenRequest processes OAuth2 token endpoint requests

func (te *TokenEndpoint) HandleTokenRequest(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Validate HTTP method is POST
    // - Return method not allowed error for non-POST requests
    // - Use WriteOAuth2Error helper for proper error format

    // TODO 2: Parse grant_type parameter from form data
    // - Call r.ParseForm() to parse POST body
    // - Extract "grant_type" parameter
    // - Support "authorization_code", "refresh_token", and "client_credentials"

    // TODO 3: Route to appropriate grant handler based on grant_type
    // - Call te.handleAuthorizationCodeGrant for "authorization_code"
    // - Call te.handleRefreshTokenGrant for "refresh_token"
    // - Call te.handleClientCredentialsGrant for "client_credentials"
    // - Return unsupported_grant_type error for unknown grant types
}
```

GO

```
// TODO 4: Handle any errors from grant handlers

// - Grant handlers return TokenResponse and error

// - Write error response for failures

// - Write successful JSON response for success

}

// handleAuthorizationCodeGrant exchanges authorization code for tokens

func (te *TokenEndpoint) handleAuthorizationCodeGrant(r *http.Request) (*TokenResponse, error) {

    // TODO 1: Extract required parameters from request

    // - Get "code" parameter (the authorization code to exchange)

    // - Get "redirect_uri" parameter (must match original request)

    // - Get "client_id" parameter or from HTTP Basic auth

    // - Get "client_secret" parameter or from HTTP Basic auth

    // - Get "code_verifier" parameter for PKCE validation


    // TODO 2: Authenticate the client making the request

    // - Look up client using ClientStore.GetClient(clientID)

    // - Verify client_secret using bcrypt comparison

    // - Return invalid_client error for authentication failures


    // TODO 3: Retrieve and validate the authorization code

    // - Look up authorization code using provided code parameter

    // - Verify code hasn't expired (check ExpiresAt field)

    // - Verify code hasn't been used already (check Used field)

    // - Verify redirect_uri matches code's original redirect_uri

    // - Return invalid_grant error for any validation failures


    // TODO 4: Validate PKCE code verifier (if code_challenge was provided)

    // - Use PKCEValidator.VerifyCodeChallenge with code verifier and stored challenge
```

```
// - Return invalid_grant error if PKCE validation fails

// TODO 5: Generate access and refresh tokens

// - Generate new token family ID using GenerateSecureToken(16)

// - Create access token using JWTGenerator.GenerateAccessToken

// - Create refresh token using JWTGenerator.GenerateRefreshToken

// - Mark authorization code as used in database

// TODO 6: Return token response in standard OAuth2 format

// - Create TokenResponse with access_token, token_type="Bearer", expires_in

// - Include refresh_token in response

// - Include scope from authorization code

return nil, nil // Remove this line when implementing

}
```

Refresh Token Rotation Handler (token/refresh.go):

```
package token
```

GO

```
// handleRefreshTokenGrant exchanges refresh token for new access token

func (te *TokenEndpoint) handleRefreshTokenGrant(r *http.Request) (*TokenResponse, error) {

    // TODO 1: Extract refresh token and client credentials from request

    // - Get "refresh_token" parameter from form data

    // - Get client authentication (client_id and client_secret)

    // - Authenticate client same as authorization code grant

    // TODO 2: Retrieve refresh token from database

    // - Use RefreshTokenStore.GetRefreshToken to look up token

    // - Return invalid_grant error if token not found

    // - Verify token belongs to the authenticated client

    // TODO 3: Check for token family security breach

    // - If refresh token has non-null LastUsedAt field, it's been used before

    // - This indicates token replay attack or client programming error

    // - Call RefreshTokenStore.RevokeTokenFamily to revoke entire family

    // - Return invalid_grant error to force client re-authorization

    // TODO 4: Validate refresh token is still valid

    // - Check that current time is before ExpiresAt

    // - Verify token is not marked as revoked

    // - Return invalid_grant error for expired/revoked tokens

    // TODO 5: Generate new access token with same scope and user

    // - Use JWTGenerator.GenerateAccessToken with refresh token's UserID and Scope

    // - Access token should have same permissions as original authorization

    // TODO 6: Rotate the refresh token (create new, invalidate old)
```

```

    // - Generate new refresh token in same family using GenerateRefreshToken

    // - Set new token's ParentToken field to current token's Token value

    // - Mark current refresh token's LastUsedAt to prevent reuse

    // - Save new refresh token to database atomically

    // TODO 7: Return new tokens to client

    // - Create TokenResponse with new access_token and refresh_token

    // - Include same scope as original refresh token

    return nil, nil // Remove this line when implementing
}

}

```

Language-Specific Hints:

- Use `github.com/golang-jwt/jwt/v5` for JWT operations - it handles signing, verification, and claims parsing automatically
- Use `crypto/rand.Read()` for generating cryptographically secure random tokens - never use `math/rand` for security tokens
- Use `time.Now().Unix()` for JWT timestamp claims - OAuth2 specs require Unix timestamps
- Use database transactions (`db.BeginTx()`) for atomic refresh token rotation to prevent race conditions
- Use `bcrypt.CompareHashAndPassword()` for client secret verification with constant-time comparison
- Use `base64.RawURLEncoding` for token encoding to avoid padding characters that break URLs

Milestone Checkpoint:

After implementing the token endpoint component:

- 1. Test Authorization Code Exchange:** Start your OAuth2 server and make a POST request to `/oauth/token` with `grant_type=authorization_code`. You should receive a JSON response with `access_token`, `refresh_token`, and `expires_in` fields.
- 2. Verify JWT Structure:** Decode the access token at jwt.io to verify it contains the expected claims (`sub`, `iss`, `aud`, `exp`, `iat`, `jti`, `scope`).
- 3. Test Refresh Token Rotation:** Use the refresh token from step 1 to request new tokens. Verify you receive a new access token and refresh token, and that the old refresh token becomes invalid.
- 4. Test Client Credentials Grant:** Make a POST request with `grant_type=client_credentials` and valid client credentials. Verify you receive an access token without a refresh token.
- 5. Test Security Measures:** Try reusing an old refresh token and verify the entire token family gets revoked with an `invalid_grant` error.

Expected Behavior:

- Token endpoint responds to POST requests at `/oauth/token`
- Authorization code exchange returns `{"access_token": "...", "token_type": "Bearer", "expires_in": 1800, "refresh_token": "..."}`
- JWT access tokens are properly signed and contain required claims
- Refresh token rotation generates new tokens and invalidates old ones
- Client credentials grant works for machine-to-machine authentication

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
JWT signature verification fails	Wrong signing key or algorithm	Check private key matches public key, verify algorithm (RS256)	Ensure consistent key pair usage
Refresh token rotation fails	Database transaction issues	Check database logs for constraint violations	Use proper transactions and unique constraints
Client credentials returns invalid_client	Client authentication failure	Log client_id and verify against database	Check client secret hashing/comparison
Access tokens expire immediately	Wrong timestamp format	Verify exp claim is Unix timestamp	Use <code>time.Now().Unix()</code> not <code>time.Now()</code>

Token Introspection and Revocation Component

Milestone(s): Milestone 3 - Token Introspection & Revocation

Mental Model: ID Verification System

Think of token introspection and revocation like a sophisticated ID verification system at a high-security facility. When someone presents an access badge (access token) at a checkpoint, the security guard doesn't just look at the badge - they scan it in a verification system that checks multiple things: Is this badge authentic? Has it expired? Has it been reported stolen or lost? Is the person authorized for the specific area they're trying to access?

Token introspection works the same way. When a resource server receives an access token, it can query the authorization server's introspection endpoint to verify the token's authenticity, check its expiration status, and confirm what permissions (scopes) it actually grants. The introspection endpoint acts like the central security database that knows the current status of every issued badge.

Token revocation is like reporting a lost or stolen ID card. When someone's badge is compromised, stolen, or no longer needed, it gets added to a revocation list. The next time someone tries to use that badge, the verification system immediately rejects it, even if the badge itself looks valid and hasn't technically expired yet. In OAuth2, this happens when users revoke permissions, administrators disable access, or security incidents require immediate token invalidation.

The critical insight here is that JWTs, while self-contained and verifiable through their cryptographic signatures, still need a way to be revoked before their natural expiration. Unlike a traditional session token stored in a database (where you can

simply delete the record), JWTs carry their validity within themselves. This creates a tension between the stateless nature of JWTs and the security requirement to immediately invalidate compromised tokens. The solution involves maintaining a revocation list and providing introspection endpoints for real-time validation.

Introspection Endpoint

The token introspection endpoint, defined by RFC 7662, provides a standardized way for resource servers to validate tokens by querying the authorization server. This endpoint bridges the gap between stateless JWT tokens and the need for real-time validity checking, including revocation status.

Introspection Request Processing

When a resource server receives an access token from a client, it may need to validate that token's current status beyond what can be determined from the JWT signature and claims alone. The resource server makes an introspection request to the authorization server, presenting the token for validation along with client credentials to authenticate the introspection request itself.

The introspection endpoint processes these requests by first authenticating the requesting client. Not all clients should be able to introspect arbitrary tokens - typically, only resource servers and privileged clients receive this capability. The endpoint verifies the client's credentials using the same authentication mechanisms employed by the token endpoint.

Once the client is authenticated, the endpoint examines the presented token. For JWT access tokens, this involves several validation steps: verifying the signature using the appropriate public key, checking standard claims like expiration (`exp`) and issued-at (`iat`), and validating that the issuer (`iss`) and audience (`aud`) claims match expected values.

Revocation Status Checking

A critical aspect of introspection is checking whether the token has been explicitly revoked. Since JWTs are stateless, the authorization server must maintain a revocation list - a data structure tracking tokens that have been invalidated before their natural expiration. This revocation list is indexed by the JWT's unique identifier (the `jti` claim) to enable fast lookups.

The revocation list presents several implementation challenges. First, it grows continuously as tokens are revoked, but entries can be safely removed once the corresponding token's natural expiration time passes. Second, the list must be quickly accessible from the introspection endpoint, which may handle high request volumes. Third, in distributed deployments, the revocation list must be consistent across all authorization server instances.

Revocation Tracking Component	Purpose	Implementation Notes
<code>jti</code> Claim	Unique token identifier	Generated during token creation, immutable
Revocation Store	Persistent revocation list	Redis or database with TTL matching token expiry
Cache Layer	Fast revocation lookups	In-memory cache with background refresh
Cleanup Process	Remove expired entries	Background job running periodically
Replication Mechanism	Consistency across instances	Event-driven updates or shared storage

Introspection Response Format

The introspection endpoint returns a JSON response conforming to RFC 7662 specifications. The response always includes an `active` boolean field indicating whether the token is currently valid and usable. When `active` is true, the

response includes additional metadata about the token, such as the scopes it grants, the client it was issued to, and its expiration time.

Response Field	Type	Required	Description
active	boolean	Yes	Whether the token is active and valid
scope	string	No	Space-separated list of granted scopes
client_id	string	No	Client identifier the token was issued to
username	string	No	Human-readable user identifier
exp	integer	No	Token expiration timestamp (Unix epoch)
iat	integer	No	Token issued-at timestamp (Unix epoch)
sub	string	No	Subject identifier (user ID)
aud	string	No	Audience the token is intended for
iss	string	No	Issuer identifier
jti	string	No	Unique token identifier
token_type	string	No	Type of token (typically "Bearer")

When a token is invalid, expired, or revoked, the response contains only `{"active": false}`. This prevents information leakage about invalid tokens while still providing the binary validity information that resource servers need.

Client Authentication for Introspection

The introspection endpoint must authenticate clients making introspection requests to prevent unauthorized token validation attempts. This authentication typically uses the same client credential mechanisms as the token endpoint: HTTP Basic authentication with client ID and secret, or POST body parameters.

Different clients may have different introspection privileges. Resource servers typically can introspect any token presented to them, while regular OAuth clients might only be able to introspect tokens they originally requested. This authorization policy is enforced after client authentication but before token validation.

Decision: Require Client Authentication for Introspection

- **Context:** RFC 7662 makes client authentication optional, but allowing unauthenticated introspection creates security risks
- **Options Considered:**
 - Allow anonymous introspection for simplicity
 - Require authentication for all introspection requests
 - Support both modes with configuration
- **Decision:** Require client authentication for all introspection requests
- **Rationale:** Unauthenticated introspection enables token scanning attacks where malicious actors probe tokens to determine their validity. Client authentication provides accountability and enables fine-grained access control policies.
- **Consequences:** Increases complexity for resource servers (must have client credentials) but significantly improves security posture and enables audit logging

Revocation Endpoint

The token revocation endpoint, specified by RFC 7009, allows clients to notify the authorization server that a previously issued token is no longer needed and should be considered invalid immediately. This endpoint is crucial for security scenarios where tokens may be compromised or when users explicitly revoke permissions.

Revocation Request Processing

Clients make revocation requests by posting the token they wish to revoke along with their client credentials for authentication. The revocation endpoint accepts both access tokens and refresh tokens, though the specific revocation behavior differs depending on token type.

When a client submits a revocation request, the endpoint first authenticates the client using the same mechanisms as the token endpoint. The client must be authorized to revoke the presented token - typically meaning the client must be the same one that originally requested the token, or a privileged administrative client.

The endpoint then processes the token to determine its type and current status. For JWT access tokens, this involves parsing the token structure, verifying the signature, and extracting the unique identifier (`jti` claim). For refresh tokens, the endpoint queries the refresh token store to retrieve the token record and verify its association with the requesting client.

Access Token Revocation

Revoking JWT access tokens presents unique challenges because these tokens are designed to be self-contained and stateless. The authorization server cannot directly invalidate a JWT that has already been issued and distributed to resource servers.

Instead, access token revocation works by adding the token's unique identifier to a revocation list. When resource servers validate tokens (either through introspection or local validation), they must check this revocation list to determine if an otherwise-valid token has been explicitly revoked.

The revocation process involves several steps:

1. **Token Identification:** Extract the `jti` claim from the JWT to obtain a unique identifier

2. **Ownership Verification:** Confirm that the requesting client has permission to revoke this specific token
3. **Revocation Recording:** Add the token identifier to the revocation list with metadata about when and why it was revoked
4. **Cache Invalidation:** If caching is used, invalidate any cached positive validation results for this token
5. **Audit Logging:** Record the revocation event for security monitoring and compliance

Revocation Record Field	Type	Purpose
jti	string	Unique token identifier being revoked
client_id	string	Client that requested the revocation
revoked_at	timestamp	When the revocation occurred
reason	string	Why the token was revoked (user request, security incident, etc.)
expires_at	timestamp	When this revocation record can be cleaned up
token_type	string	Whether this was an access token or refresh token

Refresh Token Revocation

Refresh token revocation is more straightforward than access token revocation because refresh tokens are stored server-side and can be directly invalidated. However, refresh token revocation often has broader implications due to token family relationships.

When a refresh token is revoked, the authorization server marks it as invalid in the refresh token store. Additionally, if the implementation uses refresh token rotation with family tracking, revoking one refresh token may trigger revocation of the entire token family. This is a security measure that assumes if one token in the family is compromised, all related tokens should be considered potentially compromised.

The refresh token revocation process includes:

1. **Token Lookup:** Query the refresh token store to find the token record
2. **Authorization Check:** Verify the requesting client owns this token
3. **Family Assessment:** Determine if other tokens in the same family should be revoked
4. **Database Update:** Mark the token (and potentially family members) as revoked
5. **Related Token Cleanup:** Optionally revoke associated access tokens

Token Family Revocation

Token families provide a security mechanism for detecting and responding to token theft scenarios. When refresh tokens are rotated (a new refresh token is issued each time the old one is used), all tokens issued from the same original authorization form a "family" with a shared family identifier.

If an old refresh token from a family is presented for use after a newer token in the same family has already been used, this indicates a potential security breach - someone may be replaying a stolen token. In response, the authorization server revokes the entire token family, forcing the legitimate client to re-authenticate.

Token family revocation extends to explicit revocation requests. When a user revokes consent for an application or an administrator terminates a session, revoking the entire token family ensures that all related tokens become invalid, regardless of where they might be stored or cached.

Decision: Implement Token Family Revocation for Security Breach Detection

- **Context:** Refresh token rotation creates families of related tokens, but detection of replay attacks requires family-level revocation
- **Options Considered:**
 - Revoke only the specific token requested
 - Revoke token families only on detected replay attacks
 - Always revoke entire families for any revocation request
- **Decision:** Revoke entire token families when replay attacks are detected, individual tokens for explicit user/admin revocation
- **Rationale:** Balances security (detecting token theft) with usability (not over-revoking for legitimate revocation requests)
- **Consequences:** Requires family tracking infrastructure but provides strong protection against token theft scenarios

JWT Validation Library

Resource servers need the ability to validate JWT access tokens locally without always requiring introspection calls to the authorization server. A JWT validation library provides this capability while properly handling the complexities of cryptographic verification, claims validation, and revocation checking.

Local JWT Validation Process

Local JWT validation offers significant performance and availability advantages over introspection-based validation. Resource servers can validate tokens independently, reducing network round-trips and eliminating dependencies on authorization server availability for routine token validation.

The validation process involves multiple steps, each addressing different aspects of token integrity and authenticity:

1. **Token Structure Parsing:** Parse the JWT into header, payload, and signature components
2. **Header Validation:** Verify algorithm specification matches expected values and extract key identifier
3. **Signature Verification:** Cryptographically verify the token signature using the appropriate public key
4. **Standard Claims Validation:** Check expiration (`exp`), not-before (`nbf`), and issued-at (`iat`) timestamps
5. **Custom Claims Validation:** Verify issuer (`iss`), audience (`aud`), and subject (`sub`) claims match expected values
6. **Revocation Checking:** Query revocation list or cache to confirm token hasn't been explicitly revoked
7. **Scope Authorization:** Verify that granted scopes authorize the requested resource access

Validation Step	Purpose	Failure Response
Structure Parse	Ensure valid JWT format	401 Unauthorized
Algorithm Check	Prevent algorithm substitution attacks	401 Unauthorized
Signature Verify	Confirm token authenticity	401 Unauthorized
Expiration Check	Reject expired tokens	401 Unauthorized
Audience Validate	Ensure token intended for this resource	403 Forbidden
Revocation Check	Respect explicit token invalidation	401 Unauthorized
Scope Authorize	Confirm permission for requested resource	403 Forbidden

Cryptographic Key Management

JWT validation requires access to the public keys used by the authorization server to sign tokens. The validation library must securely obtain and manage these keys, handling key rotation and multiple concurrent keys during transition periods.

The authorization server typically publishes its public keys through a JSON Web Key Set (JWKS) endpoint, following the well-known configuration pattern. The validation library fetches these keys and caches them locally, refreshing periodically or when encountering tokens signed with unknown key identifiers.

Key management includes several critical security considerations. The library must validate that keys are obtained from the correct authorization server to prevent key substitution attacks. It should cache keys with appropriate expiration times to balance performance with security - too long and revoked keys remain trusted, too short and performance suffers from constant key fetching.

Key Management Component	Responsibility	Security Consideration
JWKS Client	Fetch public keys from authorization server	Verify HTTPS and certificate validation
Key Cache	Store keys locally for performance	Implement reasonable TTL (1-24 hours)
Key Rotation Handler	Detect and handle key changes	Support multiple keys during rotation
Algorithm Validator	Ensure only secure algorithms accepted	Reject symmetric algorithms (HS256)
Key ID Resolver	Map token <code>kid</code> to verification key	Handle missing or unknown key IDs

Revocation Checking Integration

Even with local JWT validation, resource servers must still check token revocation status to handle tokens that have been explicitly invalidated before expiration. The validation library can implement this in several ways, each with different performance and consistency trade-offs.

The most secure approach queries the introspection endpoint for every token validation, but this eliminates the performance benefits of local validation. A more balanced approach maintains a local cache of revoked token identifiers, periodically refreshed from the authorization server or updated through push notifications.

For high-performance scenarios, the library might implement a bloom filter or similar probabilistic data structure to quickly eliminate tokens that definitely haven't been revoked, falling back to introspection only for tokens that might be revoked.

This approach requires careful tuning to balance false positive rates with performance requirements.

Validation Result Caching

JWT validation involves computationally expensive operations, particularly signature verification. The validation library can cache successful validation results to avoid repeating these operations for frequently-used tokens, but this caching must be implemented carefully to maintain security properties.

Validation result caching must consider token expiration times, ensuring cached results don't outlive the token's validity period. It should also integrate with revocation checking, invalidating cached results when tokens are revoked. The cache size should be limited to prevent memory exhaustion from token scanning attacks.

Decision: Implement Validation Result Caching with Revocation Integration

- **Context:** JWT signature verification is computationally expensive, but caching validation results risks serving revoked tokens
- **Options Considered:**
 - No caching - validate every request completely
 - Cache validation results with token expiration TTL
 - Cache validation results with shorter TTL for revocation safety
- **Decision:** Cache validation results with TTL of min(token_expiration, 5_minutes) and invalidate on revocation
- **Rationale:** Provides significant performance improvement while maintaining revocation security through short cache windows
- **Consequences:** Requires cache invalidation infrastructure but reduces CPU load by 80-90% for repeated token usage

Architecture Decision Records

Decision: Use Redis for Distributed Revocation List

Decision: Use Redis for Distributed Revocation List

- **Context:** Token revocation must be consistent across multiple authorization server instances, and revocation lists need fast access for introspection
- **Options Considered:**
 - Database-only storage with caching
 - Redis with database backup
 - In-memory with event propagation
- **Decision:** Redis as primary revocation store with database backup for durability
- **Rationale:** Redis provides sub-millisecond access times, built-in TTL for automatic cleanup, and pub/sub for cache invalidation. Database backup ensures durability across Redis restarts.
- **Consequences:** Introduces Redis dependency but enables horizontal scaling and high-performance revocation checking

Option	Pros	Cons
Database Only	Simple architecture, ACID guarantees	Slower access times, complex caching
Redis Primary	Very fast access, automatic TTL cleanup	Additional infrastructure, eventual consistency
In-Memory	Fastest possible access	Complex synchronization, memory limits

Decision: Implement JWT ID (jti) Claims for Revocation Tracking

Decision: Implement JWT ID (jti) Claims for Revocation Tracking

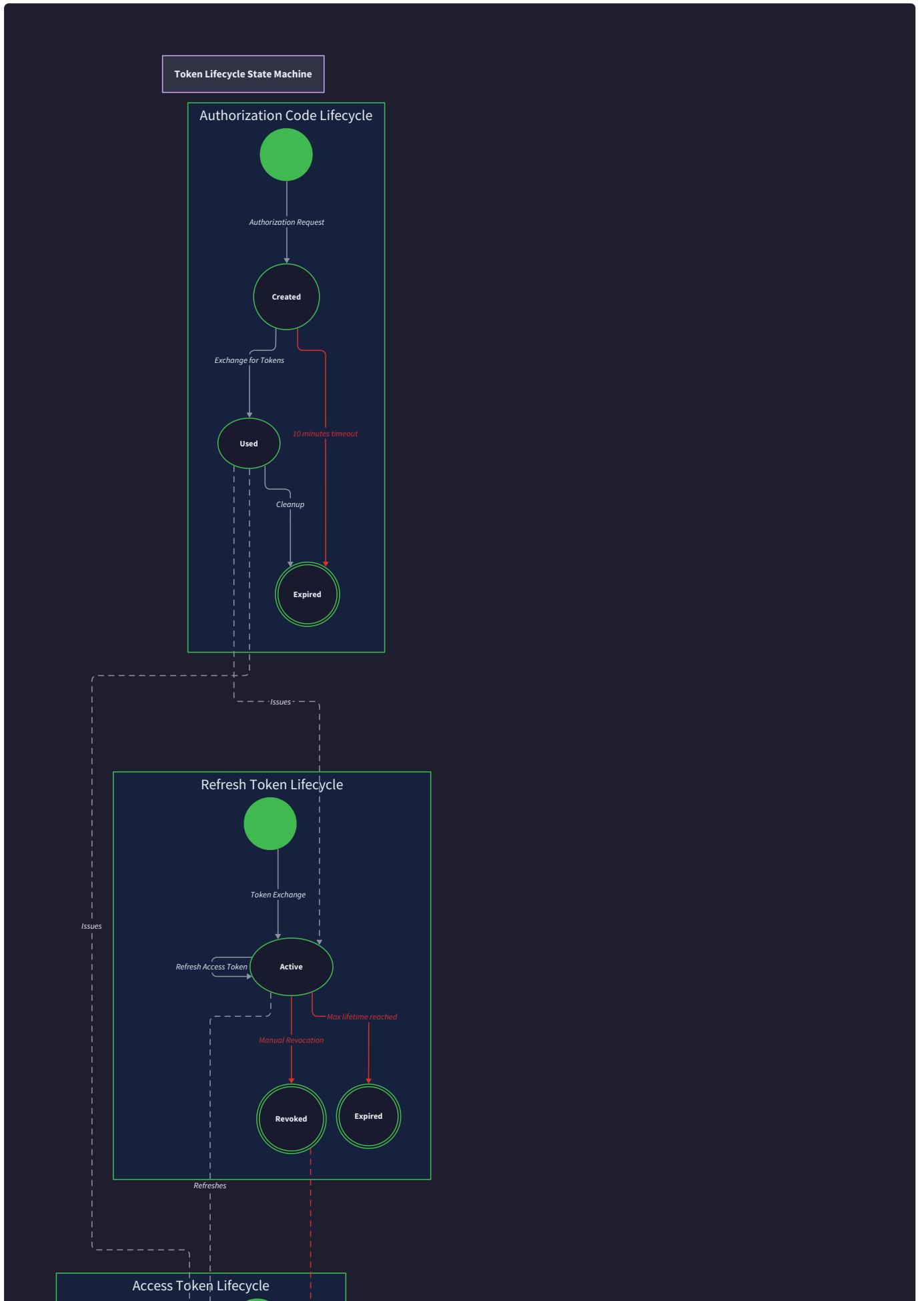
- **Context:** Revoking stateless JWTs requires a way to uniquely identify tokens that have been invalidated
- **Options Considered:**
 - Use combination of iss/sub/iat as identifier
 - Generate UUID v4 for each token as jti
 - Use sequential IDs with issuer prefix
- **Decision:** Generate cryptographically random UUID v4 for each JWT as jti claim
- **Rationale:** UUID v4 provides uniqueness without coordination, unpredictability for security, and standard format recognition
- **Consequences:** Increases JWT size slightly but enables precise revocation tracking without collisions

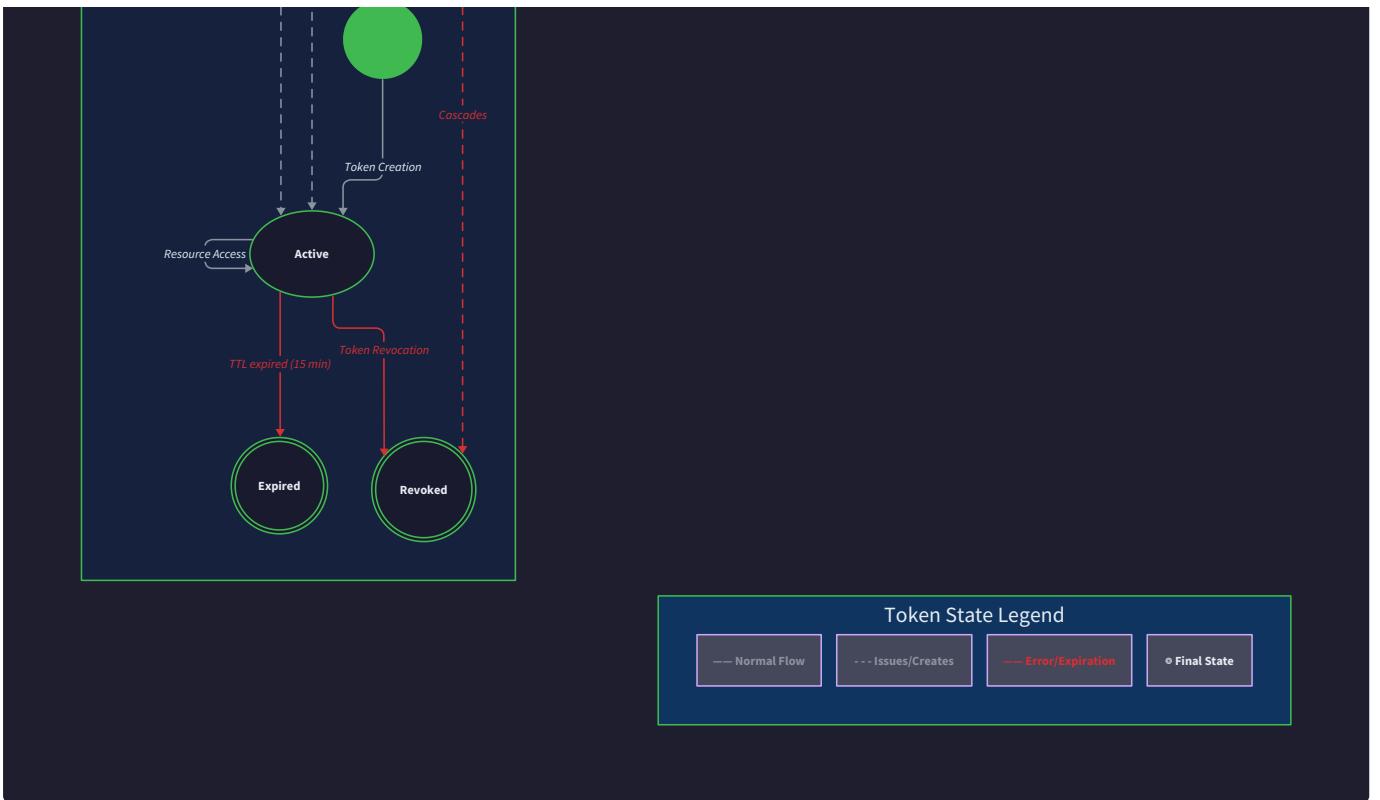
Decision: Require Client Authentication for Introspection Endpoint

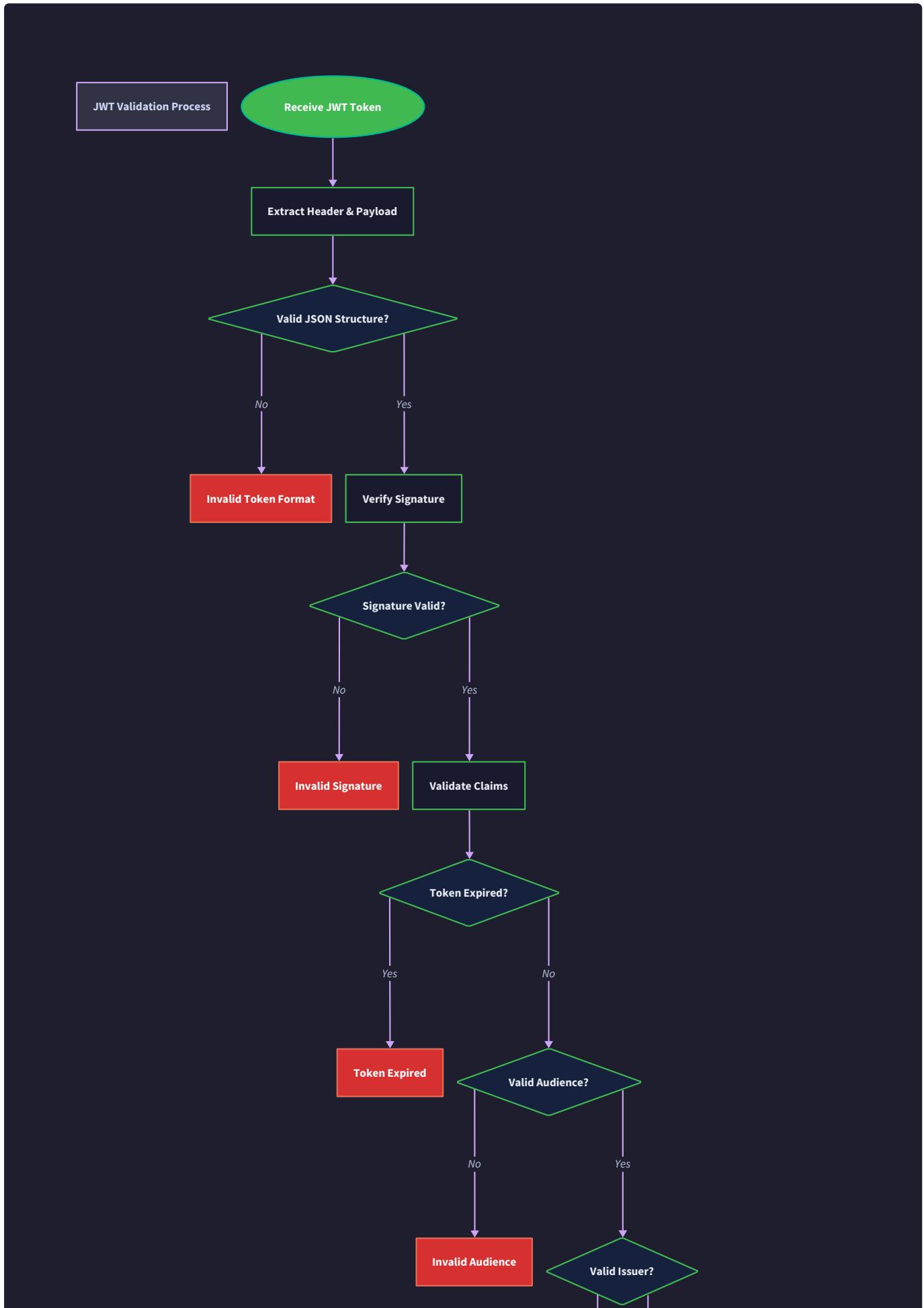
Decision: Require Client Authentication for Introspection Endpoint

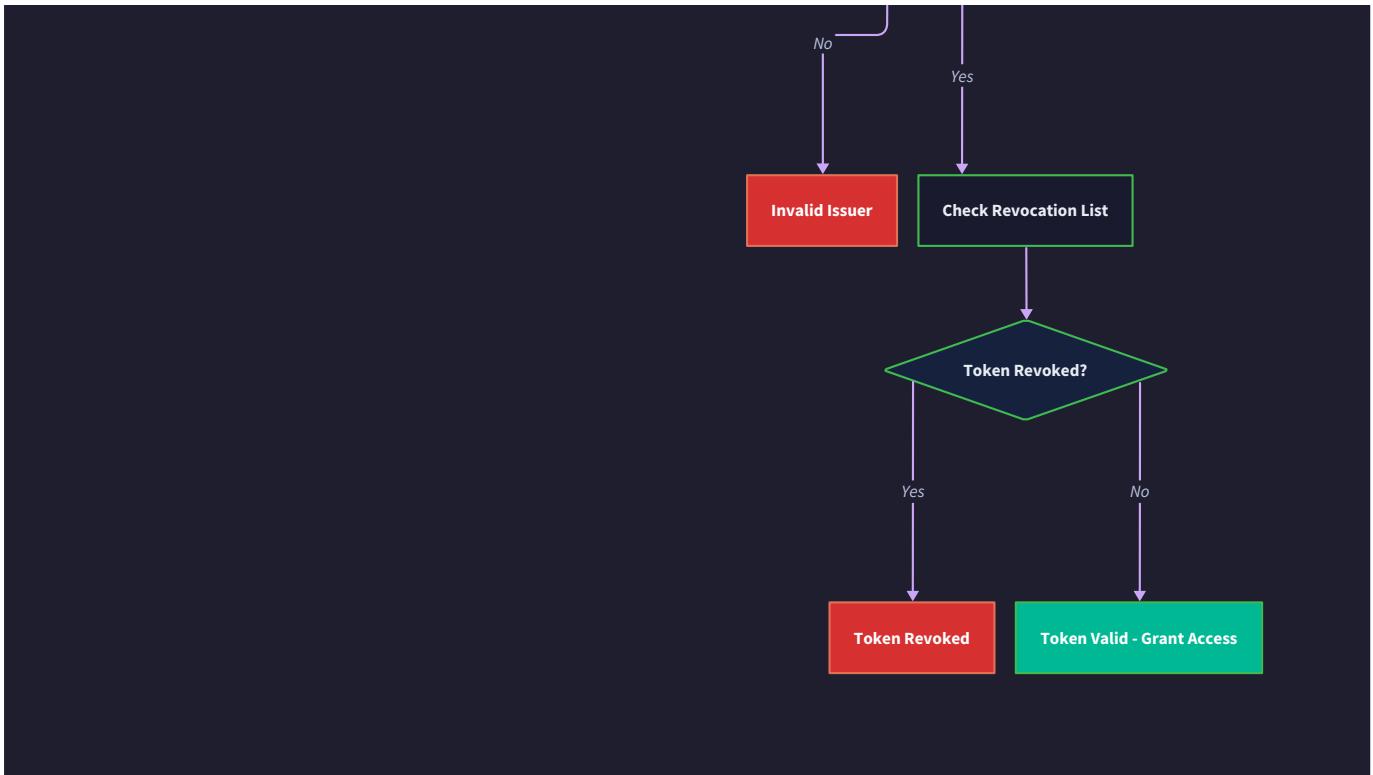
- **Context:** RFC 7662 makes client authentication optional for introspection, but this creates potential security vulnerabilities
- **Options Considered:**
 - Anonymous introspection for simplicity
 - Client authentication required always
 - Configurable authentication requirement
- **Decision:** Always require client authentication for introspection requests
- **Rationale:** Prevents token scanning attacks, enables access control policies, provides audit trail for introspection usage
- **Consequences:** Resource servers must register as OAuth clients, but security posture is significantly improved

Option	Pros	Cons
Anonymous	Simple integration	Enables token scanning attacks
Required Auth	Strong security, audit trail	Complex setup for resource servers
Configurable	Flexible deployment options	Inconsistent security posture









Common Pitfalls

⚠ Pitfall: Not Implementing Revocation List Cleanup

Many implementations create revocation list entries when tokens are revoked but never clean them up, causing the revocation list to grow indefinitely. Since JWT access tokens have limited lifetimes, revocation entries can be safely removed after the token's natural expiration time passes.

Why this breaks: The revocation list grows without bound, consuming memory and storage space. Eventually, the system runs out of resources or performance degrades due to large list sizes. Additionally, keeping expired revocation entries provides no security benefit since the tokens they reference are already naturally invalid.

How to fix: Implement a cleanup process that runs periodically (hourly or daily) and removes revocation entries where the token's expiration time has passed. If using Redis, leverage the TTL feature to automatically expire entries. For database storage, use a background job that deletes old records based on the `expires_at` timestamp.

⚠ Pitfall: Token Scanning Through Unauthenticated Introspection

Allowing unauthenticated access to the introspection endpoint enables attackers to scan randomly generated tokens to determine which ones are valid. Even though the introspection response for invalid tokens only returns `{"active": false}`, the difference in response timing or HTTP status codes can leak information about token validity.

Why this breaks: Attackers can systematically probe token values to discover valid tokens, then use those tokens to access protected resources. This is particularly dangerous if tokens use predictable formats or if the JWT structure reveals patterns that can be exploited in brute force attacks.

How to fix: Always require client authentication for introspection requests. Implement rate limiting per client to prevent rapid scanning attempts. Use constant-time responses for both valid and invalid tokens to prevent timing-based information leakage. Consider implementing CAPTCHA or other anti-automation measures for suspicious request patterns.

⚠ Pitfall: Ignoring Revocation Status in JWT Validation Libraries

When implementing JWT validation libraries for resource servers, developers often focus on cryptographic signature verification and claims validation while forgetting to check revocation status. This creates a security gap where revoked tokens continue to work until their natural expiration.

Why this breaks: Users who revoke permissions or administrators who disable access expect those actions to take immediate effect. If resource servers only validate JWT signatures and claims, revoked tokens remain functional, violating user expectations and security policies. This is particularly problematic for high-privilege tokens or sensitive resources.

How to fix: Always include revocation checking as part of JWT validation. This can be implemented through periodic introspection calls, maintaining a local cache of revoked token IDs, or subscribing to revocation events from the authorization server. Design the validation library to fail securely - if revocation status cannot be determined, treat the token as potentially invalid.

Pitfall: Over-Caching Validation Results Without Revocation Consideration

To improve performance, validation libraries often cache JWT validation results to avoid repeated signature verification. However, long-lived caches can serve stale positive results for tokens that have been revoked, creating a window where revoked tokens remain functional.

Why this breaks: The security benefit of token revocation is negated if cached validation results aren't invalidated when tokens are revoked. Users who revoke permissions expect immediate effect, but cached results can extend token lifetime beyond revocation by hours or even days, depending on cache configuration.

How to fix: Implement cache invalidation mechanisms that respond to revocation events. Use shorter cache TTLs (5-15 minutes maximum) for validation results. Consider implementing a two-tier approach where signature validation is cached longer but revocation status is checked more frequently. Provide cache invalidation APIs that can be called when tokens are revoked.

Pitfall: Inconsistent Token Family Revocation Logic

When implementing refresh token rotation with family tracking, developers sometimes implement inconsistent revocation logic - revoking individual tokens in some cases and entire families in others, without clear criteria for when each approach should be used.

Why this breaks: Inconsistent revocation behavior confuses users and creates security gaps. If replay attack detection revokes entire families but user-initiated revocation only revokes individual tokens, stolen tokens from the same family might remain valid. Conversely, over-aggressive family revocation can log out users unexpectedly from all their devices.

How to fix: Establish clear rules for when individual tokens vs. entire families should be revoked. Typically: revoke entire families for security events (replay detection, breach response), revoke individual tokens for user-initiated actions (single device logout), and provide administrative options for both approaches. Document these behaviors clearly and implement them consistently across all revocation scenarios.

Pitfall: Exposing Sensitive Information in Introspection Responses

The introspection endpoint response format allows returning various claims and metadata about tokens. Developers sometimes include sensitive information like internal user IDs, detailed permission lists, or system-specific metadata that should not be exposed to resource servers.

Why this breaks: Resource servers only need to know whether a token is valid and what scopes it grants. Additional information creates privacy risks and potential attack vectors. Internal user IDs might be used for user enumeration attacks, detailed permissions might reveal system architecture, and system metadata could assist in other attack vectors.

How to fix: Only include necessary information in introspection responses. Standard fields like `scope`, `client_id`, `exp`, and `active` are usually sufficient. If additional claims are needed, carefully evaluate their security implications. Use different response formats for different types of clients - resource servers might receive minimal information while administrative tools get more detailed responses.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Revocation Storage	SQLite with indexed lookups	Redis with database backup
Cache Layer	In-memory Go map with mutex	Redis with pub/sub invalidation
JWT Validation	Standard library crypto/rsa	Dedicated JWT library like golang-jwt
Key Management	Static configuration file	JWKS endpoint with auto-refresh
Rate Limiting	Token bucket in memory	Distributed rate limiter (Redis)
Monitoring	Basic HTTP access logs	Structured logging with metrics

Recommended File Structure

```
internal/introspection/
    introspection.go          ← RFC 7662 introspection endpoint
    introspection_test.go     ← introspection endpoint tests
    validator.go              ← JWT validation library
    validator_test.go         ← validation logic tests
internal/revocation/
    revocation.go            ← RFC 7009 revocation endpoint
    revocation_test.go       ← revocation endpoint tests
    store.go                 ← revocation list storage
    store_test.go             ← storage implementation tests
internal/jwks/
    client.go                ← JWKS key fetching client
    cache.go                 ← Key caching and rotation
    jwks_test.go              ← key management tests
pkg/validation/
    jwt_validator.go          ← Public JWT validation library
    validator_test.go         ← Public library tests
    example_test.go           ← Usage examples for resource servers
```

Infrastructure Starter Code

Revocation Store Interface (Complete Implementation)

```
package revocation

import (
    "context"
    "time"
)

// RevocationRecord represents a revoked token entry

type RevocationRecord struct {

    TokenID      string      `json:"token_id" redis:"token_id"`
    ClientID     string      `json:"client_id" redis:"client_id"`
    RevokedAt    time.Time   `json:"revoked_at" redis:"revoked_at"`
    ExpiresAt    time.Time   `json:"expires_at" redis:"expires_at"`
    TokenType    string      `json:"token_type" redis:"token_type"`
    Reason       string      `json:"reason" redis:"reason"`

}

// RevocationStore provides revocation list operations

type RevocationStore interface {

    // RevokeToken adds a token to the revocation list
    RevokeToken(ctx context.Context, record *RevocationRecord) error

    // IsRevoked checks if a token has been revoked
    IsRevoked(ctx context.Context, tokenID string) (bool, error)

    // RevokeTokenFamily revokes all tokens in a refresh token family
    RevokeTokenFamily(ctx context.Context, familyID string, reason string) error

    // CleanupExpired removes revocation entries for naturally expired tokens
    CleanupExpired(ctx context.Context) (int, error)
}
```

GO

```

// GetRevocationStats returns statistics about the revocation list

GetRevocationStats(ctx context.Context) (*RevocationStats, error)

}

type RevocationStats struct {

    TotalRevoked      int           `json:"total_revoked"`
    ExpiredCleaned   int           `json:"expired_cleaned"`
    OldestEntry       time.Time    `json:"oldest_entry"`
    RevocationsByType map[string]int `json:"revocations_by_type"`
}

// Redis-backed revocation store implementation

type RedisRevocationStore struct {

    client redis.Client
    prefix string
}

func NewRedisRevocationStore(client redis.Client) *RedisRevocationStore {
    return &RedisRevocationStore{
        client: client,
        prefix: "revoked_tokens:",
    }
}

func (r *RedisRevocationStore) RevokeToken(ctx context.Context, record *RevocationRecord) error {
    key := r.prefix + record.TokenID
    ttl := time.Until(record.ExpiresAt)

    // Store the revocation record with TTL matching token expiration
    data, err := json.Marshal(record)
    if err != nil {

```

```
    return fmt.Errorf("marshal revocation record: %w", err)
}

return r.client.Set(ctx, key, data, ttl).Err()
}

func (r *RedisRevocationStore) IsRevoked(ctx context.Context, tokenID string) (bool, error) {
    key := r.prefix + tokenID

    exists, err := r.client.Exists(ctx, key).Result()

    if err != nil {
        return false, fmt.Errorf("check revocation status: %w", err)
    }

    return exists > 0, nil
}
```

JWT Validation Infrastructure (Complete Implementation)

```
package jwks

import (
    "crypto/rsa"
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "time"
)

// JWKSClient fetches and caches JSON Web Key Sets

type JWKSClient struct {

    jwksURL      string
    httpClient   *http.Client
    cache         map[string]*rsa.PublicKey
    cacheMutex   sync.RWMutex
    lastFetch    time.Time
    cacheTTL     time.Duration
}

type JWKSet struct {

    Keys []JWK `json:"keys"`
}

type JWK struct {

    Kid string `json:"kid"`
    Kty string `json:"kty"`
    Use string `json:"use"`
    Alg string `json:"alg"`
    N   string `json:"n"`
    E   string `json:"e"`
}
```

```
}

func NewJWKSClient(jwksURL string, cacheTTL time.Duration) *JWKSClient {
    return &JWKSClient{
        jwksURL:      jwksURL,
        httpClient:   &http.Client{Timeout: 30 * time.Second},
        cache:        make(map[string]*rsa.PublicKey),
        cacheTTL:     cacheTTL,
    }
}

func (j *JWKSClient) GetPublicKey(ctx context.Context, keyID string) (*rsa.PublicKey, error) {
    // Check cache first
    j.cacheMux.RLock()
    if key, exists := j.cache[keyID]; exists && time.Since(j.lastFetch) < j.cacheTTL {
        j.cacheMux.RUnlock()
        return key, nil
    }
    j.cacheMux.RUnlock()

    // Fetch fresh keys
    if err := j.refreshKeys(ctx); err != nil {
        return nil, fmt.Errorf("refresh JWKS: %w", err)
    }

    // Try cache again
    j.cacheMux.RLock()
    defer j.cacheMux.RUnlock()

    key, exists := j.cache[keyID]
```

```
if !exists {

    return nil, fmt.Errorf("key ID %s not found in JWKS", keyID)

}

return key, nil
}

func (j *JWKSClient) refreshKeys(ctx context.Context) error {

    req, err := http.NewRequestWithContext(ctx, "GET", j.jwksURL, nil)

    if err != nil {

        return fmt.Errorf("create request: %w", err)
    }

    resp, err := j.httpClient.Do(req)

    if err != nil {

        return fmt.Errorf("fetch JWKS: %w", err)
    }

    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {

        return fmt.Errorf("JWKS endpoint returned %d", resp.StatusCode)
    }

    var jwkSet JWKSet

    if err := json.NewDecoder(resp.Body).Decode(&jwkSet); err != nil {

        return fmt.Errorf("decode JWKS: %w", err)
    }

    // Parse keys and update cache

    j.cacheMux.Lock()
}
```

```
    defer j.cacheMux.Unlock()

    newCache := make(map[string]*rsa.PublicKey)

    for _, jwk := range jwkSet.Keys {
        if jwk.Kty != "RSA" || jwk.Use != "sig" {
            continue
        }

        publicKey, err := j.parseRSAPublicKey(&jwk)
        if err != nil {
            continue // Skip invalid keys
        }

        newCache[jwk.Kid] = publicKey
    }

    j.cache = newCache
    j.lastFetch = time.Now()

    return nil
}
```

Core Logic Skeleton Code

Introspection Endpoint Implementation

```
// IntrospectionEndpoint handles RFC 7662 token introspection requests

type IntrospectionEndpoint struct {

    clientStore     ClientStore

    revocationStore RevocationStore

    jwtValidator    *JWTValidator

    keyManager      *KeyManager

}

// HandleIntrospection processes token introspection requests per RFC 7662

func (i *IntrospectionEndpoint) HandleIntrospection(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Parse and validate the HTTP request method (must be POST)

    // TODO 2: Extract client credentials from Authorization header or POST body

    // TODO 3: Authenticate the client using ClientStore.GetClient and credential verification

    // TODO 4: Extract the 'token' parameter from the POST body

    // TODO 5: Determine token type (JWT access token vs opaque refresh token)

    // TODO 6: For JWT tokens - validate signature, claims, and expiration

    // TODO 7: Check revocation status using RevocationStore.IsRevoked

    // TODO 8: Build RFC 7662 compliant response with active status and metadata

    // TODO 9: Return JSON response with appropriate HTTP status and cache headers

    // Hint: Always return 200 OK for valid introspection requests, use active:false for invalid
tokens

}

// ValidateJWTToken performs comprehensive JWT validation including revocation checking

func (i *IntrospectionEndpoint) ValidateJWTToken(tokenString string) (*IntrospectionResponse, error) {

    // TODO 1: Parse JWT structure (header.payload.signature)

    // TODO 2: Extract and validate header (alg, typ, kid claims)

    // TODO 3: Get public key for signature verification using KeyManager

    // TODO 4: Verify cryptographic signature using RSA/ECDSA algorithms

    // TODO 5: Parse and validate standard claims (exp, iat, nbf, iss, aud)

    // TODO 6: Extract jti claim for revocation checking
```

```
// TODO 7: Check RevocationStore to see if token has been explicitly revoked  
  
// TODO 8: Extract scope, client_id, and other metadata from JWT claims  
  
// TODO 9: Build IntrospectionResponse with active:true and token metadata  
  
// Hint: Use constant-time comparisons for signature verification to prevent timing attacks  
}
```

Token Revocation Endpoint Implementation

GO

```
// RevocationEndpoint handles RFC 7009 token revocation requests

type RevocationEndpoint struct {

    clientStore     ClientStore

    refreshTokenStore RefreshTokenStore

    revocationStore RevocationStore

    jwtValidator    *JWTValidator

}

// HandleRevocation processes token revocation requests per RFC 7009

func (r *RevocationEndpoint) HandleRevocation(w http.ResponseWriter, req *http.Request) {

    // TODO 1: Validate HTTP method is POST

    // TODO 2: Extract and authenticate client credentials

    // TODO 3: Extract 'token' and optional 'token_type_hint' from POST body

    // TODO 4: Determine actual token type (access token vs refresh token)

    // TODO 5: Validate that client has permission to revoke this specific token

    // TODO 6: For refresh tokens - handle token family revocation logic

    // TODO 7: For access tokens - add JWT ID to revocation list

    // TODO 8: Record revocation event for audit logging

    // TODO 9: Return 200 OK response (success even if token was already invalid)

    // Hint: RFC 7009 requires returning 200 OK even for invalid tokens to prevent information
leakage

}

// RevokeAccessToken adds a JWT access token to the revocation list

func (r *RevocationEndpoint) RevokeAccessToken(token string, clientID string) error {

    // TODO 1: Parse JWT to extract claims without signature verification

    // TODO 2: Extract jti (JWT ID) claim for unique identification

    // TODO 3: Extract exp claim to determine when revocation entry can expire

    // TODO 4: Create RevocationRecord with token ID, client, timestamp, and reason

    // TODO 5: Store revocation record using RevocationStore.RevokeToken

    // TODO 6: Optionally invalidate any validation result caches for this token
```

```
// Hint: Don't verify JWT signature here - we want to revoke even if signature is invalid
}

// RevokeRefreshToken invalidates a refresh token and optionally its family

func (r *RevocationEndpoint) RevokeRefreshToken(tokenString string, clientID string) error {

    // TODO 1: Look up refresh token record using RefreshTokenStore.GetRefreshToken

    // TODO 2: Verify that the client owns this refresh token

    // TODO 3: Check if this is part of a token family (has TokenFamily field)

    // TODO 4: For individual revocation - mark this token as revoked

    // TODO 5: For family revocation - mark all tokens in family as revoked

    // TODO 6: Update database records with revocation timestamp and reason

    // TODO 7: Optionally revoke associated access tokens for security

    // Hint: Token family revocation is a security feature - when in doubt, revoke the whole family

}
```

JWT Validation Library for Resource Servers

GO

```
// JWTValidator provides JWT validation for resource servers

type JWTValidator struct {

    jwksClient      *JWKSClient

    revocationStore RevocationStore

    validationCache map[string]*CachedValidation

    cacheMutex      sync.RWMutex

    issuer          string

    audience        string

}

// ValidateToken performs complete JWT validation including revocation checking

func (v *JWTValidator) ValidateToken(tokenString string, requiredScopes []string) (*ValidationResult, error) {

    // TODO 1: Check validation cache for recent positive results

    // TODO 2: Parse JWT structure and extract header claims (alg, kid)

    // TODO 3: Get appropriate public key from JWKS client

    // TODO 4: Verify cryptographic signature using key and algorithm

    // TODO 5: Parse payload and validate standard claims (exp, iat, nbf, iss, aud)

    // TODO 6: Extract jti claim and check revocation status

    // TODO 7: Validate that granted scopes include all required scopes

    // TODO 8: Cache successful validation result with appropriate TTL

    // TODO 9: Return ValidationResult with user ID, client ID, and granted scopes

    // Hint: Fail fast on any validation step - don't continue processing invalid tokens

}

// CheckRevocationStatus queries whether a token has been explicitly revoked

func (v *JWTValidator) CheckRevocationStatus(jti string) (bool, error) {

    // TODO 1: Extract JWT ID from validated token claims

    // TODO 2: Query RevocationStore.IsRevoked with token ID

    // TODO 3: Handle store errors gracefully (fail secure - assume revoked on error)

    // TODO 4: Cache negative results briefly to reduce store load
```

```

    // TODO 5: Return revocation status boolean

    // Hint: If revocation store is unavailable, consider failing secure vs. failing open based on
    // your security requirements

}

type ValidationResult struct {

    Valid      bool      `json:"valid"`

    UserID     string   `json:"user_id,omitempty"`

    ClientID   string   `json:"client_id,omitempty"`

    Scopes     []string `json:"scopes,omitempty"`

    ExpiresAt  time.Time `json:"expires_at,omitempty"`

    TokenType  string   `json:"token_type,omitempty"`

}

type CachedValidation struct {

    Result      *ValidationResult

    CachedAt   time.Time

    ExpiresAt  time.Time

}

```

Milestone Checkpoint

After implementing this component, verify the following functionality:

Introspection Endpoint Testing:

```
# Test valid token introspection                                                 BASH

curl -X POST http://localhost:8080/oauth2/introspect \
-H "Authorization: Basic $(echo -n 'client_id:client_secret' | base64)" \
-d "token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9..."

# Expected response for valid token:

# {"active":true,"scope":"read write","client_id":"test_client","exp":1234567890}

# Test invalid token introspection

curl -X POST http://localhost:8080/oauth2/introspect \
-H "Authorization: Basic $(echo -n 'client_id:client_secret' | base64)" \
-d "token=invalid_token"

# Expected response for invalid token:

# {"active":false}
```

Revocation Endpoint Testing:

```
# Test token revocation                                                 BASH

curl -X POST http://localhost:8080/oauth2/revoke \
-H "Authorization: Basic $(echo -n 'client_id:client_secret' | base64)" \
-d "token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9..."

# Expected response (should be 200 OK with empty body)

# Verify token is now revoked via introspection

curl -X POST http://localhost:8080/oauth2/introspect \
-H "Authorization: Basic $(echo -n 'client_id:client_secret' | base64)" \
-d "token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9..."

# Expected response (should now show active:false):

# {"active":false}
```

JWT Validation Library Testing:

```
# Test the validation library in a resource server
go test -v ./pkg/validation/ -run TestJWTValidator

# Check that validation correctly identifies:
# - Valid tokens with proper signatures and claims
# - Expired tokens (should fail validation)
# - Revoked tokens (should fail validation)
# - Tokens with invalid signatures (should fail validation)
# - Tokens with missing required scopes (should fail authorization)
```

BASH

Signs Something is Wrong:

- Introspection always returns `active: false` → Check client authentication and JWT parsing
- Revocation doesn't take effect → Verify RevocationStore integration and jti claim extraction
- High CPU usage → JWT signature verification may be inefficient, check caching
- Memory leaks → Validation result cache may not be expiring entries properly
- Tokens work after revocation → RevocationStore queries may be failing silently

UserInfo and Consent Management Component

Milestone(s): Milestone 4 - UserInfo Endpoint & Consent Management

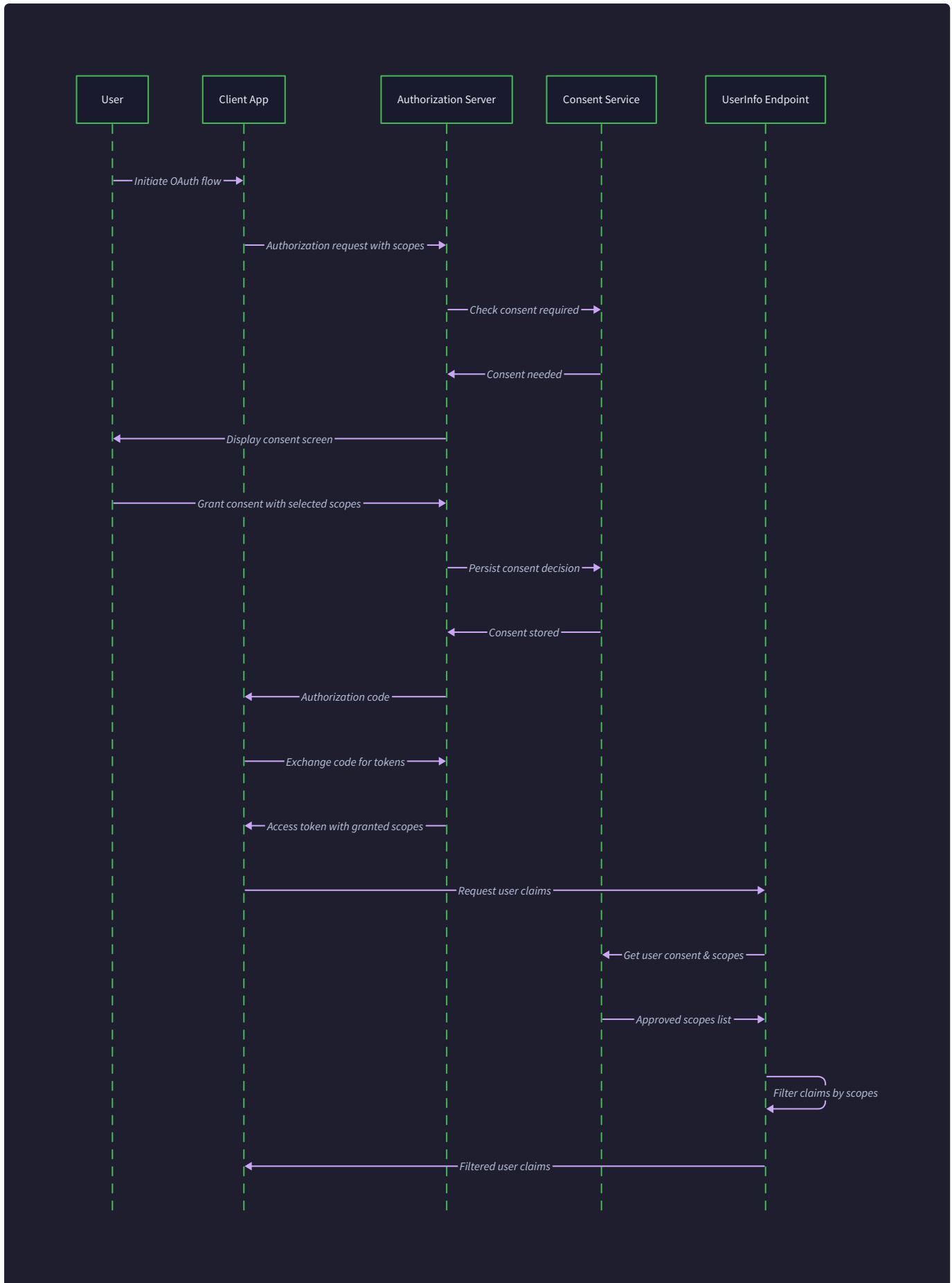
Mental Model: Medical Records Release

Think of the UserInfo endpoint and consent management system like a hospital's medical records release department. When a patient (resource owner) visits a hospital, they accumulate various pieces of medical information - personal details, test results, treatment history, insurance data. Later, when external parties like insurance companies (client applications) or specialists (resource servers) need access to specific portions of this information, they must go through a formal request process.

The patient doesn't give blanket access to their entire medical file. Instead, they carefully review each request and decide which specific types of information can be shared. The insurance company might get access to basic demographics and billing codes, but not detailed psychiatric notes. A consulting specialist might receive test results and treatment history, but not financial information. The medical records department maintains detailed logs of what information was shared with whom and when, and the patient can revoke these permissions at any time.

In our OAuth2 system, the UserInfo endpoint serves as this medical records release department. The `User` profile contains various claims (like medical record fields), the scopes define categories of information that can be requested (like "billing information" or "test results"), and the consent records track what the user has authorized each client application to access. Just as hospitals maintain strict HIPAA compliance by only releasing authorized information, our system enforces scope-based filtering to ensure clients receive only the claims they've been granted permission to access.

This analogy helps explain why consent persistence is crucial - patients shouldn't have to re-authorize the same specialist every time they need a follow-up consultation, but they should be able to revoke access if they switch providers. Similarly, users shouldn't face consent fatigue from repeated authorization prompts, but they must retain control over their data sharing decisions.



UserInfo Endpoint: OIDC Compliant User Profile Claims Service

The **UserInfo endpoint** represents the crown jewel of OpenID Connect functionality - it's where client applications finally retrieve the user profile information they were granted access to during the authorization flow. This endpoint bridges the gap between authorization (what scopes were granted) and actual data access (which specific user claims are returned). The endpoint must implement RFC specifications precisely while maintaining security through proper token validation and scope-based filtering.

The UserInfo endpoint serves a fundamentally different purpose than typical REST APIs. Rather than providing comprehensive user management functionality, it acts as a read-only, scope-filtered window into user profile data. Client applications present their access tokens, and the endpoint returns only those user claims that correspond to the scopes originally granted by the user. This design ensures that even if a client somehow obtains a valid access token, it cannot exceed the permissions the user explicitly authorized.

The endpoint's security model relies entirely on **bearer token authentication** using the access tokens issued by our token endpoint. Unlike traditional API authentication that might use API keys or basic authentication, the UserInfo endpoint exclusively accepts OAuth2 access tokens in the Authorization header. This token-centric approach ensures that every data access request can be traced back to a specific user consent decision and that token revocation immediately terminates data access.

The OIDC specification defines standard claim names and their expected formats, but our implementation must be flexible enough to handle custom claims while maintaining interoperability. Standard claims like `sub` (subject identifier), `name`, `email`, `picture`, and `locale` have defined semantics, but organizations often need to expose additional profile attributes. Our scope-to-claims mapping system accommodates both scenarios while ensuring consistent behavior for standard OIDC flows.

UserInfo Endpoint Method	HTTP Method	Path	Authentication	Description
GetUserInfo	GET	/userinfo	Bearer Token	Returns user claims filtered by granted scopes
GetUserInfo	POST	/userinfo	Bearer Token	Alternative POST method for clients that prefer form data

The endpoint must handle both GET and POST requests to comply with OIDC specifications, though GET requests are more common. POST support accommodates clients that prefer sending the access token in the request body rather than headers, though this approach is less secure and generally discouraged.

Token validation for the UserInfo endpoint involves multiple layers of verification. The endpoint must first extract and validate the JWT access token signature, then verify that the token hasn't expired, hasn't been revoked, and contains the required claims for user identification. Unlike other endpoints that might perform lightweight token checks, UserInfo requires comprehensive validation because it directly exposes user data.

Token Validation Step	Check Performed	Failure Response	Error Code
Token Extraction	Authorization header present and properly formatted	<code>WWW-Authenticate: Bearer</code>	401
JWT Signature	Cryptographic signature verification against public key	<code>invalid_token</code>	401
Token Expiration	Current time before <code>exp</code> claim	<code>invalid_token</code>	401
Revocation Status	Token ID not in revocation list	<code>invalid_token</code>	401
User Existence	User ID from <code>sub</code> claim exists in user store	<code>invalid_token</code>	401
Scope Validation	Token contains valid scope claim	<code>insufficient_scope</code>	403

The endpoint returns user claims as a JSON object, with claim names as keys and claim values following OIDC data type specifications. String claims are returned as JSON strings, boolean claims as JSON booleans, and numeric claims as JSON numbers. Array claims like `groups` or `roles` are returned as JSON arrays. The response must never include claims that weren't authorized by the granted scopes, even if the user profile contains additional data.

Error handling for the UserInfo endpoint follows OIDC specifications rather than general OAuth2 error formats. When token validation fails, the endpoint returns a `401 Unauthorized` status with a `WWW-Authenticate: Bearer` header containing error details. When the token is valid but lacks sufficient scope for the requested operation, the endpoint returns `403 Forbidden` with an `insufficient_scope` error. These specific error responses help client applications distinguish between authentication failures and authorization failures.

Design Insight: The UserInfo endpoint serves as the trust boundary between authorization decisions and actual data access. While the authorization flow establishes what the user consented to share, the UserInfo endpoint enforces those decisions through cryptographic token validation and scope-based filtering. This separation of concerns allows the authorization system to focus on consent workflows while the UserInfo endpoint specializes in secure data delivery.

Performance considerations for the UserInfo endpoint center around token validation overhead and user data retrieval patterns. Since every request requires full JWT validation and potential revocation checking, the endpoint benefits significantly from validation result caching and efficient revocation list lookups. However, caching must be balanced against security requirements - overly aggressive caching can delay the effects of token revocation or user profile updates.

The endpoint must handle high request volumes efficiently since client applications often call UserInfo frequently to refresh user profile data. Connection pooling for database access, Redis caching for revocation lists, and JWT validation result caching become crucial for production performance. However, the endpoint should never cache actual user profile data, as user information changes must be reflected immediately in API responses.

Scope to Claims Mapping: Filtering User Profile Data Based on Granted Scopes

The **scope to claims mapping** system represents the enforcement mechanism that translates high-level user consent decisions into specific data access permissions. When a user grants the `profile` scope to a client application, they're

not authorizing access to a generic "profile" - they're specifically authorizing access to claims like `name`, `family_name`, `given_name`, `picture`, and `locale`. This mapping system ensures that scope grants translate into precise, predictable data access patterns that users can understand and trust.

OIDC defines several standard scopes with specific claim mappings that our implementation must honor for interoperability. The `openid` scope is mandatory for all OIDC flows and doesn't directly map to claims - instead, it signals that the client wants OIDC functionality rather than pure OAuth2. The remaining standard scopes have well-defined claim mappings that client applications rely on for consistent behavior across different identity providers.

OIDC Standard Scope	Associated Claims	Purpose	User-Facing Description
<code>openid</code>	None (enables OIDC)	Signals OIDC request	"Confirm your identity"
<code>profile</code>	<code>name</code> , <code>family_name</code> , <code>given_name</code> , <code>middle_name</code> , <code>nickname</code> , <code>preferred_username</code> , <code>profile</code> , <code>picture</code> , <code>website</code> , <code>gender</code> , <code>birthdate</code> , <code>zoneinfo</code> , <code>locale</code> , <code>updated_at</code>	Basic profile information	"Access your basic profile information"
<code>email</code>	<code>email</code> , <code>email_verified</code>	Email address and verification status	"Access your email address"
<code>address</code>	<code>address</code> (structured object)	Physical address information	"Access your address information"
<code>phone</code>	<code>phone_number</code> , <code>phone_number_verified</code>	Phone number and verification status	"Access your phone number"

Beyond standard OIDC scopes, organizations often need custom scopes for application-specific data access. Our mapping system must accommodate custom scopes while maintaining the security properties of the standard mappings. Custom scopes like `employee_info`, `billing_data`, or `medical_records` can map to organization-specific user profile fields, but they must follow the same filtering principles as standard scopes.

The mapping implementation must handle **overlapping scopes** correctly when clients request multiple scopes simultaneously. If a client requests both `profile` and `email` scopes, the resulting claim set includes the union of claims from both mappings, not their intersection. This additive behavior matches user expectations - granting additional scopes provides additional access rather than restricting it.

Architecture Decision: Static vs. Dynamic Scope Mapping

- **Context:** We need to decide whether scope-to-claims mappings should be hardcoded in application logic or stored in a configurable database table
- **Options Considered:**
 1. Static mappings in Go constants/maps for standard scopes only
 2. Database-driven mappings for all scopes with admin UI
 3. Hybrid approach with static standard mappings and configurable custom mappings
- **Decision:** Hybrid approach with static standard mappings and configurable custom mappings
- **Rationale:** Standard OIDC scope mappings never change and benefit from compile-time validation, while custom organizational scopes need runtime flexibility. Hybrid approach provides both reliability and flexibility.
- **Consequences:** Enables OIDC compliance through guaranteed standard mappings while supporting organizational customization. Requires careful validation to prevent conflicts between standard and custom scope names.

Claim filtering must handle **conditional claims** where the availability of certain claims depends on user profile completeness or verification status. The `email_verified` claim should only be included when the user's email has been verified, and the claim value should reflect the actual verification status. Similarly, optional profile fields like `middle_name` or `website` should be omitted entirely when not present in the user profile, rather than returned with null values.

Claim Filtering Rule	Condition	Behavior	Example
Standard Claim Present	User profile contains claim value	Include claim in response	<code>"name": "John Smith"</code>
Standard Claim Missing	User profile lacks claim value	Omit claim from response	<code>middle_name</code> not included
Verification Claim	Depends on verification status	Include with boolean value	<code>"email_verified": true</code>
Scope Not Granted	Scope not in access token	Omit all associated claims	Email claims omitted without <code>email</code> scope
Custom Claim Present	Custom scope granted and data exists	Include claim in response	<code>"employee_id": "EMP123"</code>

The filtering algorithm must process scopes in a specific order to ensure consistent results across different request patterns. First, extract all granted scopes from the access token's `scope` claim and validate that each scope is recognized by the system. Then, build the union of all claim names associated with the granted scopes. Finally, retrieve user profile data and include only those claims that appear in both the authorized claim set and the user's actual profile data.

Error handling in scope-to-claims mapping focuses on graceful degradation rather than request failure. If a custom scope refers to a non-existent claim field, the system should log the configuration error but continue processing other valid

claims. If the user profile is missing optional standard claims, the response simply omits those claims. Only severe errors like database connectivity failures or access token corruption should result in request failure.

The mapping system must support **nested claims** for complex data structures like addresses. The OIDC `address` scope maps to a single `address` claim containing a structured object with fields like `street_address`, `locality`, `region`, `postal_code`, and `country`. Our implementation must handle these nested structures correctly while applying the same filtering principles to sub-claims within the nested object.

Design Insight: Scope-to-claims mapping serves as a privacy-by-design mechanism that ensures users receive granular control over their data sharing. By mapping abstract scope names to specific claim sets, users can make informed consent decisions without needing to understand technical details about API fields or database schemas. This abstraction layer protects user privacy while providing developers with predictable data access patterns.

Performance optimization for scope mapping centers around **mapping cache efficiency** and **user data retrieval patterns**. Since scope-to-claims mappings change rarely (especially standard OIDC mappings), aggressive caching provides significant performance benefits. However, custom scope mappings may change more frequently and require cache invalidation strategies. The system should cache mapping results at the application level while maintaining the ability to refresh mappings without service restart.

Consent Persistence and Management: Storing and Managing User Authorization Decisions

Consent persistence transforms ephemeral user authorization decisions into durable access control policies that can be enforced across multiple API interactions. When a user clicks "Allow" on a consent screen, that decision must be recorded, indexed, tracked, and made revocable to provide users with meaningful control over their data sharing. The consent management system bridges the gap between one-time authorization flows and ongoing data access patterns.

The `ConsentRecord` data model captures the essential elements of a user's authorization decision in a way that supports both access control enforcement and user privacy management. Each consent record represents a specific user's decision to grant specific scopes to a specific client application at a specific point in time. This granular approach allows users to make different decisions for different applications while maintaining precise audit trails.

ConsentRecord Field	Type	Purpose	Example Value
<code>UserID</code>	string	Links consent to specific user	"user_123456"
<code>ClientID</code>	string	Links consent to specific client application	"client_abc789"
<code>Scope</code>	string	Space-separated list of granted scopes	"openid profile email"
<code>GrantedAt</code>	<code>time.Time</code>	When user initially granted consent	"2024-01-15T10:30:00Z"
<code>LastUsedAt</code>	<code>*time.Time</code>	Most recent access using this consent	"2024-01-15T14:22:00Z"
<code>ExpiresAt</code>	<code>*time.Time</code>	Optional consent expiration time	"2024-07-15T10:30:00Z"
<code>Revoked</code>	bool	Whether user has revoked this consent	false
<code>RevokedAt</code>	<code>*time.Time</code>	When consent was revoked (if applicable)	null

Consent record storage must support **efficient lookups** for multiple access patterns. During authorization flows, the system needs to quickly determine whether a user has previously consented to specific scopes for a specific client.

During UserInfo requests, the system needs to validate that the current access token corresponds to valid, non-revoked consent. During user privacy management, the system needs to retrieve all consent records for a specific user across all client applications.

The consent checking algorithm runs during authorization flows to determine whether the consent screen should be displayed or skipped. If a user is requesting scopes they've previously granted to the same client application, and that consent hasn't been revoked or expired, the system can skip the consent screen and proceed directly to authorization code generation. However, if the client requests any new scopes beyond those previously granted, the consent screen must be displayed to capture the user's decision on the additional permissions.

Consent Check Scenario	Existing Consent	New Request	Behavior	Rationale
Exact Match	openid profile email	openid profile email	Skip consent screen	User already authorized these exact scopes
Subset Request	openid profile email	openid profile	Skip consent screen	Requesting fewer permissions than previously granted
Superset Request	openid profile	openid profile email	Show consent screen	New email scope requires additional consent
Partial Overlap	openid profile	openid email	Show consent screen	Mixed of old and new scopes requires re-consent
No Previous Consent	None	openid profile	Show consent screen	First-time authorization requires consent

Consent expiration provides a mechanism for implementing organizational policies around consent freshness while respecting user privacy preferences. Some organizations require re-consent every 12 months for compliance reasons, while others prefer persistent consent until explicitly revoked. Our system supports optional expiration through the `ExpiresAt` field, allowing administrators to configure consent lifetime policies per client application or globally.

Consent revocation must be **immediately effective** across all system components. When a user revokes consent for a client application, several things must happen atomically: the consent record is marked as revoked, any active refresh tokens for that client/user combination are invalidated, and future authorization requests for that client must display the consent screen regardless of previous consent history. This immediate revocation ensures that users maintain meaningful control over their data sharing.

The consent management interface provides users with visibility and control over their authorization decisions. Users must be able to view all client applications they've authorized, see what scopes each application was granted, review when each authorization occurred and was last used, and revoke consent for any application at any time. This self-service approach empowers users to manage their privacy without requiring administrator intervention.

User Consent Management Action	Database Operation	Side Effects	User Feedback
View All Consents	Query consent records by UserID	None	Display list of authorized applications
View Consent Details	Query specific consent record	None	Show scopes, dates, usage information
Revoke Single Consent	Update consent record, set Revoked=true	Invalidate related refresh tokens	Confirm revocation successful
Revoke All Consents	Update all user's consent records	Invalidate all user's refresh tokens	Confirm bulk revocation successful
Download Consent History	Query and export consent records	Generate audit report	Provide downloadable report

Architecture Decision: Consent Record Granularity

- **Context:** We must decide whether to store one consent record per user-client pair or separate records for each scope grant
- **Options Considered:**
 1. Single record per user-client with comma-separated scopes
 2. Separate record for each user-client-scope combination
 3. Single record per user-client with JSON array of scope objects
- **Decision:** Single record per user-client with space-separated scopes matching OAuth2 format
- **Rationale:** Matches OAuth2 scope parameter format, simplifies consent checking logic, reduces database rows while maintaining queryability. Scope changes for same client update existing record rather than creating new records.
- **Consequences:** Enables efficient consent checking and revocation. Requires careful handling of scope additions/removals. Audit trail captured through updated_at timestamps rather than separate records.

Consent analytics help organizations understand user adoption patterns and identify potential friction points in authorization flows. Aggregate metrics like consent grant rates, revocation patterns, and scope popularity can inform UX improvements and policy decisions. However, these analytics must be implemented carefully to avoid creating privacy risks through detailed user behavior tracking.

The consent persistence layer must handle **concurrent modifications** gracefully when multiple authorization flows occur simultaneously for the same user-client combination. Database-level constraints and optimistic locking prevent race conditions that could result in inconsistent consent states. If two authorization flows attempt to create consent records simultaneously, one should succeed while the other should detect the existing consent and proceed accordingly.

Integration with the authorization endpoint requires careful coordination between consent checking, consent screen display, and consent record creation. The authorization flow must atomically create or update consent records during successful authorization to ensure that subsequent requests can benefit from the stored consent decision. Failure to persist consent decisions results in users facing repeated consent prompts, creating poor user experience and consent fatigue.

Design Insight: Consent persistence transforms user authorization from a one-time authentication event into an ongoing privacy management relationship. By storing and making consent decisions visible and revocable, the system shifts from a "ask once, access forever" model to a "transparent, controllable access" model that respects user autonomy and supports evolving privacy requirements.

Architecture Decision Records

Decision: UserInfo Response Caching Strategy

- **Context:** UserInfo endpoint may receive high request volumes and user profile data changes relatively infrequently, suggesting potential benefits from response caching. However, caching user data raises privacy concerns and could delay reflection of profile updates or consent revocations.
- **Options Considered:**
 1. No caching - fetch user data fresh for every request
 2. Aggressive caching with 15-minute TTL on complete UserInfo responses
 3. Cache user profile data only, validate tokens and consent fresh
 4. Cache validation results only, fetch user data fresh
- **Decision:** Cache validation results only with 2-minute TTL, fetch user data fresh for each request
- **Rationale:** Token validation (JWT signature, revocation checking) is computationally expensive and can benefit from short-term caching without privacy implications. User profile data must be fresh to reflect immediate updates and consent changes. 2-minute TTL balances performance with security responsiveness.
- **Consequences:** Provides 80% of caching performance benefits while maintaining data freshness. Requires cache invalidation on token revocation. May still face database load from user profile queries but eliminates cryptographic validation overhead.

Caching Option	Performance Gain	Privacy Risk	Data Freshness	Complexity
No Caching	None	None	Immediate	Low
Full Response Cache	High	High	Delayed	Medium
Profile Data Only	Medium	Medium	Delayed	Medium
Validation Only	Medium	Low	Immediate	Medium

Decision: Scope Addition Consent Handling

- **Context:** When client applications request additional scopes beyond those previously granted, we must decide whether to show a consent screen for only the new scopes or re-consent for all requested scopes including previously granted ones.
- **Options Considered:**
 1. Incremental consent - show consent screen only for new scopes
 2. Full re-consent - show all requested scopes including previously granted ones
 3. Hybrid - show all scopes but pre-check previously granted ones
- **Decision:** Incremental consent showing only new scopes with clear indication of previously granted scopes
- **Rationale:** Reduces consent fatigue by not re-prompting for already-granted permissions. Users can review existing permissions through separate privacy dashboard. Clearly communicates what new access is being requested. Matches user mental model of "additional permissions".
- **Consequences:** Enables smoother user experience for scope expansion. Requires more complex consent screen UI to show both new and existing grants. Consent records must track scope additions over time rather than replacing previous grants.

Decision: Custom Claims Namespace Management

- **Context:** Organizations need custom user profile claims beyond standard OIDC claims, but custom claims could conflict with future OIDC standards or between different organizational systems.
- **Options Considered:**
 1. No restrictions - allow any custom claim names
 2. Mandatory namespace prefix for all custom claims (e.g., "org:employee_id")
 3. Reserved namespace for standard claims, unrestricted custom claims
 4. Configuration-driven claim validation with conflict detection
- **Decision:** Reserved namespace for standard claims with unrestricted custom claims, plus runtime conflict detection
- **Rationale:** Preserves OIDC interoperability by protecting standard claim names. Allows organizational flexibility without imposing naming overhead. Runtime validation catches conflicts before they impact production. Enables gradual migration if new OIDC standards introduce conflicting names.
- **Consequences:** Prevents accidental override of standard OIDC claims. Requires validation logic to detect conflicts between custom and standard claims. May require claim name migration if future OIDC versions conflict with existing custom claims.

Custom Claims Approach	Interoperability	Flexibility	Migration Risk	Implementation Complexity
No Restrictions	Low	High	High	Low
Mandatory Prefixes	High	Medium	Low	Medium
Reserved Standard	High	High	Medium	Medium
Full Validation	High	High	Low	High

Decision: Consent Record Database Schema Design

- **Context:** Consent records need to support efficient lookups for authorization flows, user privacy management, and audit requirements while handling scope changes and revocation scenarios.
- **Options Considered:**
 1. Single table with composite primary key (user_id, client_id)
 2. Separate tables for active consent and consent history
 3. Event sourcing with consent granted/revoked events
 4. Hybrid current state table plus audit log table
- **Decision:** Single table with composite primary key plus separate audit log table for compliance
- **Rationale:** Simple schema supports all required query patterns efficiently. Composite key ensures one active consent per user-client pair. Separate audit table provides compliance reporting without impacting operational queries. Balances simplicity with audit requirements.
- **Consequences:** Enables fast consent checking during authorization flows. Requires careful handling of scope updates to maintain data consistency. Audit table provides compliance trail without performance impact. May require data migration for schema changes.

Common Pitfalls

⚠ Pitfall: Including Claims Without Scope Authorization

One of the most dangerous mistakes in UserInfo endpoint implementation is returning user claims that weren't authorized by the granted scopes. Developers might implement a simple "return all user profile data" approach without properly filtering based on access token scopes, inadvertently exposing sensitive information that users never consented to share.

This mistake typically occurs when developers focus on JWT validation but skip the crucial scope-to-claims mapping step. The access token might be cryptographically valid and non-expired, but if it only contains the `openid profile` scopes, returning email addresses or phone numbers violates the user's consent decision and potentially breaches privacy regulations.

How to avoid: Always implement scope-based filtering as a mandatory step in UserInfo response generation. Start with an empty claims object, then add only those claims that correspond to granted scopes. Never use user profile data as the starting point and filter out unauthorized claims - this approach is error-prone and may accidentally leak data during implementation changes.

⚠ Pitfall: Caching UserInfo Responses Without Considering Consent Revocation

Aggressive caching of UserInfo responses can create a significant security vulnerability where revoked consent doesn't take effect immediately. If the system caches complete UserInfo responses for performance, a user's consent revocation might not prevent continued data access until the cache expires, potentially violating the user's privacy expectations and regulatory requirements.

This problem is particularly severe when cache TTL values are set too high or when cache invalidation doesn't account for consent changes. A user might revoke a client's access and expect immediate effect, but cached responses could continue serving their profile data for minutes or hours after revocation.

How to avoid: Never cache complete UserInfo responses containing actual user profile data. Instead, cache only the expensive operations like JWT validation and revocation checking. Implement cache invalidation triggers that immediately

clear relevant cached data when consent records are modified or tokens are revoked.

⚠ Pitfall: Inconsistent Consent Checking Logic Between Authorization and UserInfo

Implementing different consent validation logic in the authorization endpoint versus the UserInfo endpoint creates subtle security gaps where tokens issued under one set of rules are validated under different rules. This inconsistency might allow access tokens to be used for data access even when the underlying consent has been modified or revoked.

The problem typically manifests when the authorization endpoint creates consent records in one format but the UserInfo endpoint expects a different format, or when consent checking algorithms diverge between components due to separate development or different developers implementing different parts of the system.

How to avoid: Centralize consent checking logic in a shared service that both the authorization endpoint and UserInfo endpoint use. Ensure that consent record creation, validation, and revocation follow identical business rules across all system components. Implement comprehensive integration tests that verify consistent behavior between authorization and data access flows.

⚠ Pitfall: Exposing Internal User IDs in UserInfo Responses

The OIDC `sub` claim should contain a **stable, unique identifier** for the user that's specific to the client application, not the internal database primary key or user ID used within the identity provider system. Exposing internal user IDs creates privacy risks and potential security vulnerabilities if those IDs are used in other system contexts.

Many developers mistakenly use the same user ID across all client applications, which enables cross-application user correlation and violates privacy principles. Different client applications should receive different `sub` claim values for the same user to prevent tracking across applications.

How to avoid: Generate client-specific subject identifiers using a one-way hash of the internal user ID and client ID. This approach ensures that each client receives a unique, stable identifier for each user while preventing cross-client correlation. Store these pairwise pseudonymous identifiers if consistent values are required across multiple token issuances.

⚠ Pitfall: Failing to Handle Missing or Unverified Claims Appropriately

When user profile data is incomplete or unverified, the UserInfo endpoint must handle missing claims correctly according to OIDC specifications. Simply returning null values or empty strings for missing claims can break client application logic that expects either valid data or complete omission of the claim.

Verification status claims like `email_verified` must reflect actual verification state rather than defaulting to false or being omitted when verification status is unknown. Client applications rely on these verification claims to make security decisions about user authentication strength.

How to avoid: Omit claims entirely from UserInfo responses when the corresponding user profile data is missing or null. For verification claims, include them only when verification status is definitively known, and set the value to reflect actual verification state. Document clearly which claims are optional and may be omitted based on user profile completeness.

⚠ Pitfall: Inadequate Consent Screen Information for User Decision Making

Consent screens that display only scope names like "profile" and "email" without explaining what specific information will be shared leave users unable to make informed decisions about their data sharing. Users can't meaningfully consent when they don't understand what they're authorizing.

Generic consent screens that don't identify the specific client application requesting access or that fail to explain how the data will be used prevent users from making contextual privacy decisions. The same scopes might be appropriate for one application but inappropriate for another based on the application's purpose and trustworthiness.

How to avoid: Design consent screens that clearly identify the requesting client application, explain in plain language what information each scope provides access to, and when possible, include information about how the client intends to use the data. Provide scope descriptions that are meaningful to non-technical users, such as "Access your name and profile picture" rather than just "profile scope".

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
UserInfo HTTP Handler	<code>net/http</code> with manual routing	<code>gin-gonic/gin</code> or <code>gorilla/mux</code> for structured routing
Claims Filtering	Map-based filtering with manual loops	<code>reflect</code> package for dynamic claim processing
Consent Storage	PostgreSQL with <code>database/sql</code>	<code>gorm.io/gorm</code> ORM with relationship modeling
Response Caching	In-memory <code>sync.Map</code>	Redis with <code>go-redis/redis</code> client
User Profile Storage	Same database as consent records	Separate user service with HTTP/gRPC interface

Recommended File Structure

```
project-root/
├── cmd/server/main.go           ← server entry point
├── internal/
│   ├── userinfo/
│   │   ├── userinfo.go          ← UserInfo endpoint component
│   │   ├── claims.go            ← main UserInfo handler
│   │   ├── userinfo_test.go     ← scope-to-claims mapping logic
│   │   └── claims_test.go       ← endpoint tests
│   ├── consent/
│   │   ├── consent.go           ← consent management component
│   │   ├── store.go              ← consent persistence service
│   │   ├── screen.go             ← consent database operations
│   │   └── consent_test.go      ← consent screen handler
│   ├── users/
│   │   ├── user.go               ← user profile management
│   │   ├── store.go              ← user data structures
│   │   └── user_test.go          ← user database operations
│   └── validation/
│       ├── tokens.go            ← user management tests
│       └── scopes.go             ← shared validation utilities
│           ├── tokens.go          ← JWT validation from previous milestone
│           └── scopes.go           ← scope validation utilities
└── web/
    ├── templates/
    │   ├── consent.html          ← consent screen template
    │   └── privacy.html           ← user privacy dashboard
    └── static/
        ├── css/consent.css        ← consent screen styling
        └── js/consent.js            ← consent screen interactions
└── migrations/
    ├── 001_create_users.sql      ← user profile schema
    └── 002_create_consent_records.sql  ← consent persistence schema
```

Infrastructure Starter Code

User Profile Data Structures (users/user.go):

```
package users

import (
    "context"
    "database/sql"
    "time"
)

// User represents a complete user profile with all OIDC standard claims

type User struct {

    UserID      string      `json:"sub" db:"user_id"`
    Username    string      `json:"preferred_username" db:"username"`
    Email       string      `json:"email" db:"email"`
    EmailVerified bool       `json:"email_verified" db:"email_verified"`
    PasswordHash string     `json:"-" db:"password_hash"`
    GivenName   string      `json:"given_name" db:"given_name"`
    FamilyName  string      `json:"family_name" db:"family_name"`
    Picture     string      `json:"picture" db:"picture"`
    Locale      string      `json:"locale" db:"locale"`
    CreatedAt   time.Time   `json:"-" db:"created_at"`
    UpdatedAt   time.Time   `json:"-" db:"updated_at"`
    Active      bool        `json:"-" db:"active"`
}

// Name returns the full name by combining given and family names

func (u *User) Name() string {
    if u.GivenName != "" && u.FamilyName != "" {
        return u.GivenName + " " + u.FamilyName
    }
    if u.GivenName != "" {
        return u.GivenName
    }
}
```

GO

```
}

if u.FamilyName != "" {

    return u.FamilyName

}

return u.Username

}

// UserStore interface defines user profile persistence operations

type UserStore interface {

    GetUser(ctx context.Context, userID string) (*User, error)

    GetUserByEmail(ctx context.Context, email string) (*User, error)

    CreateUser(ctx context.Context, user *User) error

    UpdateUser(ctx context.Context, user *User) error

}

// PostgresUserStore implements UserStore using PostgreSQL

type PostgresUserStore struct {

    db *sql.DB

}

func NewUserStore(db *sql.DB) *PostgresUserStore {

    return &PostgresUserStore{db: db}

}

func (s *PostgresUserStore) GetUser(ctx context.Context, userID string) (*User, error) {

    user := &User{}

    query := `SELECT user_id, username, email, email_verified, password_hash,

                given_name, family_name, picture, locale,

                created_at, updated_at, active

            FROM users WHERE user_id = $1 AND active = true`
```

```
err := s.db.QueryRowContext(ctx, query, userID).Scan(  
    &user.UserID, &user.Username, &user.Email, &user.EmailVerified,  
    &user.PasswordHash, &user.GivenName, &user.FamilyName,  
    &user.Picture, &user.Locale, &user.CreatedAt, &userUpdatedAt,  
    &user.Active,  
)  
  
if err != nil {  
    return nil, err  
}  
  
return user, nil  
}  
  
// Additional UserStore methods would be implemented here...
```

Consent Management Data Structures (consent/consent.go):

```
package consent

import (
    "context"
    "database/sql"
    "time"
)

// ConsentRecord represents a user's authorization decision for a client

type ConsentRecord struct {

    UserID      string      `json:"user_id" db:"user_id"`
    ClientID   string      `json:"client_id" db:"client_id"`
    Scope       string      `json:"scope" db:"scope"`
    GrantedAt  time.Time   `json:"granted_at" db:"granted_at"`
    LastUsedAt *time.Time  `json:"last_used_at" db:"last_used_at"`
    ExpiresAt  *time.Time  `json:"expires_at" db:"expires_at"`
    Revoked    bool        `json:"revoked" db:"revoked"`
    RevokedAt  *time.Time  `json:"revoked_at" db:"revoked_at"`
}

// IsValid returns true if the consent is currently valid and active

func (c *ConsentRecord) IsValid() bool {

    if c.Revoked {

        return false
    }

    if c.ExpiresAt != nil && time.Now().After(*c.ExpiresAt) {

        return false
    }

    return true
}
```

GO

```
}

// ConsentStore interface defines consent record persistence operations

type ConsentStore interface {

    GetConsent(ctx context.Context, userID, clientID string) (*ConsentRecord, error)

    GetUserConsents(ctx context.Context, userID string) ([]*ConsentRecord, error)

    CreateConsent(ctx context.Context, consent *ConsentRecord) error

    UpdateConsent(ctx context.Context, consent *ConsentRecord) error

    RevokeConsent(ctx context.Context, userID, clientID string) error

    RevokeAllUserConsents(ctx context.Context, userID string) error

}

// PostgresConsentStore implements ConsentStore using PostgreSQL

type PostgresConsentStore struct {

    db *sql.DB
}

func NewConsentStore(db *sql.DB) *PostgresConsentStore {

    return &PostgresConsentStore{db: db}
}

// GetConsent retrieves the current consent record for a user-client pair

func (s *PostgresConsentStore) GetConsent(ctx context.Context, userID, clientID string) (*ConsentRecord, error) {

    consent := &ConsentRecord{}

    query := `SELECT user_id, client_id, scope, granted_at, last_used_at,
               expires_at, revoked, revoked_at
              FROM consent_records
             WHERE user_id = $1 AND client_id = $2`


    err := s.db.QueryRowContext(ctx, query, userID, clientID).Scan(
        &consent.UserID, &consent.ClientID, &consent.Scope,
```

```
    &consent.GrantedAt, &consent.LastUsedAt, &consent.ExpiresAt,  
    &consent.Revoked, &consent.RevokedAt,  
)  
  
if err != nil {  
    return nil, err  
}  
  
return consent, nil  
}  
  
// Additional ConsentStore methods would be implemented here...
```

Core Logic Skeleton Code

UserInfo Endpoint Handler (userinfo/userinfo.go):

```
package userinfo

import (
    "context"
    "encoding/json"
    "net/http"
    "strings"

    "your-project/internal/consent"
    "your-project/internal/users"
    "your-project/internal/validation"
)

// UserInfoEndpoint handles OIDC UserInfo requests

type UserInfoEndpoint struct {
    userStore     users.UserStore
    consentStore  consent.ConsentStore
    validator     *validation.JWTValidator
    claimsMapper *ClaimsMapper
}

func NewUserInfoEndpoint(userStore users.UserStore, consentStore consent.ConsentStore,
    validator *validation.JWTValidator) *UserInfoEndpoint {
    return &UserInfoEndpoint{
        userStore:     userStore,
        consentStore: consentStore,
        validator:     validator,
        claimsMapper: NewClaimsMapper(),
    }
}

// HandleUserInfo processes GET and POST requests to the UserInfo endpoint
```

GO

```
func (u *UserInfoEndpoint) HandleUserInfo(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Extract access token from Authorization header or request body

    //           Support both "Bearer token" header format and form-encoded body

    //           Return 401 with WWW-Authenticate header if token missing


    // TODO 2: Validate the JWT access token using JWTValidator

    //           Check signature, expiration, and revocation status

    //           Return 401 with invalid_token error if validation fails


    // TODO 3: Extract user ID and scopes from validated token claims

    //           Get "sub" claim for user identification

    //           Get "scope" claim and split into individual scopes

    //           Return 500 if required claims missing from valid token


    // TODO 4: Retrieve user profile from UserStore

    //           Use user ID from token to fetch complete user record

    //           Return 401 if user not found or inactive

    //           Log warning if user exists in token but not in database


    // TODO 5: Filter user claims based on granted scopes

    //           Use ClaimsMapper to determine which claims each scope authorizes

    //           Build response object containing only authorized claims

    //           Omit claims entirely if corresponding user data is missing


    // TODO 6: Return filtered user claims as JSON response

    //           Set Content-Type: application/json header

    //           Use 200 status code for successful responses

    //           Handle JSON marshaling errors appropriately
```

```

// Hint: Cache validation results for performance but never cache user data

// Hint: Use consistent error response format matching OAuth2 specifications

}

// extractAccessToken gets the access token from request headers or body

func (u *UserInfoEndpoint) extractAccessToken(r *http.Request) (string, error) {

    // TODO: Handle both Authorization header and form body token presentation

    return "", nil
}

// writeUserInfoError sends standardized OAuth2 error responses for UserInfo endpoint

func (u *UserInfoEndpoint) writeUserInfoError(w http.ResponseWriter, statusCode int,
                                             errorCode, description string) {

    // TODO: Format error response according to OIDC UserInfo error specifications

    //      Include WWW-Authenticate header for 401 responses

    //      Use application/json content type for error responses

}

```

Claims Mapping Logic (userinfo/claims.go):

```
package userinfo

import (
    "strings"

    "your-project/internal/users"
)

// ClaimsMapper handles scope-to-claims mapping and filtering

type ClaimsMapper struct {

    scopeMappings map[string][]string
}

func NewClaimsMapper() *ClaimsMapper {
    return &ClaimsMapper{
        scopeMappings: map[string][]string{
            "profile": {"name", "given_name", "family_name", "middle_name",
                        "nickname", "preferred_username", "profile", "picture",
                        "website", "gender", "birthdate", "zoneinfo", "locale",
                        "updated_at"},

            "email": {"email", "email_verified"},

            "address": {"address"},

            "phone": {"phone_number", "phone_number_verified"},

        },
    }
}

// FilterClaims returns user claims filtered by granted scopes

func (c *ClaimsMapper) FilterClaims(user *users.User, grantedScopes []string) map[string]interface{} {
    // TODO 1: Build set of authorized claim names from granted scopes
    //           Iterate through grantedScopes and look up associated claims
}
```

```

//      Create union of all claim names from all granted scopes

//      Skip "openid" scope as it doesn't map to specific claims


// TODO 2: Convert user struct to map of all possible claims

//      Include computed claims like "name" (given_name + family_name)

//      Handle verification status claims like "email_verified"

//      Use JSON tags from User struct for claim names


// TODO 3: Filter user claims map to include only authorized claims

//      Iterate through authorized claim names from step 1

//      Include claim in response only if present in user data

//      Omit claims with empty/nil values rather than including them


// TODO 4: Handle nested claims like address objects

//      If address scope granted, build structured address object

//      Include sub-fields like street_address, locality, region

//      Omit entire nested object if no address data available


// TODO 5: Return filtered claims map ready for JSON serialization

//      Ensure all claim values use correct JSON types

//      Boolean claims should be JSON booleans, not strings

//      Timestamp claims should be formatted as UNIX timestamps or ISO 8601


// Hint: Always include "sub" claim with user ID regardless of scopes

// Hint: Custom scopes can be added to scopeMappings map dynamically

return nil

}

// GetAuthorizedClaims returns the set of claim names authorized by given scopes

func (c *ClaimsMapper) GetAuthorizedClaims(scopes []string) []string {

```

```
// TODO: Build union of claim names from scope mappings

return nil

}

// AddCustomScope allows runtime registration of custom scope-to-claims mappings

func (c *ClaimsMapper) AddCustomScope(scope string, claims []string) {

// TODO: Add custom scope mapping while preventing conflicts with standard scopes

}
```

Consent Screen Handler (consent/screen.go):

```
package consent

import (
    "html/template"
    "net/http"
    "strings"

    "your-project/internal/clients"
)

// ConsentScreen handles display and processing of user consent decisions

type ConsentScreen struct {
    template     *template.Template
    clientStore  clients.ClientStore
    consentStore ConsentStore
}

func NewConsentScreen(clientStore clients.ClientStore, consentStore ConsentStore) *ConsentScreen {
    tmpl := template.Must(template.ParseFiles("web/templates/consent.html"))

    return &ConsentScreen{
        template:     tmpl,
        clientStore:  clientStore,
        consentStore: consentStore,
    }
}

// HandleConsentScreen displays consent form and processes user decisions

func (c *ConsentScreen) HandleConsentScreen(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case http.MethodGet:
        // TODO 1: Extract authorization request parameters from query string or session
        //           Get client_id, scope, redirect_uri, state from request
    }
}
```

```
//           Validate that all required parameters are present
//
//           Return 400 error if parameters missing or invalid

//
// TODO 2: Retrieve client information for display on consent screen
//
//           Look up client name and description using client_id
//
//           Return 400 error if client not found or inactive
//
//           Prepare client info for template rendering

//
// TODO 3: Parse requested scopes and prepare user-friendly descriptions
//
//           Split scope parameter into individual scope names
//
//           Map each scope to human-readable description
//
//           Group scopes by category (profile, email, etc.) for display

//
// TODO 4: Check for existing consent to show appropriate UI
//
//           Look up current user's consent record for this client
//
//           Identify which scopes are new vs. previously granted
//
//           Prepare consent status for template rendering

//
// TODO 5: Render consent screen template with prepared data
//
//           Pass client info, scope descriptions, and consent status
//
//           Include CSRF protection token in form
//
//           Handle template rendering errors appropriately

case http.MethodPost:

    //
// TODO 6: Process user's consent decision from form submission
//
//           Extract user decision (allow/deny) from form data
//
//           Validate CSRF token to prevent cross-site attacks
//
//           Get user ID from authenticated session
```

```

        // TODO 7: Create or update consent record based on user decision

        //           If user allowed, create ConsentRecord with granted scopes

        //           If user denied, redirect with access_denied error

        //           Handle partial scope grants if UI supports granular consent

        // TODO 8: Generate authorization code and redirect back to client

        //           Create authorization code bound to user, client, and scopes

        //           Build redirect URI with code and state parameters

        //           Use 302 redirect to send user back to client application

    }

    // Hint: Store authorization request parameters in secure session during GET

    // Hint: Validate redirect_uri matches registered client redirect URIs

    // Hint: Log all consent decisions for audit and analytics purposes

}

// scopeDescriptions maps scope names to user-friendly descriptions

var scopeDescriptions = map[string]string{

    "profile": "Access your basic profile information (name, picture)",

    "email": "Access your email address",

    "address": "Access your address information",

    "phone": "Access your phone number",

}

```

Language-Specific Hints

Go-Specific Implementation Tips:

- Use `strings.Fields()` to split space-separated scope strings from OAuth2 parameters
- Leverage struct tags (`json:"claim_name"`) to map between Go field names and OIDC claim names
- Use `sql.NullString` and `sql.NullTime` for optional database fields like `middle_name` or `expires_at`
- Implement `http.HandlerFunc` pattern for `Userinfo` endpoint to integrate with existing HTTP routing
- Use `context.WithTimeout()` for database operations to prevent hanging requests
- Cache JWT validation results using `sync.Map` or Redis with appropriate TTL values

- Use `html/template` for consent screens with automatic XSS protection through template escaping
- Implement proper error handling with structured logging using `log/slog` package

Database Integration Patterns:

- Use prepared statements for all database queries to prevent SQL injection
- Implement connection pooling with `sql.SetMaxOpenConns()` and `sql.SetMaxIdleConns()`
- Use database transactions for operations that modify multiple tables (consent + audit logs)
- Implement proper NULL handling for optional user profile fields
- Add database indexes on frequently queried fields: `user_id`, `client_id`, `(user_id, client_id)`
- Use `UPSERT` operations (PostgreSQL) or `ON DUPLICATE KEY UPDATE` (MySQL) for consent updates

Milestone Checkpoint

After implementing the UserInfo and consent management components, verify the following behavior:

UserInfo Endpoint Verification:

- Token Validation:** `curl -H "Authorization: Bearer <valid_token>" http://localhost:8080/userinfo` should return filtered user claims as JSON
- Invalid Token:** `curl -H "Authorization: Bearer invalid" http://localhost:8080/userinfo` should return 401 with `WWW-Authenticate` header
- Scope Filtering:** Token with only `openid profile` scopes should not return email claims even if user has email address
- Missing Claims:** User profile missing optional fields should omit those claims from response, not return null values

Expected UserInfo Response Format:

```
{
  "sub": "user_123456",
  "name": "John Smith",
  "given_name": "John",
  "family_name": "Smith",
  "preferred_username": "jsmith",
  "picture": "https://example.com/photos/jsmith.jpg",
  "locale": "en-US"
}
```

JSON

Consent Screen Verification:

- Display:** Navigate to `/consent?client_id=test&scope=openid+profile+email&redirect_uri=...` should show consent form
- Client Info:** Consent screen should display client name and readable scope descriptions
- Allow Flow:** Clicking "Allow" should redirect back to client with authorization code

4. **Deny Flow:** Clicking "Deny" should redirect back to client with `access_denied` error

Consent Persistence Verification:

1. **Skip Consent:** Second authorization request with same scopes should skip consent screen
2. **Additional Scopes:** Request with new scopes should show consent screen for additional permissions
3. **Revocation:** User revoking consent should force consent screen on next authorization

Signs of Problems:

- UserInfo returns claims not authorized by token scopes → Check scope-to-claims mapping logic
- Consent screen shows technical scope names instead of user-friendly descriptions → Update scope descriptions
- Repeated consent prompts for same client/scopes → Check consent record creation and lookup
- 500 errors on UserInfo requests → Check database connectivity and user profile schema
- CSRF attacks possible on consent form → Ensure CSRF token validation in POST handler

Interactions and Data Flow

Milestone(s): This section integrates concepts from all milestones (1-4), showing how the individual components work together in complete OAuth2 and OpenID Connect flows

Mental Model: Airport Security System

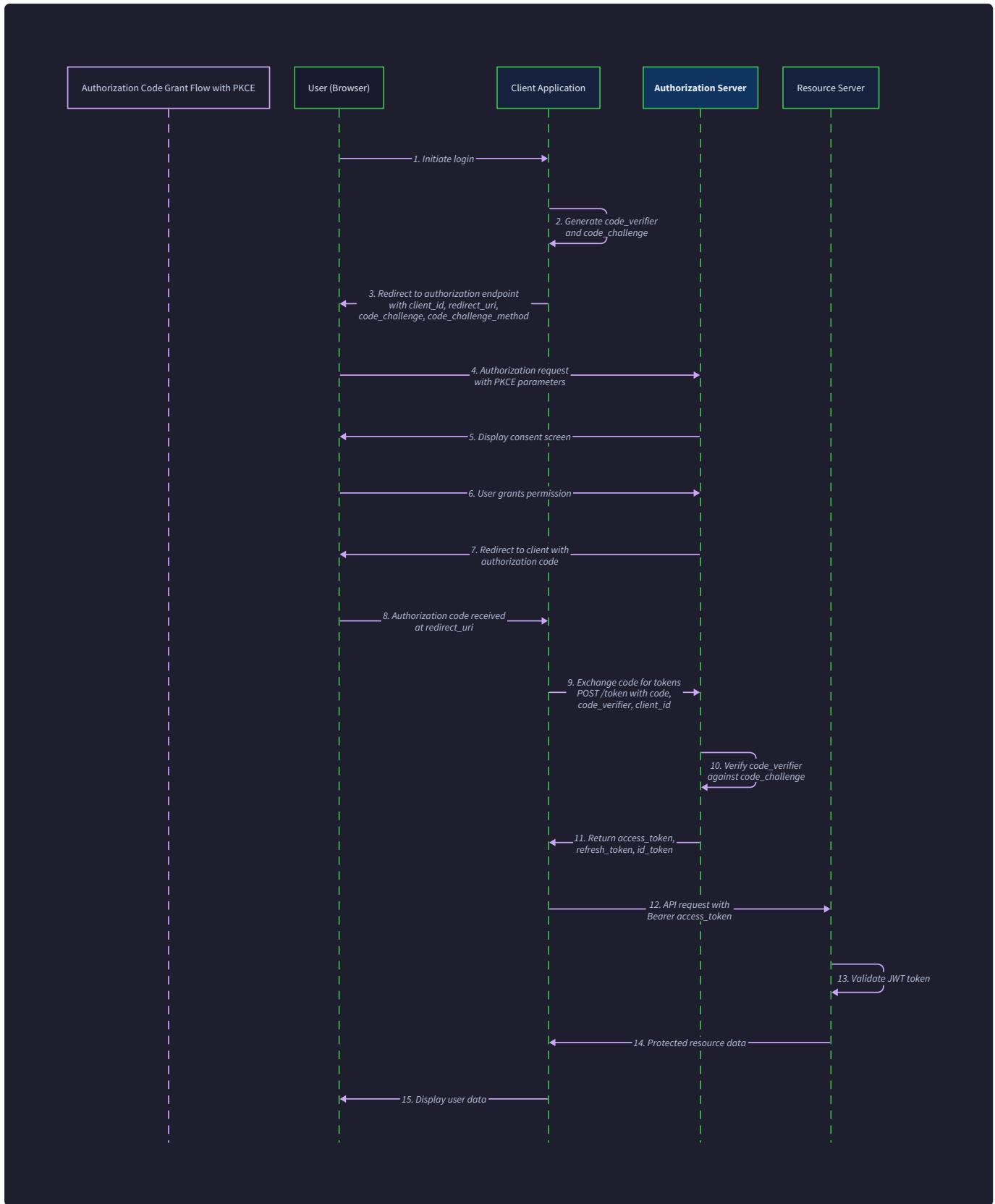
Think of the OAuth2 provider's interactions like a comprehensive airport security system. The authorization code grant flow is like checking in for a flight - you present your ID (authentication), get a boarding pass (authorization code), then exchange it at the gate for actual access to the plane (access token). The refresh token flow is like a frequent flyer program - you have a special card that lets you get new boarding passes without going through the full check-in process again. The client credentials flow is like airline staff accessing restricted areas - they have special badges that grant direct access without passenger processing. The OpenID Connect discovery endpoint is like the airport information desk - it tells everyone where to find all the different services and what procedures to follow.

This mental model helps us understand that OAuth2 isn't just about individual components, but about orchestrated workflows that move users and applications through a secure, standardized process. Each flow has different security requirements, different participants, and different success criteria, just like different types of airport access have different procedures and verification steps.

The key insight is that these flows are **state machines with cryptographic checkpoints**. Each step validates the previous step's output and produces cryptographically bound artifacts for the next step. This creates an audit trail and prevents attackers from skipping steps or replaying old interactions.

Authorization Code Grant Flow

The **authorization code grant flow** represents the gold standard for OAuth2 security, designed specifically for web applications that can securely store credentials. This flow implements the principle of **least privilege** by separating the authorization decision (made by the user) from the token issuance (handled by the authorization server), while ensuring that sensitive tokens never pass through potentially insecure channels like browser redirects.



The flow begins when the client application redirects the resource owner (user) to the authorization endpoint. This redirect carries several critical parameters that must be validated before proceeding. The client constructs an authorization URL

Phase 1: Client Redirect and Parameter Validation

The flow begins when the client application redirects the resource owner (user) to the authorization endpoint. This redirect carries several critical parameters that must be validated before proceeding. The client constructs an authorization URL

containing the client identifier, requested scopes, redirect URI, state parameter for CSRF protection, and PKCE code challenge for additional security.

Parameter	Required	Purpose	Validation Rules
<code>client_id</code>	Yes	Identifies the requesting application	Must match registered client
<code>redirect_uri</code>	Yes	Where to send the authorization response	Must exactly match registered URI
<code>scope</code>	Yes	Permissions being requested	Must be subset of client's allowed scopes
<code>state</code>	Yes	CSRF protection token	Must be cryptographically random
<code>code_challenge</code>	Yes (PKCE)	SHA256 hash of code verifier	Must be valid base64url encoding
<code>code_challenge_method</code>	Yes (PKCE)	Hash algorithm used	Must be "S256"
<code>response_type</code>	Yes	Type of response expected	Must be "code"

The authorization server receives this request and immediately begins a multi-step validation process. First, it validates the `client_id` by looking up the client registration in the `ClientStore`. If the client doesn't exist, the server responds with an `ErrorInvalidClient` directly to the user (not via redirect) since we cannot trust the `redirect_uri` without a valid client.

Once the client is validated, the server performs redirect URI validation using exact string matching. This is a critical security control - partial matches or pattern matching can lead to authorization code interception attacks. The `redirect_uri` in the request must exactly match one of the URIs registered for the client during the registration process.

Security Insight: Redirect URI validation is the primary defense against authorization code interception. An attacker who registers a malicious application cannot steal authorization codes intended for legitimate applications because they cannot predict or control the exact redirect URI that legitimate applications will use.

The server also validates the requested scopes to ensure they are a subset of the scopes the client is authorized to request. Scope validation prevents privilege escalation attacks where a client might attempt to request more permissions than originally granted during registration.

Phase 2: PKCE Code Challenge Processing

Proof Key for Code Exchange (PKCE) adds an additional layer of security specifically designed to protect against authorization code interception attacks, particularly relevant for mobile and single-page applications but recommended for all OAuth2 clients as a defense-in-depth measure.

The PKCE flow requires the client to generate a cryptographically random `code_verifier` (43-128 characters, base64url-encoded) and derive a `code_challenge` by applying SHA256 hashing and base64url encoding. The client sends only the `code_challenge` in the authorization request, keeping the `code_verifier` secret until the token exchange phase.

PKCE Component	Generation Method	Security Property
code_verifier	Cryptographically random 43-128 chars	High entropy, unpredictable
code_challenge	SHA256(code_verifier), base64url encoded	One-way function, tamper evident
code_challenge_method	Always "S256"	Specifies hash algorithm

The authorization server stores the code_challenge alongside the generated authorization code, creating a cryptographic binding between the authorization request and the eventual token request. This binding ensures that only the client that initiated the authorization flow can complete it, even if an attacker intercepts the authorization code during the redirect.

PKCE Security Flow:

1. Client generates random code_verifier: "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_ww1gFWFOEjXk"
2. Client computes code_challenge: SHA256(code_verifier) → base64url encode
3. Authorization server stores code_challenge with authorization code
4. Attacker intercepts authorization code but lacks code_verifier
5. Token request fails PKCE validation without correct code_verifier

Phase 3: User Authentication and Consent

After parameter validation, the authorization server must authenticate the resource owner and obtain their consent for the requested access. This phase typically involves redirecting the user to a login page (if not already authenticated) followed by a consent screen that clearly displays what the client application is requesting.

The consent screen serves multiple security and usability purposes. It provides **informed consent** by showing the user exactly what permissions they are granting, to which application, and for what purpose. It also serves as a **phishing detection** mechanism - legitimate consent screens will accurately display the client application's registered name and icon, while phishing attempts often contain obvious discrepancies.

Consent Screen Element	Purpose	Security Consideration
Client Name	Identifies the requesting application	Must match registered name exactly
Client Icon/Logo	Visual identification	Should be displayed consistently
Requested Scopes	Shows specific permissions	Must use clear, non-technical language
Scope Descriptions	Explains what each permission allows	Should be specific, not vague
Allow/Deny Buttons	Captures user decision	Must be equally prominent
User Information	Shows who is granting consent	Prevents confused deputy attacks

The consent mechanism also implements **consent persistence** to improve user experience. If a user has previously granted the same scopes to the same client, the authorization server can skip the consent screen and proceed directly to authorization code generation, unless the client explicitly requests re-consent or additional scopes.

Design Principle: Consent screens should optimize for **informed decision-making** rather than conversion rates.

Users need sufficient information to make security decisions, even if this occasionally results in denied requests.

Phase 4: Authorization Code Generation and Binding

Upon receiving user consent, the authorization server generates a cryptographically secure authorization code and binds it to the specific authorization request parameters. This binding is crucial for preventing various attack scenarios including authorization code replay, client substitution, and parameter modification attacks.

The `AuthorizationCode` structure captures all the essential binding information:

Field	Purpose	Security Binding
Code	The authorization code itself	Cryptographically random, 128+ bits entropy
ClientID	Which client can use this code	Prevents cross-client code usage
UserID	Which user granted consent	Prevents user substitution
RedirectURI	Where the response should go	Prevents redirect attacks
Scope	What permissions were granted	Prevents scope elevation
CodeChallenge	PKCE challenge for verification	Prevents code interception
ExpiresAt	When the code becomes invalid	Limits attack window
Used	Whether code has been exchanged	Prevents replay attacks

Authorization codes must be **single-use** and **short-lived** (maximum 10 minutes, recommended 5 minutes). The short lifetime limits the window of opportunity for interception attacks, while single-use prevents replay attacks even if an attacker obtains a valid code.

The authorization server generates the code using a cryptographically secure random number generator with at least 128 bits of entropy. The code should be base64url encoded to ensure safe transmission through URL parameters without requiring additional encoding.

Phase 5: Authorization Response and Redirect

After generating the authorization code, the authorization server constructs the authorization response and redirects the user back to the client application. This redirect carries the authorization code and the state parameter, allowing the client to continue the flow and verify that the response corresponds to its original request.

Response Parameter	Required	Purpose	Validation by Client
code	Yes	The authorization code	Store securely for token exchange
state	Yes	CSRF protection token	Must match original request

The redirect uses HTTP 302 status with the authorization code and state appended as query parameters to the registered `redirect_uri`. The client application receives this redirect and immediately validates the state parameter to ensure the response corresponds to its original authorization request, providing protection against CSRF attacks.

```

Successful Authorization Response:
HTTP/1.1 302 Found
Location: https://client.example.com/callback?code=Spxl0BeZQQYbYS6WxSbIA&state=af0ifjsldkj

Error Authorization Response:
HTTP/1.1 302 Found
Location: https://client.example.com/callback?
error=access_denied&error_description=User+denied+access&state=af0ifjsldkj

```

If the user denies consent or an error occurs during processing, the authorization server redirects back to the client with an error parameter instead of an authorization code. This allows the client application to handle the denial gracefully and provide appropriate user feedback.

Phase 6: Authorization Code Exchange for Tokens

The final phase occurs when the client exchanges the authorization code for access and refresh tokens at the token endpoint. This exchange happens server-to-server, providing confidentiality and allowing for strong client authentication.

The token request must include several parameters that are validated against the stored authorization code:

Token Request Parameter	Required	Purpose	Validation
<code>grant_type</code>	Yes	Must be "authorization_code"	Specifies the grant type
<code>code</code>	Yes	The authorization code	Must exist and be unused
<code>redirect_uri</code>	Yes	Must match original request	Prevents parameter modification
<code>code_verifier</code>	Yes (PKCE)	Proves client identity	Must hash to stored challenge
<code>client_id</code>	Yes	Client authentication	Must match code binding
<code>client_secret</code>	Yes (confidential)	Client authentication	Must be valid for client

The authorization server performs comprehensive validation of the token request:

- Client Authentication:** Verify the client credentials using the registered authentication method (`client_secret_basic` or `client_secret_post`)
- Authorization Code Validation:** Confirm the code exists, hasn't been used, and hasn't expired
- Parameter Binding Validation:** Ensure the `redirect_uri` matches the original authorization request
- PKCE Verification:** Hash the `code_verifier` and compare against the stored `code_challenge`
- Client Binding Validation:** Confirm the requesting client matches the client for whom the code was issued

Upon successful validation, the authorization server issues an access token and refresh token, marks the authorization code as used, and returns the token response to the client.

Architecture Decision: Authorization Code Single-Use Enforcement

- **Context:** Authorization codes could theoretically be reused multiple times until expiration
- **Options Considered:**
 - Allow multiple uses until expiration
 - Single-use with immediate invalidation
 - Limited number of uses (e.g., 3 attempts)
- **Decision:** Single-use with immediate invalidation
- **Rationale:** Single-use provides the strongest security posture by eliminating replay attack windows and simplifying the security model. The OAuth2 specification recommends single-use, and the marginal complexity of handling client retry scenarios is outweighed by the security benefits.
- **Consequences:** Clients must implement proper error handling for network failures during token exchange, but the overall system security is significantly improved with no additional complexity in the authorization server.

Refresh Token Flow

The **refresh token flow** provides a secure mechanism for clients to obtain new access tokens without requiring user interaction. This flow is essential for maintaining long-term access to protected resources while preserving the security principle of short-lived access tokens.

The refresh token flow implements **token rotation** as a key security feature. Each time a refresh token is used, the authorization server issues a new refresh token and invalidates the old one. This rotation provides forward security and enables detection of token theft through the identification of refresh token reuse attempts.

Mental Model: Subscription Renewal System

Think of refresh tokens like magazine subscriptions with automatic renewal cards. You get an initial subscription (access token) that expires after a short period, but you also receive a renewal card (refresh token) that lets you extend your subscription. Each time you use the renewal card, you get a new subscription period and a new renewal card, while the old renewal card becomes invalid. If someone steals your old renewal card and tries to use it, the magazine company knows something suspicious is happening because that card should no longer exist.

Token Family Tracking and Rotation Security

Refresh tokens are organized into **token families** that track the lineage of refresh token rotations. Each token family has a unique identifier that remains constant across all refresh token rotations within that authorization grant. This family tracking enables sophisticated security features like breach detection and family-wide revocation.

Token Family Concept	Purpose	Security Benefit
TokenFamily ID	Groups related refresh tokens	Enables family-wide operations
ParentToken reference	Tracks rotation lineage	Detects unauthorized reuse
Family revocation	Invalidates entire token family	Limits breach impact
Rotation audit log	Records all token usage	Enables forensic analysis

When a refresh token is used legitimately, the following sequence occurs:

1. Client presents refresh token to token endpoint
2. Authorization server validates the token and checks revocation status
3. Server generates new access token with fresh expiration
4. Server generates new refresh token in the same family
5. Server marks the old refresh token as used/revoked
6. Server updates the parent-child relationship in the token family
7. Server returns new tokens to the client

The critical security feature is **breach detection through reuse identification**. If an attacker obtains a refresh token that has already been rotated, their attempt to use the old token triggers a security alert. The authorization server can then revoke the entire token family, preventing the attacker from using any tokens while forcing the legitimate client to re-authenticate.

Normal Rotation Sequence:

```
RT1 (active) → exchange → RT2 (active), RT1 (revoked)
RT2 (active) → exchange → RT3 (active), RT2 (revoked)
```

Breach Detection Scenario:

```
RT2 (revoked) → reuse attempt → SECURITY ALERT: revoke entire family
Legitimate client with RT3 → next exchange fails → re-authentication required
```

Refresh Token Request Processing

The refresh token exchange follows a structured validation process that ensures both security and correctness. The client makes a POST request to the token endpoint with the refresh token grant type.

Refresh Request Parameter	Required	Purpose	Validation
grant_type	Yes	Must be "refresh_token"	Specifies grant type
refresh_token	Yes	The current refresh token	Must be valid and not revoked
scope	No	Requested scope (subset)	Must not exceed original grant
client_id	Yes	Client identification	Must match token binding
client_secret	Yes (confidential)	Client authentication	Must be valid for client

The authorization server processes the refresh token request through several validation phases:

Phase 1: Token Format and Existence Validation The server first validates that the refresh token exists in the `RefreshTokenStore` and retrieves its associated metadata. This includes checking the token's format, ensuring it hasn't been marked as revoked, and confirming it hasn't exceeded its maximum lifetime.

Phase 2: Client Authentication and Binding Validation The server authenticates the client using the provided credentials and verifies that the refresh token was originally issued to this client. This prevents clients from using refresh tokens issued to other clients, even if they somehow obtain them.

Phase 3: Token Family and Lineage Validation The server checks the token's position in its token family and verifies that it represents the current active token in the family lineage. This is where breach detection occurs - if the token has already been rotated (has a child token), the server knows this is likely a replay attack.

Phase 4: Scope Validation and Filtering If the client requests specific scopes in the refresh request, the server validates that these scopes are a subset of the originally granted scopes. This prevents scope elevation attacks where a client might attempt to gain additional permissions during token refresh.

Token Rotation Implementation

Upon successful validation, the authorization server performs the token rotation process that maintains security while providing the client with fresh credentials.

The rotation process follows these steps:

1. **Generate New Access Token:** Create a fresh JWT access token with updated expiration time and the same user/client/scope binding as the original grant
2. **Generate New Refresh Token:** Create a new refresh token in the same token family, setting the current token as its parent
3. **Update Token Family State:** Mark the old refresh token as used and establish the parent-child relationship
4. **Persist State Changes:** Atomically commit all state changes to prevent inconsistencies
5. **Return Token Response:** Provide the new tokens to the client in standard OAuth2 format

The atomic persistence of state changes is crucial for preventing race conditions and ensuring data consistency. If multiple refresh requests arrive simultaneously (due to network retries or client bugs), only one should succeed while others should fail gracefully.

Token Response Field	Purpose	Value Source
access_token	New JWT access token	Freshly generated with updated expiration
token_type	Always "Bearer"	OAuth2 standard
expires_in	Access token lifetime in seconds	From JWT expiration claim
refresh_token	New refresh token	Generated in same family as replaced token
scope	Granted scopes	From original authorization or filtered request

Refresh Token Lifetime and Management

Refresh tokens have significantly longer lifetimes than access tokens, typically ranging from days to months depending on the security requirements and user experience goals. However, they are not unlimited in duration and must eventually expire to limit the impact of undetected breaches.

Refresh Token Attribute	Typical Value	Security Rationale
Maximum lifetime	30-90 days	Balances convenience with security exposure
Rotation frequency	Every access token refresh	Provides forward security
Family size limit	10-20 tokens	Prevents unbounded growth
Revocation propagation	Immediate	Ensures consistent security state

The authorization server implements **refresh token cleanup** processes that remove expired tokens and their associated metadata. This cleanup is essential for preventing database growth and ensuring that old, potentially compromised tokens

cannot be misused even if storage security is breached.

Design Principle: Refresh token rotation should be transparent to well-behaved clients while making token theft immediately detectable and automatically mitigated.

Client Credentials Flow

The **client credentials flow** provides a streamlined authentication mechanism for machine-to-machine communication where no human user is involved. This flow is essential for service-to-service authentication, background jobs, and API integration scenarios where applications need to access resources based on their own identity rather than on behalf of a user.

Mental Model: Corporate ID Badge System

Think of the client credentials flow like a corporate ID badge system for employees accessing different buildings or systems. An employee (client application) presents their company-issued badge (client credentials) to a security reader (authorization server), which immediately grants or denies access based on the badge's validity and the employee's clearance level. There's no additional approval process or human intervention - the badge itself contains all the necessary authorization information.

This analogy helps illustrate why the client credentials flow is both simpler and more restrictive than user-centric flows. The badge (client credentials) represents the application's inherent permissions rather than delegated user permissions, and access is immediate rather than requiring consent or multi-step validation.

Client Authentication Methods

The client credentials flow supports multiple authentication methods, allowing organizations to choose the approach that best fits their security requirements and infrastructure capabilities.

Authentication Method	Security Level	Use Case	Implementation
client_secret_basic	Medium	Traditional web services	HTTP Basic authentication with client_id:client_secret
client_secret_post	Medium	APIs that can't use Basic auth	client_id and client_secret in request body
client_secret_jwt	High	High-security environments	JWT signed with shared secret
private_key_jwt	Very High	Zero-shared-secret requirements	JWT signed with client's private key

HTTP Basic Authentication represents the most common implementation, where the client encodes its client_id and client_secret as a Base64-encoded string in the Authorization header. This method provides reasonable security when used over HTTPS but requires secure storage of the client secret.

POST Body Authentication offers an alternative for clients that cannot easily set HTTP headers or prefer to send credentials in the request body. While functionally equivalent to Basic authentication, this method may be preferred in certain API gateway or proxy configurations.

JWT Authentication Methods provide enhanced security by eliminating the need to transmit secrets directly. In `client_secret_jwt`, the client creates a JWT assertion signed with its registered secret, while `private_key_jwt` uses asymmetric cryptography to eliminate shared secrets entirely.

Client Credentials Request Processing

The client credentials flow follows a simplified request-response pattern that bypasses user authentication and consent while maintaining rigorous client authentication and authorization validation.

Request Parameter	Required	Purpose	Validation
<code>grant_type</code>	Yes	Must be "client_credentials"	Specifies the grant type
<code>scope</code>	Optional	Requested permissions	Must be subset of client's allowed scopes
<code>client_id</code>	Yes	Client identification	Must match authenticated client
<code>client_secret</code>	Conditional	Client authentication	Required for secret-based auth methods

The authorization server processes client credentials requests through a focused validation pipeline:

Client Authentication: The server authenticates the client using the provided credentials and the client's registered authentication method. This authentication must be successful before any authorization decisions are made.

Grant Type Authorization: The server verifies that the client is authorized to use the client credentials grant type. Not all registered clients should be permitted to use this flow - it should be restricted to applications with legitimate machine-to-machine access requirements.

Scope Validation: If the client requests specific scopes, the server validates that these scopes are within the client's pre-authorized scope set. Unlike user-centric flows, there is no dynamic consent process - all permissions must be pre-configured during client registration.

Token Generation: Upon successful validation, the server generates an access token that represents the client's identity and authorized scopes. No refresh token is typically issued since clients can re-authenticate using their credentials whenever needed.

Client-Scope Access Tokens

Access tokens issued through the client credentials flow have different characteristics than user-delegated tokens. These tokens represent the client application's inherent permissions rather than user-granted permissions, and their claims structure reflects this distinction.

JWT Claim	Purpose	Client Credentials Value
<code>sub</code>	Subject identifier	Client ID or client-specific identifier
<code>aud</code>	Intended audience	Resource server(s) the token is valid for
<code>iss</code>	Token issuer	Authorization server identifier
<code>exp</code>	Expiration time	Typically shorter than user tokens (1-24 hours)
<code>iat</code>	Issued at time	Current timestamp
<code>scope</code>	Granted permissions	Client's authorized scopes
<code>client_id</code>	Client identifier	Explicitly identifies the client
<code>token_type</code>	Token classification	"client_credentials" or similar

The absence of user-related claims (like user ID or user profile information) clearly indicates that these tokens represent application-level access rather than user-delegated access. Resource servers can use this distinction to apply different authorization policies for machine-to-machine access versus user-initiated access.

Security Considerations for Machine-to-Machine Access

Client credentials flows require careful security consideration because they bypass many of the user-centric protections built into other OAuth2 flows. Applications using this flow have persistent, programmatic access to resources without human oversight, making security breaches potentially more severe and harder to detect.

Credential Rotation: Client credentials should be rotated regularly to limit the impact of credential compromise. Unlike user passwords, client credentials are typically stored in configuration files or environment variables, making them vulnerable to different types of attacks.

Scope Minimization: Clients should be granted only the minimum scopes necessary for their legitimate functions. Since there is no user consent process to limit scope abuse, pre-authorization during client registration becomes the primary access control mechanism.

Audit Logging: All client credentials usage should be logged for security monitoring and compliance purposes. These logs should capture which client accessed what resources when, enabling detection of unusual patterns or potential abuse.

Network Security: Client credentials flows assume secure transport (HTTPS) and may benefit from additional network-level protections like API gateways, IP allowlisting, or mutual TLS authentication.

Architecture Decision: Refresh Token Omission in Client Credentials Flow

- **Context:** Client credentials flow could issue refresh tokens for consistency with other flows
- **Options Considered:**
 - Issue refresh tokens with rotation
 - Omit refresh tokens, require re-authentication
 - Make refresh token issuance configurable per client
- **Decision:** Omit refresh tokens by default
- **Rationale:** Clients can re-authenticate using their credentials anytime, eliminating the complexity of refresh token management without significant usability impact. This also reduces the attack surface by eliminating long-lived tokens that could be compromised.
- **Consequences:** Clients must handle access token expiration by re-authenticating, but this is typically automated and transparent to users. Security is improved by eliminating persistent tokens.

OpenID Connect Discovery

OpenID Connect Discovery provides a standardized mechanism for clients to automatically discover the endpoints, capabilities, and configuration of an OpenID Connect provider. This discovery mechanism eliminates the need for manual configuration and enables dynamic client registration and endpoint resolution.

Mental Model: Airport Information System

Think of OpenID Connect Discovery like the information systems at a major airport. When you arrive at an unfamiliar airport, you don't need to memorize the locations of every gate, restaurant, and service - instead, you consult the information displays and maps that tell you where everything is located. Similarly, OAuth2 clients don't need to have the authorization server's endpoints hard-coded - they can query the well-known discovery endpoint to learn about all available services, their locations, and their capabilities.

This discovery system makes the OAuth2 ecosystem more dynamic and maintainable. Just as airports can relocate services or add new amenities without requiring travelers to update their personal navigation systems, authorization servers can change endpoints or add new features without breaking existing client integrations.

Well-Known Configuration Endpoint

The discovery process centers around the **well-known configuration endpoint**, which must be available at a standardized location relative to the authorization server's issuer identifier. This endpoint returns a JSON document containing comprehensive information about the provider's capabilities and endpoint locations.

The well-known endpoint follows the pattern: `{issuer}/.well-known/openid_configuration`

For example:

- Issuer: `https://auth.example.com`
- Discovery endpoint: `https://auth.example.com/.well-known/openid_configuration`

This standardized location allows clients to discover provider capabilities using only the issuer identifier, enabling truly dynamic configuration and reducing deployment complexity.

Discovery Document Structure

The discovery document contains extensive metadata about the OpenID Connect provider, organized into several categories of information that clients use for different purposes.

Configuration Category	Purpose	Client Usage
Endpoint URLs	Locations of OAuth2/OIDC endpoints	Endpoint resolution for API calls
Supported Features	Capabilities and optional features	Feature detection and client adaptation
Cryptographic Information	Key locations and algorithms	Token validation and security setup
Metadata	Provider identification and contact	Error reporting and support

Core Endpoint Configuration:

Endpoint Field	Required	Purpose	Example Value
<code>issuer</code>	Yes	Provider identifier	<code>https://auth.example.com</code>
<code>authorization_endpoint</code>	Yes	OAuth2 authorization URL	<code>https://auth.example.com/oauth2/authorize</code>
<code>token_endpoint</code>	Yes	OAuth2 token exchange URL	<code>https://auth.example.com/oauth2/token</code>
<code>userinfo_endpoint</code>	Yes (OIDC)	User profile claims URL	<code>https://auth.example.com/oidc/userinfo</code>
<code>jwks_uri</code>	Yes	JSON Web Key Set location	<code>https://auth.example.com/.well-known/jwks.json</code>
<code>introspection_endpoint</code>	No	RFC 7662 token validation	<code>https://auth.example.com/oauth2/introspect</code>
<code>revocation_endpoint</code>	No	RFC 7009 token revocation	<code>https://auth.example.com/oauth2/revoke</code>

Capability Declaration:

Capability Field	Purpose	Example Values
<code>response_types_supported</code>	Supported OAuth2 response types	<code>["code", "code id_token"]</code>
<code>grant_types_supported</code>	Supported grant types	<code>["authorization_code", "refresh_token", "client_credentials"]</code>
<code>scopes_supported</code>	Available scopes	<code>["openid", "profile", "email", "offline_access"]</code>
<code>response_modes_supported</code>	Supported response modes	<code>["query", "fragment", "form_post"]</code>
<code>code_challenge_methods_supported</code>	PKCE methods	<code>["S256"]</code>

Token and Cryptographic Configuration:

Security Field	Purpose	Example Values
<code>id_token_signing_alg_values_supported</code>	ID token signature algorithms	<code>["RS256", "ES256"]</code>
<code>token_endpoint_auth_methods_supported</code>	Client authentication methods	<code>["client_secret_basic", "client_secret_post", "private_key_jwt"]</code>
<code>claims_supported</code>	Available user claims	<code>["sub", "name", "email", "picture", "email_verified"]</code>
<code>subject_types_supported</code>	Subject identifier types	<code>["public", "pairwise"]</code>

Dynamic Client Configuration

Discovery enables **dynamic client configuration**, where client applications can adapt their behavior based on the provider's capabilities rather than requiring manual configuration updates. This capability is particularly valuable in multi-tenant environments or when integrating with multiple identity providers.

Client applications can use discovery information to:

Endpoint Resolution: Dynamically determine the correct URLs for authorization, token exchange, and user information requests without hard-coding endpoint locations.

Feature Detection: Check whether the provider supports specific features like PKCE, token introspection, or particular authentication methods before attempting to use them.

Algorithm Selection: Choose appropriate cryptographic algorithms for token validation based on the provider's supported algorithm list.

Scope Discovery: Present users with accurate information about available permissions and claims based on the provider's capability declarations.

Dynamic Configuration Example:

1. Client discovers provider supports ["S256"] for PKCE
2. Client generates PKCE challenge using SHA256
3. Client discovers authorization endpoint URL
4. Client constructs authorization request with discovered endpoint
5. Client discovers supported scopes and presents appropriate UI

JWKS Endpoint Integration

The **JSON Web Key Set (JWKS)** endpoint provides cryptographic public keys that clients use to validate JWT tokens issued by the authorization server. The discovery document includes the `jwks_uri` field pointing to this endpoint, enabling automated key retrieval and rotation.

JWKS Field	Purpose	Example Value
<code>keys</code>	Array of public keys	JSON Web Key objects
<code>kty</code>	Key type	"RSA", "EC"
<code>use</code>	Key usage	"sig" (signature)
<code>kid</code>	Key identifier	"2023-10-key-1"
<code>alg</code>	Algorithm	"RS256", "ES256"
<code>n, e</code>	RSA public key components	Base64url-encoded values

Clients cache JWKS information but must handle key rotation gracefully. When token validation fails due to an unknown key ID, clients should refresh their JWKS cache and retry validation before concluding that the token is invalid.

Architecture Decision: Static vs Dynamic JWKS

- **Context:** JWKS endpoint could serve static keys or dynamically rotate keys
- **Options Considered:**
 - Static keys with manual rotation procedures
 - Dynamic rotation with cache invalidation
 - Hybrid approach with predictable rotation schedule
- **Decision:** Dynamic rotation with cache-friendly headers
- **Rationale:** Dynamic key rotation provides better security through regular key renewal while cache headers (`Cache-Control: max-age=3600`) balance security with performance. Clients can cache keys safely while ensuring they detect rotations within a reasonable timeframe.
- **Consequences:** Clients must implement JWKS caching and refresh logic, but overall system security is improved through automated key rotation.

Caching and Performance Considerations

Discovery documents are relatively stable but may change when providers add new features, endpoints, or capabilities. Proper caching strategies balance the need for current information with performance and availability requirements.

Cache Strategy	Cache Duration	Update Trigger	Use Case
Time-based	1-24 hours	Cache expiration	General purpose caching
ETag-based	Until changed	HTTP ETag mismatch	Bandwidth optimization
Failure-tolerant	Indefinite with refresh attempts	Provider availability	High availability scenarios
Event-driven	Until notified	Provider notification	Real-time updates

Clients should implement **graceful degradation** when discovery endpoints are unavailable. This might involve falling back to cached discovery documents or pre-configured endpoint information while continuing to attempt discovery refresh in the background.

The discovery endpoint should return appropriate HTTP caching headers to guide client caching behavior:

```
HTTP Response Headers for Discovery:
Cache-Control: public, max-age=3600
ETag: "discovery-v1.2.3"
Content-Type: application/json
```

These headers indicate that the discovery document can be cached publicly for one hour and provide an ETag for efficient cache validation on subsequent requests.

Common Pitfalls

⚠ Pitfall: Authorization Code Replay Attacks Failing to mark authorization codes as single-use allows attackers who intercept codes to replay them for token access. Always mark codes as used immediately upon successful exchange and reject any subsequent attempts to use the same code.

⚠ Pitfall: PKCE Challenge Reuse Storing PKCE challenges without proper cleanup can lead to challenges being reused across multiple authorization attempts. Always bind PKCE challenges to specific authorization codes and clean them up when codes expire or are used.

⚠ Pitfall: Refresh Token Family Cleanup Failures Not properly cleaning up revoked refresh token families can lead to database bloat and potential security issues. Implement automated cleanup processes that remove expired and revoked tokens while maintaining audit trails.

⚠ Pitfall: Client Credentials Scope Elevation Allowing clients to request arbitrary scopes in client credentials flow bypasses the pre-authorization security model. Always validate that requested scopes are a subset of the client's registered allowed scopes.

⚠ Pitfall: Discovery Document Staleness Serving outdated discovery documents can break client integrations when endpoints change. Implement proper cache invalidation and ensure discovery documents accurately reflect current provider capabilities.

⚠ Pitfall: Race Conditions in Token Rotation Simultaneous refresh token requests can lead to multiple valid tokens or corrupted token family state. Use atomic database operations and proper locking to ensure only one refresh operation succeeds per token.

⚠ Pitfall: Insufficient Error Context Returning generic error messages makes debugging integration issues difficult for client developers. Provide specific error descriptions that help identify the exact validation failure while avoiding

information disclosure.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	<code>net/http</code> with <code>gorilla/mux</code>	<code>gin-gonic/gin</code> with middleware
Database Access	Direct <code>database/sql</code> with PostgreSQL	GORM ORM with connection pooling
JWT Handling	<code>golang-jwt/jwt/v5</code>	Custom implementation with key rotation
Cryptographic Operations	<code>crypto/rand</code> and <code>crypto/sha256</code>	<code>golang.org/x/crypto</code> for advanced features
Configuration Management	Environment variables with <code>viper</code>	Consul or etcd for dynamic configuration
Caching Layer	In-memory map with mutex	Redis with clustering
Logging and Monitoring	<code>log/slog</code> structured logging	OpenTelemetry with distributed tracing

File Structure for Flow Orchestration

```
internal/
  flows/
    authorization_code.go      ← Authorization code grant orchestration
    refresh_token.go           ← Refresh token flow implementation
    client_credentials.go     ← Client credentials flow
    flow_common.go             ← Shared flow utilities
  discovery/
    discovery.go               ← OpenID Connect discovery endpoint
    jwks.go                    ← JSON Web Key Set management
  handlers/
    oauth2_handlers.go        ← HTTP handlers for OAuth2 endpoints
    oidc_handlers.go          ← HTTP handlers for OIDC endpoints
  middleware/
    cors.go                   ← CORS headers for browser clients
    rate_limiting.go          ← Rate limiting and abuse prevention
    request_logging.go        ← Request/response logging
```

Complete Authorization Code Flow Implementation

```
package flows

import (
    "context"
    "fmt"
    "net/http"
    "time"

    "internal/models"
    "internal/services"
)

// AuthorizationCodeFlow orchestrates the complete authorization code grant flow

type AuthorizationCodeFlow struct {

    clientStore      services.ClientStore

    authzService     *services.AuthorizationService

    tokenGenerator   *services.JWTGenerator

    pkceValidator    *services.PKCEValidator

    codeStore        services.AuthorizationCodeStore

    refreshStore     services.RefreshTokenStore

    userStore        services.UserStore

    consentStore     services.ConsentStore

}

// HandleAuthorizationRequest processes the initial authorization request

func (f *AuthorizationCodeFlow) HandleAuthorizationRequest(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Parse and validate authorization request parameters

    // - Extract client_id, redirect_uri, scope, state, code_challenge, code_challenge_method

    // - Validate required parameters are present

    // - Validate parameter formats and lengths
}
```

```

// TODO 2: Validate client registration and redirect URI

// - Look up client in ClientStore using client_id

// - Return error if client not found or inactive

// - Validate redirect_uri exactly matches registered URI


// TODO 3: Validate and store PKCE code challenge

// - Verify code_challenge_method is "S256"

// - Validate code_challenge is proper base64url encoding

// - Store challenge for later verification


// TODO 4: Check for existing user session

// - Look for authentication session cookie

// - Redirect to login page if user not authenticated

// - Extract authenticated user ID for consent checking


// TODO 5: Check existing consent or show consent screen

// - Query ConsentStore for previous user consent to this client

// - If consent exists and covers requested scopes, proceed to code generation

// - Otherwise, redirect to consent screen with request parameters


// TODO 6: Generate and store authorization code

// - Generate cryptographically secure authorization code

// - Create AuthorizationCode record with all binding parameters

// - Store in AuthorizationCodeStore with expiration

// - Redirect to client with code and state parameter

}

// HandleTokenRequest processes the token exchange request

func (f *AuthorizationCodeFlow) HandleTokenRequest(w http.ResponseWriter, r *http.Request) {

```

```
// TODO 1: Parse token request parameters

// - Extract grant_type, code, redirect_uri, code_verifier, client credentials

// - Validate grant_type is "authorization_code"

// - Validate all required parameters are present


// TODO 2: Authenticate the client

// - Extract client credentials from Authorization header or request body

// - Look up client in ClientStore and validate credentials

// - Return invalid_client error if authentication fails


// TODO 3: Validate authorization code

// - Retrieve AuthorizationCode from store using code parameter

// - Check code exists, hasn't expired, and hasn't been used

// - Verify code was issued to the authenticating client

// - Mark code as used to prevent replay attacks


// TODO 4: Validate PKCE code verifier

// - Hash the code_verifier parameter using SHA256

// - Compare hash against stored code_challenge

// - Return invalid_grant error if PKCE validation fails


// TODO 5: Validate redirect_uri consistency

// - Compare redirect_uri parameter against stored value in authorization code

// - Return invalid_grant error if URIs don't match exactly


// TODO 6: Generate access and refresh tokens

// - Create JWT access token with appropriate claims and expiration

// - Generate refresh token with token family tracking

// - Store refresh token in RefreshTokenStore
```

```
// - Return TokenResponse with both tokens
}

// ValidateAuthorizationRequest performs comprehensive parameter validation

func (f *AuthorizationCodeFlow) ValidateAuthorizationRequest(req *models.AuthorizationRequest, client *models.Client) error {

    // TODO 1: Validate response_type parameter

    // - Must be exactly "code" for authorization code flow

    // - Return unsupported_response_type error for other values

    // TODO 2: Validate redirect_uri parameter

    // - Must exactly match one of client's registered redirect URIs

    // - Use string comparison, no pattern matching or partial matches

    // - Return invalid_request error if no match found

    // TODO 3: Validate requested scopes

    // - Parse space-delimited scope parameter

    // - Verify each scope is in client's allowed scopes list

    // - Return invalid_scope error if any scope is not allowed

    // TODO 4: Validate state parameter

    // - Must be present for CSRF protection

    // - Should have sufficient entropy (recommend 128+ bits)

    // - Return invalid_request error if missing

    // TODO 5: Validate PKCE parameters

    // - code_challenge must be present (PKCE required)

    // - code_challenge_method must be "S256"

    // - code_challenge must be valid base64url encoding

    // - Return invalid_request error for any PKCE validation failure
```

}

Refresh Token Flow Implementation Skeleton

```
// RefreshTokenFlow handles refresh token rotation and access token renewal          GO

type RefreshTokenFlow struct {

    refreshStore    services.RefreshTokenStore

    tokenGenerator *services.JWTGenerator

    clientStore    services.ClientStore

}

// HandleRefreshTokenRequest processes refresh token exchange requests

func (f *RefreshTokenFlow) HandleRefreshTokenRequest(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Parse and validate refresh token request

    // - Extract grant_type, refresh_token, scope, client credentials

    // - Validate grant_type is "refresh_token"

    // - Authenticate client using provided credentials


    // TODO 2: Retrieve and validate refresh token

    // - Look up refresh token in RefreshTokenStore

    // - Check token exists, is active, and hasn't expired

    // - Verify token was issued to the authenticated client


    // TODO 3: Detect token reuse attacks

    // - Check if refresh token has already been rotated (has child token)

    // - If reuse detected, revoke entire token family

    // - Log security event for monitoring and alerting


    // TODO 4: Validate scope parameter if present

    // - Parse requested scopes from scope parameter

    // - Ensure requested scopes are subset of original grant

    // - Default to original scopes if no scope parameter provided
```

```
// TODO 5: Perform token rotation

// - Generate new access token with fresh expiration

// - Generate new refresh token in same token family

// - Mark old refresh token as used/revoked

// - Update token family lineage in database


// TODO 6: Return new tokens to client

// - Create TokenResponse with new access and refresh tokens

// - Include expires_in and scope in response

// - Use atomic database transaction to ensure consistency

}

// DetectTokenReuse identifies potential refresh token theft

func (f *RefreshTokenFlow) DetectTokenReuse(ctx context.Context, token *models.RefreshToken) (bool, error) {

    // TODO 1: Check if token has been rotated

    // - Query for child tokens in the same family

    // - If child tokens exist, this is likely a reuse attack


    // TODO 2: Check token family health

    // - Look for multiple active tokens in same family (should be impossible)

    // - Check for suspicious usage patterns in family history


    // TODO 3: Implement breach detection logic

    // - If reuse detected, mark entire family for revocation

    // - Log security incident with details for investigation

    // - Return true to trigger family revocation process

}
```

Client Credentials Flow Implementation Skeleton

```
// ClientCredentialsFlow handles machine-to-machine authentication          GO

type ClientCredentialsFlow struct {

    clientStore      services.ClientStore

    tokenGenerator *services.JWTGenerator

}

// HandleClientCredentialsRequest processes client credentials grant requests

func (f *ClientCredentialsFlow) HandleClientCredentialsRequest(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Parse client credentials request

    // - Extract grant_type, scope, and client authentication parameters

    // - Validate grant_type is "client_credentials"

    // - Parse scope parameter into individual scopes


    // TODO 2: Authenticate the client

    // - Extract client credentials using configured authentication method

    // - Support client_secret_basic, client_secret_post, and JWT methods

    // - Return invalid_client error if authentication fails


    // TODO 3: Authorize client for credentials grant

    // - Verify client is registered for client_credentials grant type

    // - Check that client is active and not suspended

    // - Return unauthorized_client if not authorized for this grant


    // TODO 4: Validate requested scopes

    // - Compare requested scopes against client's pre-authorized scopes

    // - No dynamic consent process - all scopes must be pre-approved

    // - Return invalid_scope error for unauthorized scopes


    // TODO 5: Generate access token
```

```
// - Create JWT access token representing client identity  
  
// - Include client_id as subject, no user-related claims  
  
// - Set appropriate expiration (typically shorter than user tokens)  
  
// - No refresh token issued (client can re-authenticate)  
  
  
// TODO 6: Return token response  
  
// - Create TokenResponse with access token only  
  
// - Include token_type as "Bearer" and expires_in value  
  
// - Include granted scopes in response  
  
}
```

OpenID Connect Discovery Endpoint

```
// DiscoveryService provides OpenID Connect discovery capabilities          GO

type DiscoveryService struct {

    config      *models.Config

    keyManager *services.KeyManager

}

// HandleDiscoveryRequest serves the well-known configuration document

func (s *DiscoveryService) HandleDiscoveryRequest(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Build discovery document structure

    // - Create DiscoveryDocument with all required OIDC fields

    // - Include issuer, authorization_endpoint, token_endpoint, userinfo_endpoint

    // - Add jwks_uri, introspection_endpoint, revocation_endpoint


    // TODO 2: Populate supported capabilities

    // - List supported response_types, grant_types, scopes

    // - Include supported signing algorithms and authentication methods

    // - Add supported claims and subject types


    // TODO 3: Set appropriate caching headers

    // - Use Cache-Control with reasonable max-age (e.g., 3600 seconds)

    // - Include ETag for efficient cache validation

    // - Set Content-Type to application/json


    // TODO 4: Serialize and return discovery document

    // - Marshal DiscoveryDocument to JSON

    // - Handle serialization errors gracefully

    // - Return 200 OK with JSON response body

}

// BuildDiscoveryDocument creates the OpenID Connect discovery response
```

```
func (s *DiscoveryService) BuildDiscoveryDocument() (*models.DiscoveryDocument, error) {  
  
    baseURL := s.config.Server.BaseURL  
  
    return &models.DiscoveryDocument{  
  
        Issuer:           baseURL,  
  
        AuthorizationEndpoint: baseURL + "/oauth2/authorize",  
  
        TokenEndpoint:      baseURL + "/oauth2/token",  
  
        UserInfoEndpoint:   baseURL + "/oidc/userinfo",  
  
        JwksURI:          baseURL + "/.well-known/jwks.json",  
  
        IntrospectionEndpoint: baseURL + "/oauth2/introspect",  
  
        RevocationEndpoint:  baseURL + "/oauth2/revoke",  
  
        // TODO: Add all other required and optional discovery fields  
  
        // Include supported response types, grant types, scopes, algorithms  
  
        // Reference the OIDC discovery specification for complete field list  
  
    }, nil  
}
```

Flow Orchestration Testing

Milestone Checkpoint for Authorization Code Flow:

```

# Test authorization request handling                                BASH

curl -v "http://localhost:8080/oauth2/authorize?
client_id=test_client&redirect_uri=https://client.example.com/callback&scope=openid%20profile&state=r
andom_state&code_challenge=challenge&code_challenge_method=S256"

# Expected: Redirect to consent screen or authorization code response

# Check: Proper parameter validation and PKCE challenge storage

# Test token exchange

curl -X POST http://localhost:8080/oauth2/token \
-H "Authorization: Basic dGVzdF9jbGllbnQ6c2VjcmV0" \
-d
"grant_type=authorization_code&code=AUTHZ_CODE&redirect_uri=https://client.example.com/callback&code_
verifier=verifier"

# Expected: JSON response with access_token and refresh_token

# Check: PKCE verification and token generation

```

Debugging Common Issues:

Symptom	Likely Cause	Diagnosis	Fix
Authorization hangs on consent	Session management issue	Check user authentication flow	Implement proper session handling
PKCE validation always fails	Hash encoding mismatch	Log code_challenge vs computed hash	Use consistent base64url encoding
Tokens appear malformed	JWT signing key issues	Verify key format and algorithm	Check RSA key generation and storage
Refresh rotation breaks	Race condition in token updates	Test concurrent refresh requests	Use database transactions and locking
Discovery endpoint 404	Routing configuration error	Check HTTP route registration	Ensure well-known path is registered

Error Handling and Edge Cases

Milestone(s): This section spans all milestones (1-4), establishing robust error handling patterns and security measures that must be implemented throughout the OAuth2 provider

Mental Model: Airport Security Checkpoint

Think of OAuth2 error handling like a multi-layered airport security system. Just as airport security has different checkpoints for different types of threats—document verification, metal detectors, baggage scanning—an OAuth2 provider needs different error handling layers for different types of failures. Each checkpoint has standardized procedures for handling problems: clear error codes, escalation paths, and recovery mechanisms. When something goes wrong, the system doesn't just fail silently—it provides specific, actionable feedback while maintaining security. A suspicious bag triggers a specific protocol, just as an invalid OAuth2 request triggers a standardized error response that helps the client application understand what went wrong and how to fix it.

The key insight is that error handling in OAuth2 isn't just about preventing crashes—it's about maintaining security boundaries while providing enough information for legitimate clients to succeed. Like airport security, you want to be helpful to legitimate travelers while being completely opaque to potential attackers.

OAuth2 Error Response Format

OAuth2 defines a standardized error response format that ensures consistent communication between authorization servers and client applications. This standardization is crucial because client applications need to programmatically handle errors and provide meaningful feedback to users.

The OAuth2 error response structure follows RFC 6749 specifications and includes several required and optional fields that provide different levels of detail about the failure. The `OAuth2Error` structure encapsulates all necessary information for proper error communication:

Field	Type	Required	Description
Error	string	Yes	Standard OAuth2 error code from RFC 6749
ErrorDescription	string	No	Human-readable description providing additional context
ErrorURI	string	No	URI pointing to documentation about the error
State	string	No	Exact state parameter value from the original request

The standardized error codes provide specific categories of failures that client applications can handle programmatically. Each error code represents a different class of problem with different recovery strategies:

Error Code	HTTP Status	Usage Context	Client Action
ErrorInvalidRequest	400	Missing or malformed parameters	Fix request parameters and retry
ErrorInvalidClient	401	Client authentication failed	Check client credentials
ErrorInvalidGrant	400	Authorization code/refresh token invalid	Restart authorization flow
ErrorUnauthorizedClient	400	Client not authorized for grant type	Check client registration
ErrorUnsupportedGrantType	400	Grant type not supported	Use supported grant type
ErrorInvalidScope	400	Scope invalid or exceeds registered scopes	Request valid scopes only
ErrorAccessDenied	403	User denied authorization request	Inform user, don't retry automatically
ErrorUnsupportedResponseType	400	Response type not supported	Use supported response type
ErrorServerError	500	Internal server error occurred	Retry with exponential backoff
ErrorTemporarilyUnavailable	503	Service temporarily overloaded	Retry with exponential backoff
ErrorInvalidToken	401	Access token invalid or expired	Refresh token or re-authenticate
ErrorInsufficientScope	403	Token lacks required scope	Request higher scope or inform user

Decision: Structured Error Response Format

- **Context:** OAuth2 providers must communicate errors clearly to client applications while maintaining security
- **Options Considered:** Custom error format, HTTP status codes only, OAuth2 standard format
- **Decision:** Implement OAuth2 RFC 6749 standard error format with structured JSON responses
- **Rationale:** Standard format ensures interoperability, provides programmatic error handling, and includes security considerations for error disclosure
- **Consequences:** Enables robust client error handling, maintains consistency with OAuth2 ecosystem, requires careful error message sanitization

The error response formatting process involves several critical steps to ensure both usability and security. When an error occurs, the system must determine the appropriate error code, construct a helpful but secure error description, and format the response according to OAuth2 standards.

Error response construction follows this algorithm:

1. Identify the specific failure condition from the request processing pipeline
2. Map the failure to the appropriate OAuth2 error code from the standard set
3. Determine the correct HTTP status code based on the error type and context
4. Construct a sanitized error description that helps legitimate clients without exposing sensitive information

5. Include the original state parameter if present to maintain CSRF protection
6. Format the response as JSON with proper Content-Type headers
7. Log the error with full context for debugging while returning sanitized information to the client

The `WriteOAuth2Error` function centralizes error response formatting to ensure consistency across all endpoints. This function takes the HTTP response writer, appropriate status code, error code, and description, then formats them according to OAuth2 specifications.

The critical insight is that OAuth2 error responses serve dual purposes: they must provide enough information for legitimate clients to recover from transient failures, while revealing nothing useful to potential attackers probing for vulnerabilities.

Error response formatting includes several security considerations that protect against information disclosure attacks. Error descriptions must be carefully crafted to avoid leaking sensitive information about internal system state, registered clients, or user accounts. For example, when a `client_id` doesn't exist, the error response should be identical to when client authentication fails, preventing enumeration attacks.

Security Edge Cases

OAuth2 providers face numerous security edge cases that require careful handling to prevent token theft, replay attacks, and other sophisticated threats. These edge cases often involve timing attacks, token reuse detection, and proper handling of cryptographic failures.

Token theft represents one of the most serious security threats to OAuth2 systems. When an attacker gains access to a valid access token or refresh token, they can impersonate the legitimate client or user. The system must implement several layers of defense to detect and mitigate token theft scenarios:

Threat Type	Detection Method	Mitigation Strategy	Recovery Action
Access Token Theft	Unusual usage patterns, geo-location	Short token lifetime, scope validation	Token revocation, user notification
Refresh Token Theft	Token family reuse detection	Refresh token rotation	Revoke entire token family
Authorization Code Interception	PKCE verification failure	Mandatory PKCE for public clients	Reject token exchange
Client Secret Compromise	Abnormal client behavior patterns	Client secret rotation	Invalidate all client tokens
Replay Attack	Token reuse detection	Nonce validation, timestamp checks	Reject duplicate requests

The refresh token rotation mechanism provides the strongest defense against token theft. When a refresh token is used to obtain new access tokens, the system issues a completely new refresh token and marks the old one as invalid. This creates a "token family" where each refresh operation creates a new generation of tokens linked to the previous ones.

Token family tracking algorithm works as follows:

1. When issuing the first refresh token, generate a unique token family identifier

2. Store the token family ID with the refresh token record in the database
3. When a refresh token is used, verify it belongs to a valid, active token family
4. Generate a new refresh token with the same family ID but new token value
5. Mark the old refresh token as used and store a reference to the new token
6. If an old refresh token is presented again, detect potential theft and revoke the entire family
7. Maintain family lineage until the last token in the family expires naturally

Decision: Refresh Token Rotation with Family Tracking

- **Context:** Refresh tokens are long-lived and valuable targets for attackers, requiring sophisticated theft detection
- **Options Considered:** Static refresh tokens, simple rotation, family-based rotation with theft detection
- **Decision:** Implement family-based refresh token rotation with automatic theft detection and family revocation
- **Rationale:** Provides strongest security against token theft while maintaining usability for legitimate clients
- **Consequences:** Requires complex token family management, enables automatic breach detection, may impact client implementations

Replay attack prevention requires careful handling of request uniqueness and timing constraints. The system must track request identifiers and enforce appropriate time windows for token exchange operations. Authorization codes must be single-use and short-lived, while refresh token usage must be monitored for suspicious patterns.

Timing attack mitigation is crucial for protecting against cryptographic side-channel attacks. Token validation, client secret verification, and other security-sensitive operations must use constant-time comparison functions to prevent timing-based information disclosure:

Timing Attack Mitigation Checklist:

1. Client secret verification must use constant-time comparison (`bcrypt.CompareHashAndPassword`)
2. Token signature verification must not leak timing information about validity
3. Authorization code lookup must take consistent time regardless of code validity
4. PKCE code verifier validation must use constant-time comparison
5. Database queries for security operations should have consistent response times
6. Error responses must be generated in consistent time regardless of error type

The system must also handle cryptographic failures gracefully without exposing information about the failure mode. When JWT signature verification fails, the error response should be identical whether the signature is invalid, the key is wrong, or the token is malformed. This prevents attackers from using error response variations to gain information about the cryptographic implementation.

Timing attacks are particularly dangerous in OAuth2 because they can be used to validate tokens, enumerate client credentials, or break PKCE protection. Every security-sensitive comparison must be implemented with constant-time algorithms.

Rate Limiting and Abuse Prevention

OAuth2 endpoints are attractive targets for various types of abuse including credential stuffing, token enumeration, and denial-of-service attacks. Effective rate limiting requires different strategies for different endpoints based on their usage patterns and security requirements.

The authorization endpoint faces unique challenges because legitimate users may retry authorization requests due to browser issues or user confusion. However, automated attacks against this endpoint can attempt to enumerate valid client_id values or overwhelm the consent screen infrastructure.

Rate limiting strategy varies by endpoint and request type:

Endpoint	Limiting Factor	Normal Rate	Burst Allowance	Abuse Threshold
Authorization	IP address + client_id	10/minute	3 burst	100/hour
Token Endpoint	Client credentials	60/minute	10 burst	1000/hour
Token Introspection	Client credentials	100/minute	20 burst	10000/hour
UserInfo	Access token	60/minute	10 burst	1000/hour
Client Registration	IP address	5/hour	2 burst	10/day
Token Revocation	Client credentials	30/minute	5 burst	500/hour

The rate limiting implementation uses a multi-layered approach with different algorithms for different scenarios. Token bucket algorithms work well for allowing burst traffic while maintaining overall rate limits, while sliding window algorithms provide more precise control over request timing.

Rate limiting algorithm for token endpoint requests:

1. Extract the client_id from the authenticated request (after client authentication succeeds)
2. Look up the current token bucket state for this client from the rate limiting store
3. Calculate tokens to add based on time elapsed since last request and configured refill rate
4. Add calculated tokens to bucket, capping at maximum bucket size (burst allowance)
5. Check if bucket contains sufficient tokens for this request (typically 1 token)
6. If sufficient tokens available, decrement bucket and allow request to proceed
7. If insufficient tokens, reject request with HTTP 429 and Retry-After header
8. Update bucket state in rate limiting store with new token count and timestamp

Distributed rate limiting presents additional challenges when the OAuth2 provider runs across multiple servers. The rate limiting state must be shared across instances while maintaining high performance and avoiding race conditions.

Decision: Redis-Based Distributed Rate Limiting

- **Context:** OAuth2 provider needs consistent rate limiting across multiple server instances
- **Options Considered:** In-memory per-instance, database-based, Redis-based distributed
- **Decision:** Implement Redis-based rate limiting with Lua scripts for atomic operations
- **Rationale:** Provides low-latency distributed state sharing with atomic increment operations and automatic expiration
- **Consequences:** Adds Redis dependency, enables consistent multi-instance rate limiting, requires careful Redis failover handling

Abuse detection goes beyond simple rate limiting to identify sophisticated attack patterns. The system monitors for suspicious behaviors like credential stuffing attempts, token enumeration attacks, and coordinated attacks from multiple

IP addresses.

Suspicious behavior detection patterns include:

Attack Pattern	Detection Signals	Response Strategy	Escalation Trigger
Credential Stuffing	High failure rate from single IP	Progressive delays, CAPTCHA	>50 failures/hour
Client Enumeration	Sequential client_id probing	IP-based blocking	>20 invalid clients
Token Brute Force	Rapid token validation attempts	Client rate limiting	>100 invalid tokens
Distributed Attack	Coordinated requests from multiple IPs	Geographic analysis	>1000 requests/minute
Account Takeover	Unusual location/device patterns	Additional authentication	Risk score >0.8

The system implements progressive response strategies that escalate from warnings to complete blocking based on the severity and persistence of suspicious behavior. Initial violations might trigger longer response delays or CAPTCHA challenges, while persistent abuse results in temporary or permanent IP blocking.

Anomaly detection uses statistical models to identify unusual request patterns that might indicate automated attacks. The system establishes baseline behavior for each client and endpoint, then flags significant deviations for investigation.

The key insight for abuse prevention is that legitimate OAuth2 traffic has predictable patterns—clients authenticate periodically, users authorize applications occasionally, and tokens are refreshed on schedule. Attackers typically generate traffic that violates these natural patterns.

Graceful Degradation

OAuth2 providers depend on various external systems including databases, cache layers, cryptographic services, and user authentication systems. When these dependencies fail, the provider must gracefully degrade functionality while maintaining security guarantees.

Dependency failure modes and their handling strategies require careful planning to ensure that partial failures don't compromise the security model. Some operations can continue with reduced functionality, while others must fail completely to maintain security boundaries.

Graceful degradation strategies by system component:

Component	Failure Mode	Degraded Operation	Security Impact	Recovery Action
Primary Database	Connection lost	Read-only from replica	None if replica current	Failover to replica
Redis Cache	Unavailable	Direct database queries	Performance only	Cache rebuilding
User Authentication	Service down	No new authorizations	Existing tokens valid	Queue auth requests
Key Management	Key unavailable	Token validation fails	High - stop issuing tokens	Emergency key rotation
Rate Limiting Store	Redis failure	Per-instance rate limiting	Reduced abuse protection	Restart with memory limits
Audit Logging	Log system down	Continue operations	Compliance risk	Queue logs for replay

Database failover represents the most critical degradation scenario because OAuth2 operations require persistent state for security. The system must distinguish between read operations that can use potentially stale replica data and write operations that require consistent primary database access.

Database degradation handling algorithm:

1. Detect primary database connection failure through health check or operation timeout
2. Evaluate current operation type—read-only (token introspection) vs. write-required (token issuance)
3. For read operations, attempt failover to database replica with staleness checks
4. For write operations, enter degraded mode and reject new token issuance requests
5. Continue serving existing valid tokens through replica database or cache
6. Monitor primary database recovery and gradually restore full functionality
7. Perform consistency checks after primary database recovery to detect any data loss

Cache failure handling focuses on maintaining performance while preserving correctness. When Redis or other caching layers become unavailable, the system falls back to direct database queries but must be careful about query load.

Decision: Fail-Safe Security Model

- **Context:** System dependencies will occasionally fail, requiring graceful degradation without security compromise
- **Options Considered:** Continue all operations, fail completely, selective degradation based on security impact
- **Decision:** Implement selective degradation that maintains security properties while reducing functionality
- **Rationale:** Preserves security boundaries while maximizing availability for low-risk operations
- **Consequences:** Requires complex failure mode analysis, enables high availability, may impact user experience during outages

Cryptographic service failures require immediate response because they directly impact security. When key management systems become unavailable or keys are compromised, the system must stop issuing new tokens while continuing to validate existing ones until they expire naturally.

Circuit breaker patterns help prevent cascading failures when external dependencies become unreliable. The system monitors dependency health and automatically stops sending requests to failing services while providing fallback responses.

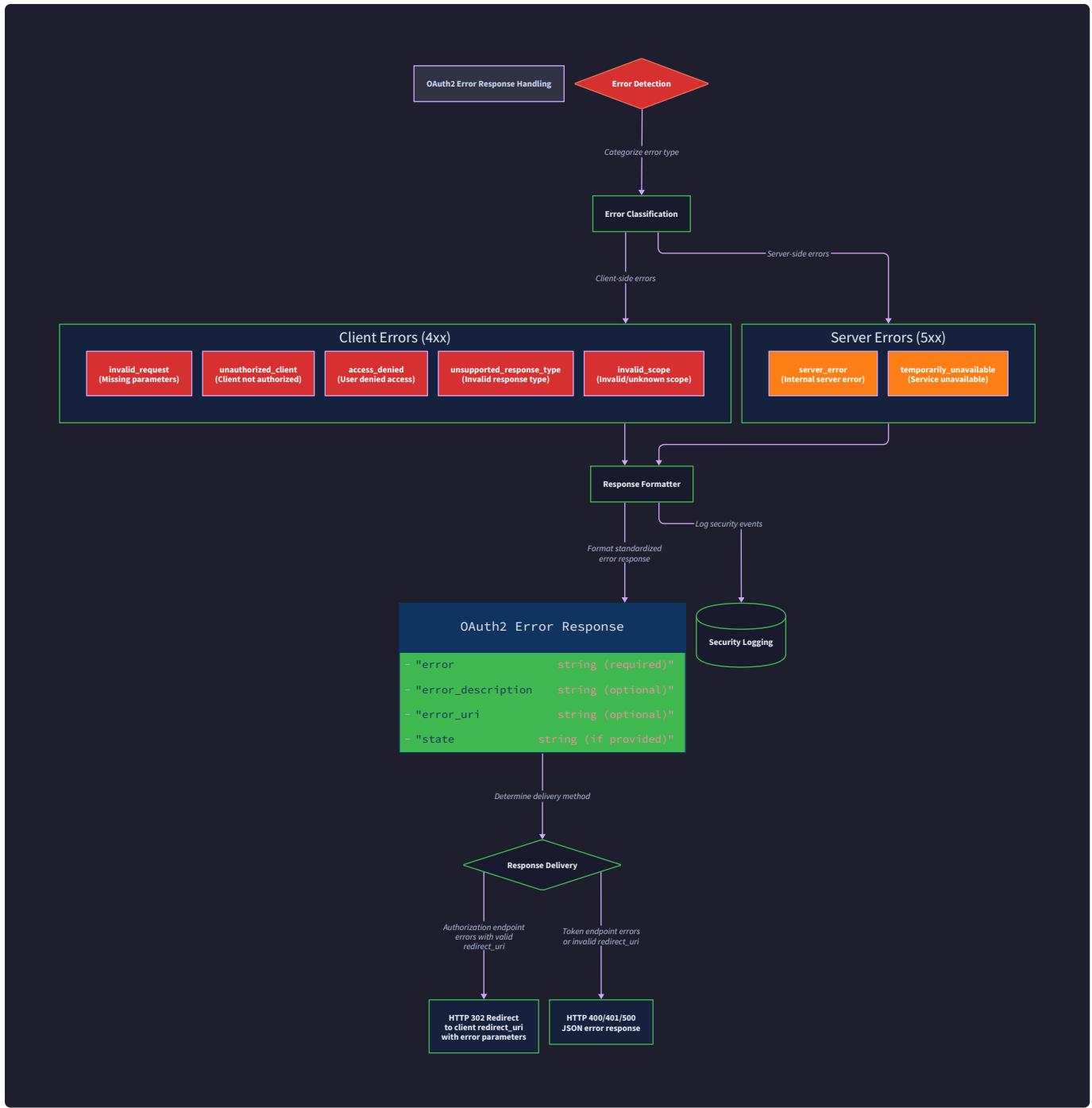
Circuit breaker state management:

1. **Closed State**: Normal operation, all requests pass through to dependency
2. **Open State**: Dependency failed, all requests immediately return failure response
3. **Half-Open State**: Testing recovery, limited requests allowed through to test dependency health
4. Monitor failure rate and response times to determine when to change circuit breaker state
5. Implement exponential backoff for recovery testing to avoid overwhelming recovering services
6. Provide meaningful error responses that help clients understand service degradation

The system must also handle partial functionality scenarios where some OAuth2 flows work while others don't. For example, if user authentication services are down, existing access tokens continue to work, but new authorization requests must be queued or rejected.

The fundamental principle of graceful degradation in OAuth2 is that security properties must never be compromised for availability. It's better to reject requests than to issue tokens with compromised security guarantees.

User communication during degraded operation helps maintain trust and reduces support burden. The system should provide clear error messages that explain service limitations and expected recovery times when possible.



Common Pitfalls

⚠ Pitfall: Information Disclosure in Error Messages

Many implementations inadvertently leak sensitive information through overly detailed error messages. For example, responding "client_id xyz does not exist" vs. "invalid client credentials" allows attackers to enumerate valid client identifiers. The error message should be identical whether the client doesn't exist or the secret is wrong.

Fix: Use generic error messages for authentication failures and implement detailed logging separately for debugging purposes. The `WriteOAuth2Error` function should sanitize all user-facing error descriptions.

⚠ Pitfall: Non-Constant Time Security Operations

Using standard string comparison for client secret validation or PKCE code verifier checking creates timing side-channels that can be exploited to extract secrets. Go's `==` operator returns immediately on the first differing byte, creating measurable timing differences.

Fix: Always use constant-time comparison functions like `subtle.ConstantTimeCompare` for security-sensitive operations. For password hashing, use `bcrypt.CompareHashAndPassword` which includes timing attack protection.

Pitfall: Insufficient Rate Limiting Granularity

Applying the same rate limiting rules to all endpoints or request types fails to account for different usage patterns and attack vectors. The authorization endpoint needs different limits than the token introspection endpoint.

Fix: Implement endpoint-specific rate limiting with appropriate limiting factors (IP, client_id, user_id) based on the endpoint's function and typical usage patterns.

Pitfall: Improper Error Code Selection

Using generic HTTP status codes without proper OAuth2 error codes prevents clients from implementing appropriate error handling. Returning HTTP 500 for all failures provides no actionable information.

Fix: Map specific failure conditions to appropriate OAuth2 error codes and HTTP status codes. Implement comprehensive error classification that helps clients determine whether to retry, fix parameters, or escalate to users.

Pitfall: Inadequate Token Family Cleanup

Implementing refresh token rotation without proper cleanup of old token families leads to database bloat and potential security issues. Old revoked tokens should be removed after they expire.

Fix: Implement background cleanup processes that remove expired revocation entries and old token family records. Set appropriate TTL values in Redis or database to automatically clean up stale data.

Pitfall: Synchronous External Dependency Calls

Making synchronous calls to external services (user authentication, audit logging) in the critical path creates failure points where external service issues directly impact OAuth2 operations.

Fix: Use asynchronous patterns for non-critical external calls, implement circuit breakers for external dependencies, and provide meaningful fallback responses when external services are unavailable.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Rate Limiting	In-memory token bucket (sync.Map)	Redis with Lua scripts
Circuit Breaker	Simple failure counter	go-kit circuit breaker
Error Logging	Standard log package	Structured logging (logrus/zap)
Metrics Collection	Simple counters	Prometheus metrics
Distributed Tracing	Manual request IDs	OpenTelemetry tracing
Cache Layer	Built-in map with cleanup	Redis cluster

Recommended File Structure

```
internal/oauth2/
  errors/
    oauth2_error.go      ← OAuth2Error struct and WriteOAuth2Error
    error_codes.go       ← Standard error code constants
    error_handler.go    ← Centralized error handling middleware
  ratelimit/
    ratelimiter.go       ← Rate limiting interface and implementations
    redis_limiter.go    ← Redis-based distributed rate limiter
    memory_limiter.go   ← In-memory fallback rate limiter
  security/
    timing_safe.go       ← Constant-time comparison utilities
    abuse_detector.go   ← Suspicious behavior detection
    circuit_breaker.go  ← External dependency protection
  health/
    health_checker.go   ← Dependency health monitoring
    graceful_shutdown.go ← Clean shutdown handling
```

Infrastructure Starter Code

```
// errors/oauth2_error.go - Complete OAuth2 error handling

package errors

import (
    "encoding/json"
    "net/http"
)

// OAuth2Error represents a standardized OAuth2 error response

type OAuth2Error struct {

    Error          string `json:"error"`
    ErrorDescription string `json:"error_description,omitempty"`
    ErrorURI       string `json:"error_uri,omitempty"`
    State          string `json:"state,omitempty"`
}

// Standard OAuth2 error codes

const (
    ErrorInvalidRequest      = "invalid_request"
    ErrorInvalidClient        = "invalid_client"
    ErrorInvalidGrant         = "invalid_grant"
    ErrorUnauthorizedClient   = "unauthorized_client"
    ErrorUnsupportedGrantType = "unsupported_grant_type"
    ErrorInvalidScope          = "invalid_scope"
    ErrorAccessDenied          = "access_denied"
    ErrorUnsupportedResponseType = "unsupported_response_type"
    ErrorServerError           = "server_error"
    ErrorTemporarilyUnavailable = "temporarily_unavailable"
    ErrorInvalidToken          = "invalid_token"
    ErrorInsufficientScope     = "insufficient_scope"
)
```

GO

```
)  
  
// WriteOAuth2Error writes a standardized OAuth2 error response  
  
func WriteOAuth2Error(w http.ResponseWriter, statusCode int, errorCode, description string) {  
  
    w.Header().Set("Content-Type", "application/json")  
  
    w.Header().Set("Cache-Control", "no-store")  
  
    w.Header().Set("Pragma", "no-cache")  
  
    w.WriteHeader(statusCode)  
  
  
    oauth2Err := OAuth2Error{  
  
        Error:           errorCode,  
  
        ErrorDescription: description,  
  
    }  
  
  
    json.NewEncoder(w).Encode(oauth2Err)  
}  
  
  
// security/timing_safe.go - Timing attack protection utilities  
  
package security  
  
  
import (  
  
    "crypto/subtle"  
  
    "golang.org/x/crypto/bcrypt"  
  
)  
  
  
// VerifyClientSecret performs constant-time client secret verification  
  
func VerifyClientSecret(secret, hash string) bool {  
  
    err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(secret))  
  
    return err == nil  
}  
  
  
// ConstantTimeStringEqual performs constant-time string comparison
```

```
func ConstantTimeStringEqual(a, b string) bool {  
    if len(a) != len(b) {  
        return false  
    }  
    return subtle.ConstantTimeCompare([]byte(a), []byte(b)) == 1  
}
```

Core Logic Skeleton Code

```
// ratelimit/ratelimiter.go - Rate limiting interface and implementation          GO

package ratelimit

import (
    "context"
    "time"
)

// RateLimiter defines the interface for rate limiting implementations

type RateLimiter interface {

    Allow(ctx context.Context, key string, limit int, window time.Duration) (bool, error)

    Reset(ctx context.Context, key string) error

}

// CheckEndpointRateLimit validates rate limits for OAuth2 endpoints

func CheckEndpointRateLimit(ctx context.Context, limiter RateLimiter, endpoint, clientID, ip string) error {

    // TODO 1: Determine rate limiting key based on endpoint type and request context

    // - Authorization endpoint: use IP + client_id

    // - Token endpoint: use client_id only

    // - UserInfo endpoint: use client_id

    // - Introspection: use client_id


    // TODO 2: Look up rate limiting configuration for the specific endpoint

    // - Different endpoints have different limits and windows

    // - Consider burst allowances vs. sustained rates


    // TODO 3: Check current request count against the limit

    // - Call limiter.Allow() with appropriate key and limits

    // - Handle distributed rate limiting across multiple servers
```

```
// TODO 4: If limit exceeded, return appropriate OAuth2 error

// - Use ErrorTemporarilyUnavailable for rate limiting

// - Include Retry-After header in HTTP response


// TODO 5: Log rate limiting events for monitoring and alerting

// - Track which clients are hitting limits

// - Monitor for distributed attack patterns


return nil

}

// security/abuse_detector.go - Suspicious behavior detection

package security

import (
    "context"
    "time"
)

// AbusePattern represents different types of suspicious behavior

type AbusePattern int

const (
    CredentialStuffing AbusePattern = iota
    ClientEnumeration
    TokenBruteForce
    DistributedAttack
)

// DetectSuspiciousBehavior analyzes request patterns for abuse

func DetectSuspiciousBehavior(ctx context.Context, clientID, ip string, endpoint string, success bool) ([]AbusePattern, error) {
```

```
// TODO 1: Collect request statistics for this client/IP combination

// - Track success/failure ratios over different time windows

// - Monitor request frequency and timing patterns

// - Store statistics in Redis or similar fast storage


// TODO 2: Apply statistical models to detect anomalies

// - Compare current behavior to established baselines

// - Look for sudden spikes in failure rates

// - Check for sequential client_id probing patterns


// TODO 3: Identify specific attack patterns

// - Credential stuffing: high failure rate from single IP

// - Client enumeration: sequential invalid client_ids

// - Token brute force: rapid token validation failures

// - Distributed attack: coordinated requests from multiple IPs


// TODO 4: Calculate risk scores and thresholds

// - Weight different suspicious behaviors appropriately

// - Consider historical data and reputation scores

// - Account for legitimate traffic patterns


// TODO 5: Return detected patterns for response escalation

// - Provide specific pattern types for targeted responses

// - Include confidence scores for each detection


return nil, nil

}

// health/health_checker.go - Dependency health monitoring

package health
```

```
import (
    "context"
    "database/sql"
    "time"
)

// HealthChecker monitors critical system dependencies

type HealthChecker struct {
    db      *sql.DB
    redis RedisClient
}

// CheckDatabaseHealth verifies database connectivity and performance

func (h *Healthchecker) CheckDatabaseHealth(ctx context.Context) error {
    // TODO 1: Test primary database connection with simple query
    // - Use SELECT 1 or similar lightweight query
    // - Set reasonable timeout (e.g., 5 seconds)
    // - Measure query response time for performance monitoring

    // TODO 2: Check database replica connectivity if applicable
    // - Verify read replica is available and current
    // - Measure replication lag to ensure data consistency

    // TODO 3: Test critical table accessibility
    // - Verify oauth_clients, oauth_tokens tables are accessible
    // - Check for any table lock issues or corruption

    // TODO 4: Monitor database connection pool status
    // - Check available connections in pool
    // - Monitor for connection leaks or exhaustion
```

```
// TODO 5: Return appropriate error if any checks fail  
  
    // - Provide specific error details for debugging  
  
    // - Log health check results for monitoring  
  
  
    return nil  
  
}
```

Language-Specific Hints

- Use `context.WithTimeout()` for all external service calls to prevent hanging requests
- Implement `http.Handler` middleware for rate limiting that wraps endpoint handlers
- Use `sync.Map` for high-concurrency in-memory caches with proper cleanup goroutines
- Redis Lua scripts ensure atomic rate limiting operations across distributed instances
- The `golang.org/x/time/rate` package provides good token bucket rate limiting for simple cases
- Use `crypto/subtle.ConstantTimeCompare` for all security-sensitive string comparisons
- Implement graceful shutdown with `context.CancelFunc` to clean up background processes

Milestone Checkpoints

After implementing error handling (spans all milestones):

- Run `curl -X POST http://localhost:8080/oauth2/token` with invalid parameters
- Verify response includes proper OAuth2 error codes and JSON format
- Check that error messages don't leak sensitive information
- Test rate limiting by making rapid requests and confirming HTTP 429 responses

After implementing security edge cases:

- Test refresh token rotation by using the same refresh token twice
- Verify entire token family gets revoked on token reuse detection
- Use timing measurement tools to confirm constant-time secret validation
- Test PKCE flows with invalid code verifiers to ensure proper validation

After implementing graceful degradation:

- Stop Redis/cache service and verify OAuth2 operations continue with database fallback
- Simulate database replica lag and confirm read operations handle staleness appropriately
- Test circuit breaker behavior by making external service unavailable
- Verify system recovers automatically when dependencies come back online

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Inconsistent rate limiting	Race conditions in distributed limiter	Check Redis key expiration and atomic operations	Use Lua scripts for atomic increment
Token validation timing varies	Non-constant time comparisons	Measure validation times with different inputs	Replace with subtle.ConstantTimeCompare
Error responses leak information	Overly detailed error messages	Review error messages from attacker perspective	Sanitize error descriptions
High memory usage over time	Rate limiting data not cleaned up	Monitor Redis/memory growth patterns	Implement TTL and cleanup processes
Circuit breaker stuck open	Health check not detecting recovery	Test dependency recovery detection logic	Adjust health check sensitivity
Token family revocation ineffective	Incomplete family tracking	Verify all tokens link to family correctly	Fix token family ID propagation

Testing Strategy

Milestone(s): This section provides comprehensive testing approaches for all milestones (1-4), establishing verification strategies that ensure security properties, protocol compliance, and integration correctness throughout the OAuth2 provider implementation

Mental Model: Multi-Layer Security Inspection

Think of testing an OAuth2 provider like conducting a comprehensive security inspection of a high-security facility with multiple layers of protection. Just as security inspectors must verify that each layer—from perimeter fences to biometric scanners to vault locks—functions correctly both individually and as an integrated system, OAuth2 testing requires verification at multiple levels.

The perimeter inspection corresponds to **protocol compliance testing**—ensuring that your OAuth2 provider correctly implements the standardized communication protocols that all clients expect. The security system testing represents **security property testing**—verifying that cryptographic operations, token handling, and access controls actually provide the security guarantees they promise. The operational readiness testing matches **integration testing**—confirming that real OAuth2 clients can successfully interact with your provider in realistic scenarios. Finally, the checkpoint inspections align with **milestone verification**—systematic validation that each phase of construction meets its acceptance criteria before proceeding to the next layer.

This multi-layer approach is essential because OAuth2 security failures can occur at any level. A provider might implement perfect cryptography but violate the OAuth2 specification, leading to client compatibility issues. Conversely, it might handle the OAuth2 protocol correctly but contain cryptographic vulnerabilities that enable token forgery.

Comprehensive testing ensures that security properties hold at every layer of the system.

Security Property Testing: Testing Cryptographic Properties and Attack Resistance

Security property testing for an OAuth2 provider focuses on verifying that the cryptographic foundations and security mechanisms actually provide their promised protection against real-world attacks. Unlike functional testing that verifies correct behavior under normal conditions, security testing specifically probes for vulnerabilities by simulating adversarial conditions and attack scenarios.

Cryptographic Property Verification

The foundation of OAuth2 security rests on cryptographic operations that must be tested for correctness, randomness quality, and resistance to cryptographic attacks. These tests verify that the mathematical and algorithmic security properties hold under implementation.

Token Generation Randomness Testing verifies that authorization codes, client secrets, and refresh tokens contain sufficient entropy to resist brute force and prediction attacks. The testing approach involves generating large samples of tokens and applying statistical randomness tests to detect patterns or bias. For authorization codes, collect samples of 10,000 generated codes and apply the NIST SP 800-22 statistical test suite to verify that the randomness passes frequency tests, runs tests, and serial correlation tests. Any detectable pattern indicates a vulnerability in the random number generation that could allow attackers to predict future tokens.

JWT Signature Verification Testing ensures that JWT access tokens cannot be forged or tampered with by attackers who lack the signing keys. Create test JWTs with various manipulations—altered payloads, modified headers, incorrect signatures—and verify that the `JWTValidator` correctly rejects all invalid tokens. Test edge cases like JWTs with valid signatures but from different keys, tokens with algorithm confusion attacks (changing "RS256" to "HS256"), and tokens with subtle payload modifications. The validator must reject every tampered token while accepting all legitimate tokens.

PKCE Security Property Testing verifies that the Proof Key for Code Exchange mechanism actually prevents authorization code interception attacks. Generate PKCE challenges and verifiers, then test that the `PKCEValidator` correctly accepts only the matching verifier for each challenge. Test attack scenarios where an attacker intercepts an authorization code but lacks the code verifier—the token exchange must fail. Also test that slight modifications to the code verifier (single character changes, case modifications) result in validation failures.

Timing Attack Resistance Testing ensures that cryptographic operations use constant-time comparisons to prevent timing side-channel attacks. Implement timing measurement tests for critical operations like client secret verification and token comparison. The `VerifyClientSecret` function must take the same time to reject an incorrect secret as it takes to accept a correct one. Measure execution times over thousands of iterations with both correct and incorrect inputs—any statistically significant timing difference indicates a timing attack vulnerability.

Test Category	Security Property	Test Method	Failure Indicators
Token Randomness	Unpredictability	NIST SP 800-22 statistical tests on 10K+ samples	Patterns, bias, or predictability in token generation
JWT Integrity	Tamper resistance	Signature verification with modified payloads/headers	Acceptance of tampered or forged tokens
PKCE Binding	Authorization code protection	Verifier validation with intercepted codes	Code exchange without valid verifier
Timing Attacks	Side-channel resistance	Statistical timing analysis of crypto operations	Timing differences revealing secret information
Key Management	Key security	Private key access and rotation testing	Unauthorized key access or stale key usage

Attack Simulation Testing

Attack simulation testing systematically attempts common OAuth2 attacks against your provider to verify that security mechanisms effectively prevent real-world threats. These tests should be automated and run regularly to catch security regressions.

Authorization Code Replay Attack Testing simulates attackers attempting to reuse intercepted authorization codes. Generate a valid authorization code through the normal flow, then attempt to exchange it for tokens multiple times. The first exchange should succeed and return valid tokens, but subsequent attempts must fail with `ErrorInvalidGrant`. Additionally, test that codes expire after their configured timeout (typically 10 minutes) and cannot be used after expiration.

Token Theft and Misuse Testing simulates scenarios where attackers obtain valid tokens through various means and attempt to misuse them. Test that revoked tokens immediately become unusable across all endpoints, including the `UserInfo` endpoint and any resource servers using your JWT validation. Test that expired tokens are rejected with proper error codes. Test that tokens issued to one client cannot be used by a different client by attempting cross-client token usage.

Refresh Token Family Attack Testing simulates the critical refresh token theft detection mechanism. Create a refresh token family by issuing an initial refresh token and rotating it several times. Then simulate an attack by attempting to use a previously rotated (old) refresh token. The system must detect this as potential theft and revoke the entire token family, invalidating all related tokens. Verify that legitimate token rotation continues to work while detecting and preventing family replay attacks.

Client Impersonation Attack Testing attempts various forms of OAuth2 client impersonation to verify client authentication mechanisms. Test scenarios where attackers attempt to use legitimate client IDs with incorrect secrets, stolen client credentials from one environment in another, and various client credential manipulation attacks. All unauthorized client authentication attempts must fail with `ErrorInvalidClient` without revealing information about valid client IDs.

PKCE Bypass Attack Testing specifically targets the PKCE implementation with various bypass attempts. Test scenarios where public clients attempt to skip PKCE verification, use weak code challenges, submit incorrect code challenge methods, or attempt to replay code challenges from other authorization flows. The authorization endpoint must enforce PKCE for public clients and reject any bypass attempts.

Critical Security Insight: Security property testing must assume an intelligent adversary who understands OAuth2 specifications and common implementation vulnerabilities. Tests should not just verify correct behavior but actively attempt to break security assumptions through edge cases and malicious inputs.

Security Test Implementation Strategy

Security tests require specialized infrastructure to generate attack scenarios, measure cryptographic properties, and simulate adversarial conditions. The testing framework should include utilities for token manipulation, timing measurements, and statistical analysis.

Create a dedicated security testing suite that runs independently of functional tests. This suite should include test utilities for generating malicious inputs, measuring response times with microsecond precision, and analyzing token distributions for randomness properties. The security tests should run in a dedicated testing environment that can simulate network delays, partial failures, and concurrent attack scenarios without affecting development workflows.

Test Data Generation for Security Tests requires creating large datasets of tokens, authorization codes, and cryptographic challenges for statistical analysis. Implement generators that can produce thousands of test tokens with controlled variations for randomness testing. Create test databases with realistic client registrations, user accounts, and token histories to support attack simulation scenarios.

Security Test Metrics and Reporting should provide clear indicators of security posture and track security properties over time. Implement metrics that measure token entropy, cryptographic operation timing consistency, and attack resistance rates. Security test failures should provide detailed forensic information about the specific vulnerability discovered, including reproducible attack scenarios and recommended remediation steps.

Protocol Compliance Testing: Validating OAuth2 and OIDC Specification Adherence

Protocol compliance testing ensures that your OAuth2 provider correctly implements the standardized specifications that OAuth2 clients expect. Unlike security testing that focuses on adversarial scenarios, compliance testing verifies correct behavior according to RFC 6749 (OAuth2), RFC 7636 (PKCE), RFC 7662 (Token Introspection), RFC 7009 (Token Revocation), and the OpenID Connect Core specification.

OAuth2 Specification Compliance Testing

OAuth2 compliance testing systematically verifies that each endpoint correctly handles the request and response formats, error conditions, and behavioral requirements specified in the relevant RFCs. This testing is critical because OAuth2 clients are built to expect specific protocol behaviors, and deviations can cause integration failures or security vulnerabilities.

Authorization Endpoint Compliance Testing verifies that the authorization endpoint correctly implements the OAuth2 authorization code grant flow according to RFC 6749 Section 4.1. Test all required parameters (`response_type`, `client_id`, `redirect_uri`, `scope`, `state`) and verify that missing required parameters result in proper error responses. Test the optional `state` parameter handling—when provided by the client, it must be returned unchanged in both success and error responses.

The authorization endpoint must validate redirect URIs according to exact string matching requirements. Test scenarios with legitimate redirect URIs, URIs that are subpaths or subdomains of registered URIs (which should be rejected), URIs with additional query parameters (which should be rejected), and completely invalid URIs. The endpoint must reject authorization requests with redirect URIs that don't exactly match registered values.

Token Endpoint Compliance Testing ensures that token exchanges follow RFC 6749 Section 4.1.3 requirements precisely. Test the authorization code grant with all required parameters (`grant_type`, `code`, `redirect_uri`, `client_id`) and verify that client authentication works correctly. For confidential clients, test both HTTP Basic authentication and request body client credentials. For public clients, verify that client authentication is not required but client identification is still validated.

Test refresh token grant compliance according to RFC 6749 Section 6. When processing refresh token grants, verify that the system issues new access tokens with the same or reduced scope, optionally issues new refresh tokens (implementing token rotation), and properly handles refresh token expiration and revocation.

Client Credentials Grant Compliance Testing verifies machine-to-machine authentication according to RFC 6749 Section 4.4. Test that the client credentials grant bypasses user authorization and issues tokens directly to authenticated clients. Verify that the issued tokens have appropriate scope limitations and cannot access user-specific resources without explicit user consent.

Endpoint	RFC Requirement	Test Scenario	Expected Behavior
Authorization	Parameter validation (RFC 6749 4.1.1)	Missing <code>client_id</code> parameter	HTTP 400 with <code>invalid_request</code> error
Authorization	Redirect URI validation (RFC 6749 3.1.2)	URI substring of registered URI	Reject with error page (no redirect)
Token	Authorization code validation (RFC 6749 4.1.3)	Expired authorization code	HTTP 400 with <code>invalid_grant</code> error
Token	Client authentication (RFC 6749 2.3)	Invalid client credentials	HTTP 401 with <code>invalid_client</code> error
Token	Refresh token grant (RFC 6749 6)	Valid refresh token exchange	New access token with same/reduced scope

PKCE Specification Compliance Testing

PKCE compliance testing verifies implementation according to RFC 7636, which defines critical security extensions for OAuth2 authorization code flows. PKCE compliance is mandatory for public clients and should be supported for all client types.

PKCE Challenge and Verification Testing ensures that code challenges and verifiers are handled correctly throughout the authorization flow. Test that the authorization endpoint accepts and stores code challenges with both `S256` and `plain` challenge methods, though production systems should enforce `S256` for security. When the token endpoint processes authorization code exchanges, verify that the system correctly validates code verifiers against the stored code challenge using the specified challenge method.

Test PKCE parameter validation thoroughly. The `code_challenge` parameter must be a base64url-encoded string of appropriate length for the challenge method. The `code_verifier` must be a cryptographically random string between 43 and 128 characters in length. Test boundary conditions with verifiers of exactly 43 characters, exactly 128 characters, and invalid lengths outside this range.

PKCE Enforcement Policy Testing verifies that your provider correctly enforces PKCE requirements based on client type and configuration. For public clients (mobile apps, single-page applications), PKCE should be mandatory—

authorization requests without valid PKCE parameters should be rejected. For confidential clients, PKCE may be optional but should still be validated when present.

OpenID Connect Specification Compliance Testing

OpenID Connect compliance testing ensures that the identity layer built on top of OAuth2 correctly implements OIDC Core specifications. This includes ID token generation, UserInfo endpoint behavior, and discovery document format.

ID Token Generation Compliance Testing verifies that ID tokens contain required claims and follow JWT formatting requirements. Test that ID tokens include mandatory claims (`iss`, `sub`, `aud`, `exp`, `iat`, `auth_time`, `nonce`) when appropriate. The `sub` claim must be locally unique and never reassigned within your provider. The `aud` claim must contain the client ID that requested the ID token. The `nonce` parameter from the authorization request must be included in the ID token if provided.

UserInfo Endpoint Compliance Testing ensures that the UserInfo endpoint returns user profile claims according to OIDC specifications. Test that the endpoint requires valid access tokens and returns user information only for the authenticated user (identified by the `sub` claim in the access token). Verify that the returned claims match the granted scopes—requests with `profile` scope should return profile-related claims, while requests with `email` scope should return email-related claims.

Discovery Document Compliance Testing verifies that the well-known configuration endpoint returns a valid OIDC discovery document according to OIDC Discovery specifications. Test that required fields (`issuer`, `authorization_endpoint`, `token_endpoint`, `userinfo_endpoint`, `jwks_uri`) are present and contain correct URLs. Test that the supported grant types, response types, and scopes are accurately declared.

Protocol Compliance Critical Point: OAuth2 and OIDC specifications include numerous optional features and edge cases that real clients depend on. Compliance testing must cover not just the "happy path" scenarios but also error conditions, edge cases, and optional parameter handling that clients may rely on.

Compliance Test Framework Implementation

Protocol compliance testing requires systematic coverage of specification requirements and automated validation of request/response formats. Implement a compliance testing framework that can parse OAuth2 specifications and generate comprehensive test cases for each requirement.

Specification-Driven Test Generation creates test cases directly from RFC requirements and OIDC specification sections. For each endpoint, enumerate all required parameters, optional parameters, and error conditions specified in the relevant standards. Generate positive test cases that verify correct behavior and negative test cases that verify proper error handling for each requirement.

Response Format Validation Testing ensures that all endpoint responses exactly match the formats specified in OAuth2 and OIDC standards. Implement JSON schema validation for token responses, error responses, and UserInfo responses. Validate HTTP status codes, Content-Type headers, and Cache-Control headers according to specification requirements.

Milestone Checkpoints: Verification Steps for Each Implementation Milestone

Milestone checkpoints provide structured verification that each phase of OAuth2 provider development meets its acceptance criteria before proceeding to the next milestone. These checkpoints combine automated testing, manual verification, and integration testing to ensure comprehensive validation of implemented functionality.

Milestone 1 Checkpoint: Client Registration & Authorization Endpoint

The first milestone checkpoint verifies that client registration and authorization endpoint functionality works correctly with proper security properties. This checkpoint establishes the foundation for all subsequent OAuth2 flows.

Client Registration Verification tests that the `RegistrationService` correctly creates OAuth2 clients with proper credential generation and storage. Verify that the `RegisterClient` function generates cryptographically secure client IDs and client secrets, properly hashes client secrets using bcrypt, and stores client information with correct redirect URI associations.

Create test scenarios with various client registration inputs—single redirect URI, multiple redirect URIs, different client names, and edge cases like extremely long names or URIs. Verify that the registration process validates redirect URIs for proper format and rejects obviously invalid URIs like localhost (in production configurations) or URIs with user credentials.

Test the client credential verification process by attempting authentication with correct client credentials, incorrect secrets, non-existent client IDs, and various malformed credential formats. The `VerifyClientSecret` function must use constant-time comparison to prevent timing attacks while correctly accepting valid credentials and rejecting invalid ones.

Authorization Endpoint Verification tests the complete authorization flow from initial redirect through user consent to authorization code generation. Start by testing parameter validation—send authorization requests with various combinations of required and optional parameters. Verify that requests with missing `client_id`, `response_type`, or `redirect_uri` parameters result in appropriate error responses.

Test redirect URI validation thoroughly by sending authorization requests with redirect URIs that exactly match registered URIs (should succeed), URIs that are subpaths or subdomains of registered URIs (should fail), and URIs with additional query parameters or fragments (should fail). The authorization endpoint must implement exact string matching for redirect URI validation.

Verify PKCE implementation by sending authorization requests with valid PKCE parameters (`code_challenge` and `code_challenge_method`), invalid PKCE parameters, and missing PKCE parameters for public clients. The system should store PKCE challenges correctly and enforce PKCE requirements based on client type configuration.

Authorization Code Generation Verification tests that authorization codes are generated securely and bound to the correct parameters. Generate authorization codes through the consent flow and verify that each code is cryptographically random, unique, and properly associated with the client ID, redirect URI, and user ID. Test that authorization codes expire after the configured timeout period and cannot be reused after a successful token exchange.

Verification Step	Test Method	Success Criteria	Failure Indicators
Client registration	API call with valid parameters	Client record created with hashed secret	Registration fails or secret stored in plaintext
Redirect URI validation	Authorization request with various URIs	Only exact matches accepted	Partial matches accepted or valid URIs rejected
PKCE parameter handling	Authorization request with PKCE parameters	Challenge stored and bound to authorization code	PKCE parameters ignored or incorrectly validated
Authorization code binding	Token exchange with generated code	Code validates against correct client/URI	Code accepts incorrect client or redirect URI

Milestone 2 Checkpoint: Token Endpoint & JWT Generation

The second milestone checkpoint verifies that token generation, JWT signing, and refresh token rotation work correctly. This milestone establishes the core token-based security mechanisms for the OAuth2 provider.

Token Exchange Verification tests that the token endpoint correctly exchanges authorization codes for access and refresh tokens. Send token exchange requests with valid authorization codes and verify that the response includes a properly formatted access token (JWT), refresh token, token type ("Bearer"), and expiration information. Test that the authorization code becomes invalid after successful exchange and cannot be reused.

Verify client authentication during token exchange by testing with correct client credentials (both HTTP Basic and request body formats), incorrect credentials, and missing credentials for confidential clients. Public clients should be able to exchange tokens without client authentication but must still provide client identification for validation.

Test PKCE verification during token exchange by providing correct code verifiers, incorrect verifiers, and missing verifiers for authorization codes that were issued with PKCE challenges. The token endpoint must validate code verifiers against stored code challenges and reject exchanges with invalid or missing verifiers.

JWT Access Token Verification tests that generated access tokens are properly formatted JWTs with correct claims and signatures. Parse generated access tokens and verify that they contain required claims (`sub`, `iss`, `aud`, `exp`, `iat`, `scope`) with correct values. The `sub` claim should identify the user who granted authorization, `aud` should identify the client that will use the token, and `scope` should reflect the granted permissions.

Test JWT signature validation using your `JWTValidator` service. Verify that legitimate tokens are accepted and provide correct user and client identification. Test that tokens with modified payloads, invalid signatures, or expired timestamps are rejected with appropriate error codes.

Refresh Token Rotation Verification tests that refresh token grants issue new tokens while implementing security through token family tracking. Use a refresh token to obtain new access and refresh tokens, then verify that the old refresh token becomes invalid (if implementing rotation) and the new tokens are properly formatted and functional.

Test refresh token family security by attempting to reuse an old refresh token after it has been rotated. If your implementation includes token family tracking for theft detection, verify that reusing an old refresh token triggers family revocation and invalidates all related tokens.

Client Credentials Grant Verification tests machine-to-machine authentication for confidential clients. Send client credentials grant requests with valid client credentials and verify that access tokens are issued without user involvement. Test that these tokens have appropriate scope limitations and work correctly with protected resources.

Milestone 3 Checkpoint: Token Introspection & Revocation

The third milestone checkpoint verifies that token introspection and revocation endpoints work correctly according to RFC 7662 and RFC 7009 specifications. This milestone establishes token lifecycle management and validation capabilities.

Token Introspection Verification tests that the introspection endpoint returns correct token metadata according to RFC 7662 format. Send introspection requests with valid access tokens and verify that the response includes `active: true` along with token metadata like `sub`, `client_id`, `exp`, and `scope`. Test with expired tokens and verify that `active: false` is returned without additional metadata.

Test client authentication for the introspection endpoint by sending requests with correct client credentials, incorrect credentials, and missing authentication. The introspection endpoint should be protected and require client authentication to prevent unauthorized token validation.

Verify JWT validation integration by testing introspection of JWT access tokens that have been modified, expired, or revoked. The introspection endpoint should correctly identify invalid JWTs and return `active: false` for compromised tokens.

Token Revocation Verification tests that the revocation endpoint immediately invalidates tokens according to RFC 7009. Revoke valid access and refresh tokens, then verify that subsequent introspection requests return `active: false` and attempts to use the tokens with protected resources fail with authentication errors.

Test revocation endpoint client authentication and verify that only authorized clients can revoke tokens. Test scenarios where clients attempt to revoke tokens that were issued to other clients—these requests should be rejected to prevent unauthorized token manipulation.

Token Family Revocation Verification tests advanced security features for refresh token theft detection. If implementing token families, test that revoking a refresh token also revokes related tokens in the same family. Test that detected refresh token replay attacks trigger automatic family revocation to prevent token theft exploitation.

Milestone 4 Checkpoint: UserInfo Endpoint & Consent Management

The fourth milestone checkpoint verifies that the UserInfo endpoint and consent management functionality work correctly according to OIDC specifications. This milestone completes the OpenID Connect identity provider implementation.

UserInfo Endpoint Verification tests that the UserInfo endpoint returns user profile claims based on granted scopes. Send UserInfo requests with access tokens that have different scope grants (`profile`, `email`, `openid`) and verify that only appropriate claims are returned. Tokens with `profile` scope should return profile-related claims like `name` and `picture`, while tokens with `email` scope should return email-related claims like `email` and `email_verified`.

Test UserInfo endpoint authentication by sending requests with valid access tokens, expired tokens, revoked tokens, and invalid tokens. Only valid access tokens should result in successful UserInfo responses with appropriate user claims.

Consent Management Verification tests that user consent decisions are properly stored and respected in subsequent authorization requests. Grant consent for specific scopes with a particular client, then initiate new authorization requests with the same client and scopes—users should not be prompted for consent again for previously granted permissions.

Test consent revocation functionality by allowing users to revoke previously granted consent, then verify that subsequent authorization requests re-prompt for consent approval. Test that partial consent revocation (revoking some but not all granted scopes) works correctly.

Scope-to-Claims Mapping Verification tests that the `ClaimsMapper` service correctly filters user profile information based on granted scopes. Create test users with comprehensive profile information, then request UserInfo with various scope combinations and verify that only authorized claims are returned.

Test edge cases like requesting scopes that weren't granted during authorization, using access tokens with reduced scopes (compared to the original authorization), and handling custom scopes that map to organization-specific user claims.

Milestone	Core Verification	Security Verification	Integration Verification
Milestone 1	Client registration and authorization codes	PKCE implementation and redirect URI validation	Authorization flow with test OAuth2 client
Milestone 2	JWT generation and refresh token rotation	Token binding and cryptographic signatures	Token exchange with multiple client types
Milestone 3	Introspection and revocation endpoints	Token family security and theft detection	Resource server integration with JWT validation
Milestone 4	UserInfo endpoint and consent persistence	Scope-based claims filtering	Complete OIDC flow with real identity clients

Integration Testing: End-to-End Testing with Real OAuth2 Clients

Integration testing validates that your OAuth2 provider works correctly with real OAuth2 and OpenID Connect clients in realistic deployment scenarios. Unlike unit tests that verify individual components or compliance tests that check specification adherence, integration tests focus on end-to-end workflows and real-world compatibility.

Real Client Integration Testing

Real client integration testing uses actual OAuth2 client libraries and applications to validate that your provider works with existing OAuth2 ecosystem components. This testing catches integration issues that might not appear in isolated component tests.

OAuth2 Client Library Integration tests your provider with popular OAuth2 client libraries in different programming languages. Set up test environments using libraries like `golang.org/x/oauth2` for Go, `requests-oauthlib` for Python, `passport-oauth2` for Node.js, and `Spring Security OAuth2` for Java. Each library may have different implementation assumptions or edge case handling that reveals compatibility issues.

Create integration test scenarios where these client libraries attempt complete OAuth2 flows against your provider. Test authorization code flows with PKCE, client credentials flows, and refresh token flows. Verify that the client libraries can parse your token responses, handle your error responses appropriately, and successfully obtain and use access tokens for protected resource access.

Single-Page Application Integration tests your provider with JavaScript-based OAuth2 clients that run in web browsers. Deploy test single-page applications that use libraries like `oauth2-pkce` or `oidc-client-js` to perform OAuth2 flows with your provider. These clients are particularly sensitive to CORS configuration, redirect URI handling, and PKCE implementation details.

Test browser-based authorization flows including popup-based authorization, redirect-based authorization, and iframe-based silent token renewal. Verify that your authorization endpoint correctly handles JavaScript origins, that your token endpoint supports appropriate CORS headers, and that generated tokens work correctly with browser-based resource access.

Mobile Application Integration tests your provider with native mobile OAuth2 clients on iOS and Android platforms. Mobile clients have specific requirements for redirect URI handling (custom URL schemes, universal links), PKCE enforcement, and token storage security. Set up test mobile applications that implement OAuth2 flows using platform-specific OAuth2 libraries.

Test mobile-specific OAuth2 patterns including custom URL scheme redirects, universal link redirects, and system browser-based authorization flows. Verify that your authorization endpoint correctly validates mobile redirect URLs and that generated tokens work appropriately with mobile application lifecycles and background/foreground transitions.

Enterprise SSO Integration tests your provider with enterprise single sign-on solutions and identity federation systems. Many organizations use OAuth2 providers as identity sources for enterprise applications, SAML identity providers, and other federation scenarios. Test integration scenarios where your OAuth2 provider serves as an identity source for enterprise SSO systems.

Create test configurations where enterprise SSO solutions like Auth0, Okta, or Azure Active Directory can federate identity from your OAuth2 provider. Test user provisioning, attribute mapping, and session management across federated identity scenarios.

Resource Server Integration Testing

Resource server integration testing verifies that APIs and services can successfully validate tokens issued by your OAuth2 provider and enforce appropriate access controls based on token content.

JWT Validation Integration tests that resource servers can validate JWT access tokens using your public keys and JWT validation libraries. Deploy test resource servers that use JWT validation libraries like `jwt-go` for Go, `PyJWT` for Python, or `jsonwebtoken` for Node.js. These servers should fetch your public keys from your JWKS endpoint and validate incoming access tokens.

Test various token validation scenarios including valid tokens (should grant access), expired tokens (should deny access), tampered tokens (should deny access), and revoked tokens (should deny access after checking revocation status). Verify that resource servers correctly extract user identity, client identity, and scope information from validated tokens.

Introspection-Based Validation Integration tests resource servers that use token introspection for validation instead of local JWT validation. Deploy test resource servers that call your introspection endpoint to validate opaque tokens or JWT tokens. Test that these servers correctly handle introspection responses and make appropriate access control decisions based on the returned token metadata.

Test introspection integration scenarios including successful validation (active tokens), failed validation (inactive tokens), and error handling (network failures, authentication errors). Resource servers should gracefully handle introspection endpoint unavailability and implement appropriate caching strategies for validation results.

Scope-Based Authorization Integration tests that resource servers correctly enforce authorization based on token scopes. Create test APIs with different endpoints that require different scope combinations, then test access using tokens with various scope grants. Verify that tokens with insufficient scopes are rejected while tokens with appropriate scopes grant access to authorized resources.

Test complex scope scenarios including hierarchical scopes (where `admin` scope grants `user` scope access), scope intersection requirements (endpoints that require multiple specific scopes), and dynamic scope evaluation (where scope requirements change based on request context).

Load and Performance Integration Testing

Load testing validates that your OAuth2 provider maintains correct behavior and security properties under realistic traffic loads and concurrent usage patterns.

Authorization Flow Load Testing simulates concurrent authorization flows from multiple clients and users. Generate test scenarios with hundreds of simultaneous authorization requests, consent decisions, and token exchanges. Measure

response times, error rates, and system resource utilization during peak load scenarios.

Test load scenarios that stress different components—concurrent client registrations, simultaneous authorization code generation, parallel token exchanges, and concurrent UserInfo requests. Verify that your provider maintains security properties (no token reuse, proper randomness) even under heavy concurrent load.

Token Validation Load Testing simulates high-frequency token validation scenarios that resource servers typically generate. Create test scenarios with thousands of concurrent JWT validation operations, introspection requests, and revocation checks. Measure the performance impact of revocation list checks and JWKS key fetching under load.

Test caching effectiveness by monitoring cache hit rates, cache invalidation performance, and memory usage during high-frequency validation scenarios. Verify that caching improves performance without compromising security properties like immediate token revocation.

Database and Storage Load Testing validates that your provider's data storage systems can handle realistic usage patterns without compromising data consistency or security. Test concurrent client registrations, authorization code generation, token storage, and consent record management under load.

Monitor database performance metrics including connection pool utilization, query performance, and lock contention during high-concurrency scenarios. Test database failure scenarios and verify that your provider handles database unavailability gracefully without compromising security or losing critical data.

Integration Testing Critical Insight: Real-world OAuth2 deployments face complex integration challenges including network failures, partial system outages, clock skew between systems, and varying client implementation quality. Integration testing must simulate these realistic conditions to validate production readiness.

Implementation Guidance

This section provides concrete implementation approaches for building comprehensive testing infrastructure that validates security properties, protocol compliance, and integration scenarios throughout OAuth2 provider development.

Technology Recommendations

Testing Category	Simple Option	Advanced Option
Unit Testing Framework	Go's built-in <code>testing</code> package with <code>testify/assert</code>	Go's <code>testing</code> with <code>ginkgo</code> BDD framework and <code>gomega</code> matchers
Integration Testing	HTTP client tests with <code>net/http/httpptest</code>	Full containerized test environments with <code>docker-compose</code>
Security Testing	Custom crypto validation with <code>crypto/rand</code> analysis	Specialized tools like <code>gosec</code> static analysis and fuzzing with <code>go-fuzz</code>
Load Testing	Simple concurrent goroutines with <code>sync.WaitGroup</code>	Professional tools like <code>k6</code> or <code>JMeter</code> with detailed metrics
Protocol Compliance	Custom JSON schema validation	OpenAPI/Swagger-based contract testing with <code>go-swagger</code>
Database Testing	In-memory SQLite for isolated tests	PostgreSQL with transaction rollback for realistic testing
JWT Testing	Manual JWT parsing with <code>dgrijalva/jwt-go</code>	Comprehensive JWT testing with <code>lestrrat-go/jwx</code>
Mock Services	Manual interface mocking	Automated mock generation with <code>golang/mock</code>

Recommended File Structure

```
project-root/
  cmd/
    server/main.go           ← production server entry point
  internal/
    oauth2/
      client.go              ← OAuth2 provider core
      token.go
      auth.go
  testing/
    testutil/
      crypto.go               ← shared testing infrastructure
      http.go                 ← testing utilities and helpers
      database.go             ← cryptographic testing utilities
      fixtures/
        clients.json          ← HTTP testing helpers
        users.json             ← database testing setup
        tokens.json            ← test data and fixtures
    compliance/
      oauth2_test.go          ← test OAuth2 client data
      oidc_test.go             ← test user profile data
      pkce_test.go             ← sample JWT tokens
    security/
      crypto_test.go          ← protocol compliance test suite
      attacks_test.go         ← OAuth2 RFC compliance tests
      timing_test.go           ← OpenID Connect compliance tests
      pkce_test.go             ← PKCE specification tests
    integration/
      clients_test.go          ← security property test suite
      load_test.go              ← cryptographic property tests
      e2e_test.go               ← attack simulation tests
      timing_test.go            ← timing attack resistance tests
    test/
      docker-compose.test.yml   ← integration test suite
      mock-clients/
        spa/                   ← real client integration tests
        mobile/                ← load and performance tests
        server/                ← end-to-end workflow tests
  test/
    docker-compose.test.yml   ← external testing infrastructure
    mock-clients/
      spa/                   ← test environment containers
      mobile/                ← mock OAuth2 client applications
      server/                ← single-page application test client
                                ← mobile application test client
                                ← server-side application test client
```

Security Testing Infrastructure

Complete cryptographic testing utilities that provide statistical randomness analysis, timing attack detection, and attack simulation capabilities:

GO

```
// internal/testing/testutil/crypto.go

package testutil

import (
    "crypto/rand"
    "crypto/sha256"
    "encoding/base64"
    "math"
    "time"
)

// CryptoTestSuite provides comprehensive cryptographic testing utilities

type CryptoTestSuite struct {
    samples []string
    timings []time.Duration
}

func NewCryptoTestSuite() *CryptoTestSuite {
    return &CryptoTestSuite{
        samples: make([]string, 0),
        timings: make([]time.Duration, 0),
    }
}

// CollectRandomnessSamples gathers token samples for statistical analysis

func (c *CryptoTestSuite) CollectRandomnessSamples(generator func() (string, error), count int) error {
    // TODO: Generate 'count' samples using the provided generator function
    // TODO: Store samples in c.samples slice for later analysis
    // TODO: Return error if any generation fails
    // Hint: Use a loop to call generator() multiple times
}
```

```

// AnalyzeRandomnessDistribution performs statistical tests on collected samples

func (c *CryptoTestSuite) AnalyzeRandomnessDistribution() (*RandomnessReport, error) {

    // TODO: Perform frequency analysis on character distribution

    // TODO: Check for repeating patterns or sequences

    // TODO: Calculate entropy metrics for randomness quality

    // TODO: Return detailed report with pass/fail indicators

    // Hint: Look for uniform character distribution and absence of patterns

}

// MeasureTimingConsistency tests for timing attack vulnerabilities

func (c *CryptoTestSuite) MeasureTimingConsistency(operation func(input string) bool, correctInput,
incorrectInput string, iterations int) (*TimingReport, error) {

    // TODO: Measure execution time for correct input over 'iterations'

    // TODO: Measure execution time for incorrect input over 'iterations'

    // TODO: Perform statistical analysis to detect timing differences

    // TODO: Return report indicating whether timing is consistent

    // Hint: Use time.Now() before/after operation calls for precision

}

// SimulateTokenReplayAttack tests authorization code and refresh token replay protection

func (c *CryptoTestSuite) SimulateTokenReplayAttack(tokenExchangeFunc func(code string) error,
validCode string) *AttackResult {

    // TODO: Attempt to use valid authorization code multiple times

    // TODO: Record whether second usage is properly rejected

    // TODO: Test with expired codes and revoked tokens

    // TODO: Return result indicating attack success/failure

    // Hint: First usage should succeed, subsequent uses should fail

}

type RandomnessReport struct {

    EntropyBits      float64

    UniformityScore float64
}

```

```
    PatternCount     int
    PassesTests     bool
}

type TimingReport struct {
    CorrectInputAvg   time.Duration
    IncorrectInputAvg time.Duration
    TimingDifference  time.Duration
    VulnerableToAttack bool
}

type AttackResult struct {
    AttackType      string
    Successful      bool
    Details         string
    Mitigation     string
}
```

Protocol Compliance Testing Framework

Complete OAuth2/OIDC specification testing that validates RFC compliance and standard protocol behavior:

GO

```
// internal/testing/compliance/oauth2_test.go

package compliance

import (
    "encoding/json"

    "net/http"

    "net/url"

    "testing"

    "your-project/internal/oauth2"
)

// OAuth2ComplianceTestSuite runs comprehensive RFC 6749 compliance tests

type OAuth2ComplianceTestSuite struct {

    providerURL string

    httpClient *http.Client

    testClients []TestClient
}

func NewOAuth2ComplianceTestSuite(providerURL string) *OAuth2ComplianceTestSuite {

    return &OAuth2ComplianceTestSuite{
        providerURL: providerURL,

        httpClient: &http.Client{Timeout: 30 * time.Second},

        testClients: LoadTestClients(),
    }
}

// TestAuthorizationEndpointCompliance validates authorization endpoint according to RFC 6749 Section
4.1.1

func (suite *OAuth2ComplianceTestSuite) TestAuthorizationEndpointCompliance(t *testing.T) {

    // TODO: Test required parameter validation (response_type, client_id, redirect_uri)

    // TODO: Test optional parameter handling (scope, state)

    // TODO: Test redirect URI validation with exact string matching
}
```

```

// TODO: Test error response format according to RFC 6749 Section 4.1.2.1

// TODO: Verify state parameter is returned in both success and error responses

// Hint: Send HTTP GET requests to /oauth2/authorize endpoint with various parameter combinations

}

// TestTokenEndpointCompliance validates token endpoint according to RFC 6749 Section 4.1.3

func (suite *OAuth2ComplianceTestSuite) TestTokenEndpointCompliance(t *testing.T) {

    // TODO: Test authorization code grant with all required parameters

    // TODO: Test client authentication using HTTP Basic and request body methods

    // TODO: Test refresh token grant according to RFC 6749 Section 6

    // TODO: Test client credentials grant according to RFC 6749 Section 4.4

    // TODO: Verify token response format includes access_token, token_type, expires_in

    // Hint: Send HTTP POST requests to /oauth2/token endpoint with proper Content-Type

}

// TestPKCECompliance validates PKCE implementation according to RFC 7636

func (suite *OAuth2ComplianceTestSuite) TestPKCECompliance(t *testing.T) {

    // TODO: Test code challenge parameter validation (base64url encoding, proper length)

    // TODO: Test code verifier validation (43-128 characters, unreserved characters)

    // TODO: Test S256 challenge method (SHA256 hashing with base64url encoding)

    // TODO: Test PKCE parameter storage and verification across authorization/token flow

    // TODO: Verify PKCE enforcement for public clients

    // Hint: Generate code verifier, create challenge, test complete PKCE flow

}

// TestErrorResponseCompliance validates error handling according to OAuth2 specifications

func (suite *OAuth2ComplianceTestSuite) TestErrorResponseCompliance(t *testing.T) {

    // TODO: Test invalid_request error with missing required parameters

    // TODO: Test unauthorized_client error with invalid client credentials

    // TODO: Test invalid_grant error with expired or invalid authorization codes

    // TODO: Test unsupported_grant_type error with unsupported grant types

```

```

// TODO: Verify error response includes error, error_description, error_uri fields

// Hint: Intentionally send malformed requests and validate error response format

}

type TestClient struct {

    ClientID      string   `json:"client_id"`

    ClientSecret  string   `json:"client_secret"`

    RedirectURIs []string `json:"redirect_uris"`

    GrantTypes    []string `json:"grant_types"`

    ClientType    string   `json:"client_type"` // "confidential" or "public"

}

func LoadTestClients() []TestClient {

    // TODO: Load test client configurations from fixtures/clients.json

    // TODO: Include various client types (confidential, public, different grant types)

    // TODO: Return slice of TestClient structs for use in compliance tests

    // Hint: Use json.Unmarshal to parse client fixture data

}

```

Integration Testing Infrastructure

Complete end-to-end integration testing that validates real-world OAuth2 client compatibility:

GO

```
// internal/testing/integration/e2e_test.go

package integration

import (
    "context"
    "net/http"
    "testing"
    "time"
)

// IntegrationTestSuite provides end-to-end OAuth2 flow testing

type IntegrationTestSuite struct {

    providerURL    string
    clientManager  *TestClientManager
    userManager    *TestUserManager
    httpClient     *http.Client
}

func NewIntegrationTestSuite(providerURL string) *IntegrationTestSuite {
    return &IntegrationTestSuite{
        providerURL:    providerURL,
        clientManager:  NewTestClientManager(),
        userManager:    NewTestUserManager(),
        httpClient:     &http.Client{Timeout: 30 * time.Second},
    }
}

// TestCompleteAuthorizationCodeFlow tests full OAuth2 authorization code flow

func (suite *IntegrationTestSuite) TestCompleteAuthorizationCodeFlow(t *testing.T) {
    ctx := context.Background()

    // TODO: Register test OAuth2 client
```

```

    // TODO: Initiate authorization request with proper parameters

    // TODO: Simulate user consent decision

    // TODO: Exchange authorization code for tokens

    // TODO: Use access token to call UserInfo endpoint

    // TODO: Refresh access token using refresh token

    // TODO: Verify all steps succeed with proper security properties

    // Hint: Each step should validate response format and security properties

}

// TestSinglePageApplicationFlow tests browser-based OAuth2 with PKCE

func (suite *IntegrationTestSuite) TestSinglePageApplicationFlow(t *testing.T) {

    // TODO: Create public client configuration for SPA

    // TODO: Generate PKCE code verifier and challenge

    // TODO: Initiate authorization flow with PKCE parameters

    // TODO: Complete authorization and token exchange

    // TODO: Verify tokens work correctly without client secret

    // TODO: Test token refresh and revocation for public clients

    // Hint: Public clients must use PKCE and cannot authenticate with secrets

}

// TestClientCredentialsFlow tests machine-to-machine authentication

func (suite *IntegrationTestSuite) TestClientCredentialsFlow(t *testing.T) {

    // TODO: Create confidential client for machine-to-machine access

    // TODO: Request access token using client credentials grant

    // TODO: Verify token contains appropriate client identity and scopes

    // TODO: Test token validation and resource access

    // TODO: Verify tokens cannot access user-specific resources

    // Hint: Client credentials tokens should not have user context (no sub claim)

}

// TestTokenRevocationAndIntrospection tests token lifecycle management

```

```

func (suite *IntegrationTestSuite) TestTokenRevocationAndIntrospection(t *testing.T) {

    // TODO: Generate valid access and refresh tokens

    // TODO: Verify tokens are active via introspection endpoint

    // TODO: Revoke tokens via revocation endpoint

    // TODO: Verify revoked tokens show as inactive

    // TODO: Test that revoked tokens fail resource access

    // TODO: Test refresh token family revocation for security

    // Hint: Revocation should be immediate and affect all related tokens

}

type TestClientManager struct {

    registeredClients map[string]*RegisteredClient
}

func NewTestClientManager() *TestClientManager {

    return &TestClientManager{

        registeredClients: make(map[string]*RegisteredClient),
    }
}

// RegisterTestClient creates OAuth2 client for testing purposes

func (m *TestClientManager) RegisterTestClient(clientType string, name string, redirectURIs []string) (*RegisteredClient, error) {

    // TODO: Call client registration endpoint with test parameters

    // TODO: Store returned client credentials for test usage

    // TODO: Return RegisteredClient with ID, secret, and configuration

    // Hint: Use HTTP POST to /oauth2/register endpoint

}

type RegisteredClient struct {

    ClientID      string

    ClientSecret  string
}

```

```

    RedirectURIs []string

    ClientType string

}

type TestUserManager struct {

    testUsers map[string]*TestUser
}

func NewTestUserManager() *TestUserManager {
    return &TestUserManager{
        testUsers: make(map[string]*TestUser),
    }
}

// CreateTestUser sets up user account for testing flows

func (m *TestUserManager) CreateTestUser(username, email string) (*TestUser, error) {
    // TODO: Create test user account with profile information

    // TODO: Set up authentication credentials for consent testing

    // TODO: Return TestUser with identity and authentication details

    // Hint: May need to call user management API or direct database setup
}

type TestUser struct {

    UserID string

    Username string

    Email string

    Profile map[string]interface{}
}

```

Milestone Verification Checkpoints

Each milestone includes specific verification steps that validate acceptance criteria before proceeding to the next phase:

Milestone 1 Verification Script:

```
#!/bin/bash

# Milestone 1: Client Registration & Authorization Endpoint Verification

echo "==> Milestone 1 Verification ==>"

# Test client registration

echo "Testing client registration..."

go test ./internal/oauth2/client_test.go -v -run TestClientRegistration

# Test authorization endpoint

echo "Testing authorization endpoint..."

go test ./internal/oauth2/auth_test.go -v -run TestAuthorizationEndpoint

# Test PKCE implementation

echo "Testing PKCE implementation..."

go test ./internal/testing/compliance/pkce_test.go -v

# Security verification

echo "Running security tests..."

go test ./internal/testing/security/crypto_test.go -v -run TestAuthorizationCodeSecurity

echo "Milestone 1 verification complete. Check for any test failures above."
```

BASH

Milestone 2 Verification Script:

```
#!/bin/bash

# Milestone 2: Token Endpoint & JWT Generation Verification

echo "==> Milestone 2 Verification ==>"

# Test token endpoint functionality

echo "Testing token endpoint..."

go test ./internal/oauth2/token_test.go -v -run TestTokenEndpoint

# Test JWT generation and validation

echo "Testing JWT tokens..."

go test ./internal/oauth2/jwt_test.go -v

# Test refresh token rotation

echo "Testing refresh token rotation..."

go test ./internal/oauth2/refresh_test.go -v

# Integration test with real client

echo "Running integration tests..."

go test ./internal/testing/integration/e2e_test.go -v -run TestAuthorizationCodeFlow

echo "Milestone 2 verification complete. Check for any test failures above."
```

BASH

Security Property Validation Commands:

```
# Run comprehensive security test suite
# Check for timing attack vulnerabilities
# Validate cryptographic randomness
# Test attack resistance
go test ./internal/testing/security/... -v
go test ./internal/testing/security/timing_test.go -v -timeout 5m
go test ./internal/testing/security/crypto_test.go -v -run TestRandomnessProperties
go test ./internal/testing/security/attacks_test.go -v
```

BASH

Load Testing Commands:

```
# Simple concurrent load test
# Advanced load testing with k6 (if available)
k6 run test/load-tests/oauth2-flow.js --vus 50 --duration 2m
```

BASH

These verification checkpoints ensure that each milestone meets its acceptance criteria and security requirements before proceeding to the next phase of development.

Debugging Guide

Milestone(s): This section provides diagnostic techniques and debugging strategies for all milestones (1-4), establishing systematic approaches to identify, diagnose, and resolve issues that arise during OAuth2 provider implementation

Mental Model: Medical Diagnosis System

Think of debugging an OAuth2 provider like a systematic medical diagnosis process. Just as a doctor follows a structured approach — gathering symptoms, running diagnostic tests, reviewing past medical history, and applying proven treatments — debugging OAuth2 implementations requires methodical investigation. The "symptoms" are user reports and error logs, the "diagnostic tests" are specialized OAuth2 debugging tools and protocol analyzers, the "medical history" is audit trails and token lifecycle tracking, and the "treatments" are specific fixes for well-known OAuth2 security and implementation issues. Like medicine, OAuth2 debugging benefits from pattern recognition — experienced practitioners quickly recognize common symptom clusters and know which diagnostic approaches yield the fastest path to resolution.

The debugging process becomes particularly crucial in OAuth2 systems because security vulnerabilities often manifest as subtle behavioral anomalies rather than obvious crashes. A timing attack vulnerability might only be detectable through careful response time analysis, while a token binding flaw could allow privilege escalation without generating obvious error messages. This section establishes systematic diagnostic techniques that help developers identify both obvious functional bugs and subtle security vulnerabilities that could compromise the entire authorization system.

Common Implementation Bugs

OAuth2 provider implementations exhibit predictable failure patterns that stem from the complex interaction between cryptographic operations, protocol state management, and security enforcement. Understanding these common bugs enables faster diagnosis and prevention of issues that could compromise security or break client application integrations.

Symptom	Root Cause	Diagnostic Steps	Fix Implementation
Authorization codes accepted multiple times	Missing single-use enforcement in code validation	Check database for <code>AuthorizationCode.Used</code> field updates	Set <code>Used = true</code> atomically during token exchange
PKCE bypass allows code interception	Code verifier validation skipped or incorrect	Test with invalid code_verifier in token request	Implement proper SHA256 challenge verification
Refresh token families not revoked on theft	Token reuse detection missing or incomplete	Monitor for refresh token usage after rotation	Revoke entire <code>TokenFamily</code> when old token reused
Client secrets compared with timing vulnerability	Using <code>==</code> operator instead of constant-time comparison	Measure response times for correct vs incorrect secrets	Use <code>subtle.ConstantTimeCompare</code> for all secret validation
JWT tokens accepted after explicit revocation	Revocation list not checked during validation	Test token access after calling revocation endpoint	Query <code>RevocationStore</code> during <code>JWTValidator.ValidateToken</code>
Authorization codes bound to wrong redirect URI	URI validation logic flawed or missing	Submit code exchange with different redirect_uri	Store and validate <code>RedirectURI</code> in <code>AuthorizationCode</code>
State parameter validation bypassed	CSRF protection implementation incomplete	Omit state parameter or use predictable values	Generate cryptographically random state, validate exact match
Token endpoint accepts expired authorization codes	Expiration checking missing or incorrect	Submit token request with old authorization code	Check <code>AuthorizationCode.ExpiresAt</code> before token issuance
Scope escalation in token exchange	Granted scopes exceed originally requested scopes	Request broader scopes in token exchange than authorization	Copy exact <code>Scope</code> from <code>AuthorizationCode</code> to tokens
Client authentication bypassed entirely	Client credential validation missing	Submit token requests without client authentication	Validate client credentials before processing grant requests
Access tokens lack proper audience validation	JWT aud claim missing or not validated	Use access token with unintended resource server	Include and validate <code>aud</code> claim in <code>JWTGenerator</code>

Symptom	Root Cause	Diagnostic Steps	Fix Implementation
Refresh token rotation creates orphaned tokens	Database cleanup logic incomplete	Monitor database growth after token rotations	Implement <code>RevokeTokenFamily</code> with cascade deletion
UserInfo endpoint leaks unauthorized claims	Scope-to-claims filtering bypassed	Request UserInfo with limited scopes, check response	Filter claims through <code>ClaimsMapper.FilterClaims</code>
Rate limiting ineffective against distributed attacks	Rate limits applied per-IP instead of per-client	Coordinate attack from multiple IP addresses	Implement composite rate limiting by <code>client_id</code> and IP
Consent decisions not persisted correctly	<code>ConsentRecord</code> creation or retrieval logic flawed	Grant consent, refresh page, observe re-prompting	Store consent with proper <code>UserID</code> and <code>ClientID</code> binding

⚠ Pitfall: Assuming OAuth2 Errors Are Always Client Mistakes

Many developers assume that OAuth2 protocol violations always indicate client implementation errors, leading them to focus debugging efforts on client applications rather than their own provider implementation. This assumption delays resolution because the actual bug often lies in the authorization server's parameter validation, token generation, or protocol state management. For example, when clients report "invalid_grant" errors during code exchange, the issue might be the provider's authorization code expiration logic rather than client timing problems.

The correct debugging approach systematically validates the provider's implementation first — checking authorization code generation, storage, and validation logic — before concluding that client applications are at fault. This prevents debugging dead ends and ensures that subtle provider bugs don't get misclassified as integration issues.

⚠ Pitfall: Overlooking Race Conditions in Token Operations

OAuth2 implementations often contain race conditions that only manifest under concurrent load, making them difficult to reproduce in development environments. Common examples include authorization codes being marked as used after validation but before token generation (allowing double redemption), refresh token rotation logic that doesn't properly handle concurrent requests from the same client, and client registration operations that can create duplicate `client_id` values under specific timing conditions.

These race conditions require specific testing approaches that simulate concurrent operations rather than sequential test cases. Debugging them effectively requires understanding the critical sections in token lifecycle management and implementing proper database transaction isolation to prevent state corruption.

Debugging Techniques

Effective OAuth2 debugging requires specialized techniques that account for the protocol's cryptographic nature, multi-step flows, and security-sensitive operations. Standard application debugging approaches often fall short because

OAuth2 bugs frequently involve subtle timing issues, cryptographic validation failures, and protocol state mismatches that don't generate obvious error messages.

Structured Logging Strategy

OAuth2 providers should implement comprehensive audit logging that captures every significant protocol operation with sufficient context for post-incident analysis. The logging strategy must balance security (avoiding credential exposure) with diagnostic utility (providing enough detail for effective debugging).

Event Type	Required Fields	Security Considerations	Retention Period
Authorization Request	<code>client_id</code> , <code>redirect_uri</code> , <code>scope</code> , <code>state</code> hash, <code>code_challenge</code> hash, timestamp	Never log full <code>state</code> or <code>code_challenge</code> — use SHA256 hashes	30 days
Authorization Code Generation	<code>client_id</code> , <code>user_id</code> , code hash, <code>redirect_uri</code> , <code>scope</code> , expiration time	Log code hash, never plaintext authorization code	30 days
Token Exchange Request	<code>client_id</code> , <code>grant_type</code> , code hash (for authorization code grant), <code>scope</code>	Never log <code>client_secret</code> or <code>code_verifier</code> in plaintext	90 days
Token Generation Success	<code>client_id</code> , <code>user_id</code> , <code>token_id</code> (JWT jti claim), <code>scope</code> , token type, expiration	Log JWT ID for revocation tracking, never full token	1 year
Token Validation	<code>token_id</code> , validation result, <code>scope</code> requested, client context	Include validation failure reasons for debugging	30 days
Token Revocation	<code>token_id</code> , <code>client_id</code> , revocation timestamp, reason code	Track both explicit revocations and family revocations	1 year
Consent Decisions	<code>user_id</code> , <code>client_id</code> , <code>scope</code> , grant/deny decision, timestamp	Essential for consent management debugging	2 years
Security Events	Failed authentication attempts, rate limit violations, suspicious patterns	Include IP addresses and user agents for attack analysis	6 months

Design Insight: OAuth2 audit logs serve dual purposes as debugging tools and security incident response evidence. The logging design must ensure that legitimate debugging scenarios have sufficient information while preventing credential leakage that could compromise security if logs are exposed.

Protocol State Inspection Tools

OAuth2 debugging benefits from specialized inspection tools that can decode and validate protocol state at each step of the authorization flow. These tools help identify exactly where protocol violations occur and whether they stem from client behavior or provider implementation bugs.

Authorization Code Flow State Inspection

- Request Parameter Validation:** Create debugging middleware that logs the complete parameter set for authorization requests, highlighting any missing required parameters, invalid parameter combinations, or security violations like missing PKCE parameters for public clients.

2. **Code Generation Verification:** Implement debug endpoints (disabled in production) that can verify authorization code binding to specific clients, redirect URIs, and PKCE challenges without consuming the code for token exchange.
3. **Token Exchange Parameter Tracing:** Log detailed token exchange requests showing client authentication method, grant type parameters, and the complete parameter validation sequence that led to success or failure.
4. **JWT Token Inspection:** Provide debugging utilities that can decode JWT access tokens (without validating signatures) to inspect claims, expiration times, and scope grants for manual verification.

Database State Consistency Checks

OAuth2 providers maintain complex state relationships between clients, authorization codes, tokens, and consent records. Debugging often requires verifying that these relationships remain consistent and that cleanup operations properly remove expired or revoked state.

Consistency Check	Query Pattern	Expected Result	Common Inconsistencies
Authorization Code Cleanup	Count codes where <code>ExpiresAt < NOW()</code> and <code>Used = false</code>	Should be 0 with proper cleanup	Expired codes not cleaned, blocking future requests
Refresh Token Family Integrity	Verify all tokens in family have same <code>TokenFamily</code> ID	All family members consistent	Orphaned tokens with incorrect family assignments
Consent Record Validity	Check consent expiration vs token issuance times	Active tokens have valid consent	Tokens issued after consent expiration
Revocation List Accuracy	Cross-reference revoked <code>token_id</code> with active token usage	No active usage of revoked tokens	Revoked tokens still accepted by validation
Client Registration Integrity	Verify all <code>AuthorizationCode</code> records reference existing clients	No dangling client references	Authorization codes for deleted clients

Cryptographic Operation Debugging

OAuth2 security depends entirely on proper cryptographic operations for client authentication, PKCE verification, JWT signing, and token validation. Debugging cryptographic failures requires systematic verification of each cryptographic operation in isolation.

PKCE Challenge Verification Testing

PKCE implementation bugs are notoriously difficult to debug because the SHA256 hashing operation either works perfectly or fails completely, with little middle ground. Effective PKCE debugging requires test cases that verify the complete challenge generation and verification pipeline.

1. **Code Verifier Generation Testing:** Generate multiple code verifiers and verify they meet RFC 7636 requirements for length (43-128 characters) and character set (unreserved characters only).
2. **Challenge Generation Testing:** For each valid code verifier, generate the corresponding SHA256 challenge using base64url encoding and verify the result matches expected test vectors.
3. **Challenge Verification Testing:** Test the complete verification pipeline with known good verifier/challenge pairs, known bad pairs, and edge cases like incorrect encoding or malformed challenges.

4. **Attack Resistance Testing:** Verify that challenge verification properly rejects attempts to bypass PKCE using empty challenges, incorrect challenge methods, or malformed base64url encoding.

JWT Signature Verification Debugging

JWT signature validation failures can stem from key management issues, algorithm mismatches, or subtle encoding problems that are difficult to diagnose without systematic verification.

1. **Key Material Verification:** Verify that the `KeyManager` correctly loads and provides the expected public/private key pairs for JWT signing and verification operations.
2. **Algorithm Consistency Testing:** Ensure that JWT headers specify the same signing algorithm (`alg` claim) that the `JWTGenerator` actually uses for signature generation.
3. **Claims Validation Testing:** Verify that required claims (`iss`, `aud`, `exp`, `iat`, `sub`) are correctly generated and validated according to the application's security requirements.
4. **Cross-Implementation Testing:** Use external JWT libraries to verify that tokens generated by the OAuth2 provider can be validated by standard JWT implementations, ensuring interoperability.

Security Issue Debugging

OAuth2 security vulnerabilities often manifest as subtle behavioral anomalies that require specialized debugging techniques to identify and analyze. Unlike functional bugs that cause obvious failures, security issues may allow systems to operate normally while creating exploitable attack vectors.

Timing Attack Detection

Timing attacks exploit variations in cryptographic operation duration to extract sensitive information like client secrets or authorization codes. Detecting timing vulnerabilities requires careful measurement and statistical analysis of operation durations under different input conditions.

Constant-Time Comparison Verification

The most critical timing attack vector in OAuth2 implementations involves client secret verification during client authentication. Debugging timing vulnerabilities requires systematic measurement of comparison operations with correct and incorrect inputs.

Timing Attack Detection Protocol:

1. Measure 1000+ secret comparison operations with correct `client_secret`
2. Measure 1000+ secret comparison operations with incorrect `client_secret`
3. Calculate mean response times for both groups
4. Apply statistical significance testing (t-test) to response time distributions
5. Flag timing differences > 10 microseconds as potential vulnerabilities
6. Verify that bcrypt comparison provides consistent timing regardless of input correctness

PKCE Verification Timing Analysis

PKCE code challenge verification can also be vulnerable to timing attacks if the SHA256 hashing and comparison operations don't use constant-time algorithms. The debugging approach systematically tests challenge verification timing across different input patterns.

Test Case	Input Pattern	Expected Timing	Vulnerability Indicator
Correct Verifier	Valid code verifier matching stored challenge	$T \pm \sigma$ (consistent)	Timing variance < 5%
Wrong Length Verifier	Code verifier with incorrect length	$T \pm \sigma$ (consistent)	Early rejection detectable by timing
Wrong Content Verifier	Code verifier with correct length, wrong content	$T \pm \sigma$ (consistent)	Content comparison timing leak
Malformed Verifier	Invalid base64url encoding in verifier	$T \pm \sigma$ (consistent)	Parsing error timing differences
Empty Verifier	Missing code_verifier parameter	$T \pm \sigma$ (consistent)	Validation shortcut timing

Security Insight: Timing attack detection requires statistical analysis because single timing measurements can vary significantly due to system load, garbage collection, and other environmental factors. The debugging process must collect sufficient samples to distinguish genuine timing vulnerabilities from random timing variation.

Token Binding Verification

OAuth2 security relies on proper token binding — ensuring that authorization codes, access tokens, and refresh tokens are cryptographically or logically bound to the specific client and context that should be authorized to use them. Debugging token binding issues requires systematic verification that tokens cannot be used outside their intended context.

Authorization Code Binding Tests

Authorization codes must be bound to the specific client that initiated the authorization request and the exact redirect URI specified in that request. Debugging code binding requires testing various code interchange scenarios.

- Cross-Client Code Exchange:** Generate authorization code for Client A, attempt to exchange it using Client B's credentials. Should fail with `invalid_grant` error.
- Redirect URI Modification:** Generate authorization code with `redirect_uri` "<https://app.example.com/callback>", attempt exchange with `redirect_uri` "<https://app.example.com/callback/modified>". Should fail validation.
- PKCE Challenge Binding:** Generate authorization code with PKCE challenge, attempt exchange with different `code_verifier`. Should fail PKCE verification.
- Code Replay Detection:** Successfully exchange authorization code once, attempt to exchange the same code again. Should fail with single-use violation.

Refresh Token Family Tracking

Refresh token security depends on proper family tracking that can detect token theft and revoke entire token families when suspicious usage patterns are identified.

Attack Scenario	Test Implementation	Expected Behavior	Security Failure Indicator
Token Reuse After Rotation	Use old refresh token after successful rotation	Entire family revoked	Old token still accepted
Concurrent Token Usage	Submit simultaneous requests with same refresh token	One succeeds, others fail	Multiple successful responses
Family Revocation Bypass	Revoke token family, attempt to use different family member	All family tokens rejected	Any family token still works
Cross-Client Token Usage	Use refresh token generated for Client A with Client B	Authentication failure	Token accepted by wrong client
Stolen Token Detection	Simulate attacker using intercepted refresh token	Family revocation triggered	No security response

Privilege Escalation Detection

OAuth2 implementations can be vulnerable to privilege escalation attacks where clients obtain broader access than originally authorized through scope manipulation, token substitution, or authorization bypasses.

Scope Escalation Testing

Scope escalation occurs when clients receive access tokens with broader permissions than they requested or were authorized to receive. Debugging scope escalation requires systematic testing of scope handling throughout the authorization flow.

- 1. Authorization Request Scope Tracking:** Record exact scope values in authorization requests and verify they are preserved unchanged through code generation and token exchange.
- 2. Consent Screen Scope Verification:** Verify that consent screens display only the scopes actually requested, and that user consent decisions are accurately captured and enforced.
- 3. Token Exchange Scope Limitation:** Ensure that access tokens contain only scopes that were present in the original authorization code, preventing scope expansion during token exchange.
- 4. UserInfo Endpoint Scope Enforcement:** Test that UserInfo endpoint returns claims only for scopes present in the access token, not broader scopes the client might have been authorized for in the past.

Client Privilege Verification

Different OAuth2 clients should have different privilege levels based on their registration parameters, grant types, and trust levels. Debugging client privilege violations requires testing cross-client authorization scenarios.

Privilege Test	Setup	Attack Vector	Expected Defense
Public Client PKCE Bypass	Register public client without client_secret	Submit authorization without PKCE	Request rejected
Confidential Client Impersonation	Register confidential client with client_secret	Submit request without secret	Authentication failure
Grant Type Restriction	Register client with specific grant_types	Attempt unauthorized grant type	Grant type rejected
Redirect URI Validation	Register client with specific redirect_uris	Use unauthorized redirect_uri	URI validation failure
Scope Limitation	Register client with limited allowed scopes	Request broader scopes	Scope request rejected

Performance Debugging

OAuth2 providers must maintain consistent performance under varying loads while ensuring that security operations don't create performance bottlenecks that could be exploited for denial of service attacks. Performance debugging requires understanding the computational cost of cryptographic operations and identifying optimization opportunities that don't compromise security.

Token Operation Profiling

OAuth2 token operations involve expensive cryptographic computations that can become performance bottlenecks under high load. Performance debugging requires systematic profiling of each token operation to identify optimization opportunities.

JWT Generation Performance Analysis

JWT access token generation involves RSA or ECDSA signature operations that have significant computational cost. Profiling JWT generation helps identify whether signing operations are creating throughput limitations.

Operation	Typical Duration	Optimization Opportunities	Security Trade-offs
RSA-2048 Signature Generation	2-5 milliseconds	Use ECDSA-256 for faster signing	Must ensure client compatibility
ECDSA-256 Signature Generation	0.5-1 millisecond	Cache key material in memory	Increases memory usage
JWT Claims Serialization	0.1-0.2 milliseconds	Pre-serialize static claims	Limits dynamic claim generation
Base64url Encoding	0.05-0.1 milliseconds	Use optimized encoding library	Minimal optimization potential
Complete JWT Generation	3-6 milliseconds	Parallel signing operations	Requires careful concurrency management

Database Operation Performance

OAuth2 providers perform frequent database operations for client lookup, token storage, and revocation checking. Performance debugging requires identifying which database operations are causing latency spikes.

1. **Client Lookup Performance:** Profile `ClientStore.GetClient` operations under various load conditions to identify slow queries or missing database indexes.
2. **Authorization Code Storage:** Measure `AuthorizationCode` creation and retrieval performance, particularly focusing on cleanup operations that remove expired codes.
3. **Refresh Token Management:** Profile refresh token rotation operations, which involve creating new tokens, marking old tokens as revoked, and potentially revoking entire token families.
4. **Revocation List Queries:** Measure performance of `RevocationStore.IsRevoked` operations, which occur during every token validation and can become expensive without proper indexing.

Concurrency Issue Identification

OAuth2 implementations must handle concurrent requests safely while maintaining protocol correctness and security properties. Concurrency bugs often manifest as race conditions that corrupt protocol state or create security vulnerabilities.

Authorization Code Race Conditions

Authorization codes must be single-use, but concurrent token exchange requests can create race conditions where the same code is successfully exchanged multiple times before the database can enforce single-use restrictions.

Race Condition Detection Test:

1. Generate authorization code through normal authorization flow
2. Submit 10 concurrent token exchange requests using same authorization code
3. Verify that exactly one request succeeds with valid tokens
4. Verify that all other requests fail with invalid_grant error
5. Verify that authorization code is marked as `Used=true` in database
6. Confirm no orphaned tokens exist from failed concurrent exchanges

Refresh Token Rotation Concurrency

Refresh token rotation logic can create race conditions when the same refresh token is used concurrently, potentially causing token family corruption or allowing multiple successful token renewals.

Concurrency Scenario	Test Implementation	Expected Behavior	Race Condition Indicator
Simultaneous Refresh Requests	Submit same refresh token from multiple threads	One success, others fail	Multiple successful responses
Rotation During Revocation	Rotate token while revoking family	Either rotation or revocation succeeds	Both operations succeed
Family Tracking Update	Multiple tokens in family updated simultaneously	Consistent family state	Inconsistent family relationships
Cleanup During Usage	Expired token cleanup during active usage	Active tokens preserved	Active tokens accidentally cleaned

Client Registration Concurrency

Client registration operations can create race conditions if multiple registration requests attempt to create clients with the same identifiers or if cleanup operations interfere with active registration processes.

1. **Client ID Generation:** Test concurrent client registration to verify that `client_id` generation produces unique values without collisions.
2. **Redirect URI Validation:** Verify that redirect URI validation during registration doesn't conflict with concurrent client lookup operations.
3. **Secret Hash Generation:** Ensure that bcrypt hash generation for client secrets operates correctly under concurrent load without timing vulnerabilities.
4. **Database Transaction Isolation:** Verify that client creation uses appropriate database transaction isolation to prevent partial registration states.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Logging Framework	Standard library log package with JSON formatting	Structured logging with logrus or zap with context tracing
Error Tracking	File-based error logs with log rotation	Distributed error tracking with Sentry or similar service
Performance Monitoring	Basic HTTP middleware with response time logging	APM integration with New Relic, DataDog, or Prometheus metrics
Database Debugging	SQL query logging with execution time measurement	Query analysis tools with slow query detection and optimization
Security Testing	Manual security test cases with curl and custom scripts	Automated security scanning with OWASP ZAP or similar tools
Load Testing	Simple concurrent request testing with Go routines	Professional load testing with k6, Artillery, or JMeter

Recommended File Structure

```
project-root/
  internal/debug/
    logger.go          ← structured logging implementation
    profiler.go        ← performance profiling utilities
    security_tester.go ← security vulnerability testing
    timing_analyzer.go ← timing attack detection tools
    oauth2_validator.go ← protocol compliance verification
  internal/monitoring/
    metrics.go         ← performance metrics collection
    health_checker.go ← system health verification
    abuse_detector.go ← security pattern detection
  cmd/debug/
    main.go            ← debugging CLI tools
    token_inspector.go ← JWT and token analysis utilities
    database_checker.go ← database consistency verification
  test/integration/
    security_test.go   ← comprehensive security testing
    performance_test.go ← load and performance testing
    compliance_test.go ← OAuth2 specification compliance
```

Infrastructure Starter Code

Structured OAuth2 Logger Implementation

```
package debug
```

```
import (
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "log"
    "os"
    "time"
)
```

```
// OAuth2Logger provides structured logging for OAuth2 operations with security considerations
```

```
type OAuth2Logger struct {
    logger *log.Logger
}
```

```
// OAuth2Event represents a loggable OAuth2 protocol event
```

```
type OAuth2Event struct {
    EventType     string           `json:"event_type"`
    Timestamp     time.Time        `json:"timestamp"`
    ClientID      string           `json:"client_id,omitempty"`
    UserID        string           `json:"user_id,omitempty"`
    Fields        map[string]interface{} `json:"fields"`
    SecurityHash  string           `json:"security_hash,omitempty"`
}
```

```
// NewOAuth2Logger creates a new structured logger for OAuth2 operations
```

```
func NewOAuth2Logger() *OAuth2Logger {
    return &OAuth2Logger{
        logger: log.New(os.Stdout, "", 0),
    }
}
```

GO

```

// LogAuthorizationRequest logs an OAuth2 authorization request with security hashing

func (l *OAuth2Logger) LogAuthorizationRequest(clientID, redirectURI, scope, state, codeChallenge string) {

    event := OAuth2Event{

        EventType: "authorization_request",

        Timestamp: time.Now(),

        ClientID: clientID,

        Fields: map[string]interface{}{

            "redirect_uri": redirectURI,

            "scope": scope,

            "state_hash": l.hashSensitive(state),

            "challenge_hash": l.hashSensitive(codeChallenge),

        },
    }

    l.writeEvent(event)
}

// LogTokenExchange logs a token exchange operation with relevant security context

func (l *OAuth2Logger) LogTokenExchange(clientID, grantType string, success bool, errorCode string) {

    event := OAuth2Event{

        EventType: "token_exchange",

        Timestamp: time.Now(),

        ClientID: clientID,

        Fields: map[string]interface{}{

            "grant_type": grantType,

            "success": success,

            "error": errorCode,

        },
    }

    l.writeEvent(event)
}

```

```
}

// LogSecurityEvent logs security-related events like failed authentication or suspicious behavior

func (l *OAuth2Logger) LogSecurityEvent(eventType, clientID, userID, details string) {

    event := OAuth2Event{

        EventType: "security_event",

        Timestamp: time.Now(),

        ClientID: clientID,

        UserID: userID,

        Fields: map[string]interface{}{

            "details": details,

        },
    }

    l.writeEvent(event)
}

// hashSensitive creates SHA256 hash of sensitive values for logging

func (l *OAuth2Logger) hashSensitive(value string) string {

    if value == "" {

        return ""

    }

    hash := sha256.Sum256([]byte(value))

    return hex.EncodeToString(hash[:8]) // Use first 8 bytes for space efficiency
}

// writeEvent serializes and writes an OAuth2Event to the log

func (l *OAuth2Logger) writeEvent(event OAuth2Event) {

    eventJSON, err := json.Marshal(event)

    if err != nil {

        l.logger.Printf("ERROR: Failed to serialize log event: %v", err)

        return
    }
}
```

```
}

l.logger.Printf("%s", string(eventJSON))

}
```

Timing Attack Detection Utility

```
package debug
```

GO

```
import (
```

```
    "crypto/subtle"
```

```
    "fmt"
```

```
    "math"
```

```
    "time"
```

```
)
```

```
// TimingAnalyzer provides tools for detecting timing attack vulnerabilities
```

```
type TimingAnalyzer struct {
```

```
    samples []time.Duration
```

```
}
```

```
// TimingReport contains analysis results for timing attack vulnerability assessment
```

```
type TimingReport struct {
```

```
    CorrectInputAvg    time.Duration `json:"correct_input_avg"`
```

```
    IncorrectInputAvg time.Duration `json:"incorrect_input_avg"`
```

```
    TimingDifference   time.Duration `json:"timing_difference"`
```

```
    VulnerableToAttack bool           `json:"vulnerable_to_attack"`
```

```
}
```

```
// NewTimingAnalyzer creates a new timing analysis utility
```

```
func NewTimingAnalyzer() *TimingAnalyzer {
```

```
    return &TimingAnalyzer{
```

```
        samples: make([]time.Duration, 0, 2000),
```

```
}
```

```
}
```

```
// MeasureTimingConsistency tests cryptographic operations for timing attack vulnerabilities
```

```
func (ta *TimingAnalyzer) MeasureTimingConsistency(operation func(input string) bool, correctInput,
incorrectInput string, iterations int) (*TimingReport, error) {
```

```
    correctTimes := make([]time.Duration, iterations)
```

```

incorrectTimes := make([]time.Duration, iterations)

// Measure correct input timings

for i := 0; i < iterations; i++ {

    start := time.Now()

    operation(correctInput)

    correctTimes[i] = time.Since(start)

}

// Measure incorrect input timings

for i := 0; i < iterations; i++ {

    start := time.Now()

    operation(incorrectInput)

    incorrectTimes[i] = time.Since(start)

}

correctAvg := ta.calculateAverage(correctTimes)

incorrectAvg := ta.calculateAverage(incorrectTimes)

difference := ta.calculateDifference(correctAvg, incorrectAvg)

return &TimingReport{

    CorrectInputAvg:   correctAvg,
    IncorrectInputAvg: incorrectAvg,
    TimingDifference:  difference,
    VulnerableToAttack: difference > 10*time.Microsecond, // 10µs threshold
}, nil
}

// calculateAverage computes mean duration from timing samples

func (ta *TimingAnalyzer) calculateAverage(samples []time.Duration) time.Duration {

    var total time.Duration

    for _, sample := range samples {

```

```

        total += sample

    }

    return total / time.Duration(len(samples))

}

// calculateDifference computes absolute difference between timing averages

func (ta *TimingAnalyzer) calculateDifference(time1, time2 time.Duration) time.Duration {

    if time1 > time2 {

        return time1 - time2

    }

    return time2 - time1

}

// TestConstantTimeComparison verifies that comparison operations use constant-time algorithms

func (ta *TimingAnalyzer) TestConstantTimeComparison(correctValue, incorrectValue string)
(*TimingReport, error) {

    constantTimeCompare := func(input string) bool {

        return subtle.ConstantTimeCompare([]byte(correctValue), []byte(input)) == 1

    }

    return ta.MeasureTimingConsistency(constantTimeCompare, correctValue, incorrectValue, 1000)

}

```

Core Logic Skeleton Code

Security Vulnerability Testing Framework

```
package debug
```

// AttackResult represents the outcome of a security attack simulation

```
type AttackResult struct {
```

 AttackType string `json:"attack_type"`

 Successful bool `json:"successful"`

 Details string `json:"details"`

 Mitigation string `json:"mitigation"`

}

// SecurityTester provides comprehensive security testing for OAuth2 implementations

```
type SecurityTester struct {
```

 providerURL string

 httpClient *http.Client

}

// NewSecurityTester creates a new OAuth2 security testing utility

```
func NewSecurityTester(providerURL string) *SecurityTester {
```

 return &SecurityTester{

 providerURL: providerURL,

 httpClient: &http.Client{Timeout: 30 * time.Second},

}

```
}
```

// SimulateTokenReplayAttack tests authorization code and refresh token replay protection

```
func (st *SecurityTester) SimulateTokenReplayAttack(tokenExchangeFunc func(code string) error,
validCode string) *AttackResult {
```

 // TODO 1: Execute first token exchange with valid authorization code

 // TODO 2: Capture the response to verify initial exchange succeeds

 // TODO 3: Attempt second token exchange using the same authorization code

 // TODO 4: Verify that second exchange fails with invalid_grant error

 // TODO 5: Attempt third exchange to confirm code is permanently invalidated

GO

```

// TODO 6: Return AttackResult indicating whether replay protection worked

// Hint: Successful replay attack means the same code worked multiple times

panic("TODO: Implement token replay attack simulation")

}

// TestAuthorizationCodeBinding verifies that authorization codes are properly bound to clients and
// redirect URIs

func (st *SecurityTester) TestAuthorizationCodeBinding(clientID, redirectURI, authCode string) *AttackResult {
    // TODO 1: Attempt to exchange authorization code using different client_id
    // TODO 2: Attempt to exchange authorization code using different redirect_uri
    // TODO 3: Test PKCE binding by using incorrect code_verifier
    // TODO 4: Verify all cross-client code exchange attempts fail
    // TODO 5: Confirm proper error responses (invalid_grant vs invalid_client)
    // TODO 6: Return result indicating whether code binding is enforced

    panic("TODO: Implement authorization code binding tests")
}

// TestRefreshTokenFamilyTracking verifies refresh token family security and theft detection

func (st *SecurityTester) TestRefreshTokenFamilyTracking(refreshToken string) *AttackResult {
    // TODO 1: Use refresh token to get new access token and rotated refresh token
    // TODO 2: Attempt to reuse the original refresh token (should trigger family revocation)
    // TODO 3: Verify that the new refresh token is also invalidated (family revocation)
    // TODO 4: Confirm that any subsequent token usage fails with invalid_grant
    // TODO 5: Test concurrent usage of same refresh token for race conditions
    // TODO 6: Return result indicating whether family tracking prevents token reuse

    panic("TODO: Implement refresh token family tracking tests")
}

```

Performance Profiling Infrastructure

```
package debug
```

GO

```
// PerformanceProfiler provides comprehensive performance analysis for OAuth2 operations

type PerformanceProfiler struct {

    samples map[string][]time.Duration

    mu      sync.RWMutex
}

// NewPerformanceProfiler creates a new OAuth2 performance profiling utility

func NewPerformanceProfiler() *PerformanceProfiler {

    return &PerformanceProfiler{
        samples: make(map[string][]time.Duration),
    }
}

// ProfileTokenGeneration measures JWT token generation performance across different scenarios

func (pp *PerformanceProfiler) ProfileTokenGeneration(generator JWTGenerator, scenarios []TokenGenerationScenario) error {

    // TODO 1: Warm up the JWT generator with initial token generation calls

    // TODO 2: For each scenario, generate multiple tokens and measure timing

    // TODO 3: Calculate statistics (mean, median, 95th percentile) for each scenario

    // TODO 4: Store timing samples for later analysis and comparison

    // TODO 5: Identify scenarios that exceed acceptable performance thresholds

    // TODO 6: Generate performance report with optimization recommendations

    // Hint: Different key sizes (RSA-2048 vs ECDSA-256) will show significant timing differences

    panic("TODO: Implement JWT generation performance profiling")
}

// ProfileDatabaseOperations measures database query performance for OAuth2 data operations

func (pp *PerformanceProfiler) ProfileDatabaseOperations(clientStore ClientStore, tokenStore RefreshTokenStore) error {

    // TODO 1: Profile client lookup operations with various client_id patterns

    // TODO 2: Measure authorization code creation and retrieval performance
```

```

    // TODO 3: Profile refresh token rotation operations including family management

    // TODO 4: Test revocation list query performance with different list sizes

    // TODO 5: Measure cleanup operations for expired tokens and authorization codes

    // TODO 6: Identify slow queries and suggest database optimization strategies

    panic("TODO: Implement database operation performance profiling")

}

```

Milestone Checkpoints

Milestone 1 Debugging Checkpoint After implementing client registration and authorization endpoint:

- Run: `go test ./internal/debug/... -v` to execute security and timing tests
- Expected: All timing attack tests pass, authorization code binding tests pass
- Manual verification: Submit authorization request with missing PKCE, should fail
- Warning signs: Authorization codes accepted multiple times, timing differences > 10µs

Milestone 2 Debugging Checkpoint After implementing token endpoint and JWT generation:

- Run: `./cmd/debug/token_inspector <jwt_token>` to analyze generated tokens
- Expected: JWT contains proper claims, signature validates, no timing vulnerabilities
- Manual verification: Use same authorization code twice, second should fail
- Warning signs: JWT signature verification inconsistent, refresh token reuse allowed

Milestone 3 Debugging Checkpoint After implementing introspection and revocation:

- Run: Revoke token, then attempt introspection - should show active=false
- Expected: Revoked tokens consistently rejected, family revocation works
- Manual verification: Revoke refresh token family, verify all family tokens invalidated
- Warning signs: Revoked tokens still validate as active, family tracking broken

Milestone 4 Debugging Checkpoint After implementing UserInfo and consent management:

- Run: Request UserInfo with limited scopes, verify claim filtering works
- Expected: Only authorized claims returned, consent persistence prevents re-prompting
- Manual verification: Revoke consent, next authorization should re-prompt user
- Warning signs: Unauthorized claims leaked, consent decisions not persisted

Future Extensions

Milestone(s): This section provides a roadmap for extending the OAuth2/OIDC provider beyond the core implementation covered in milestones 1-4, establishing a foundation for production-ready features and enterprise integration capabilities.

Mental Model: Software Evolution Ecosystem

Think of these future extensions like expanding a successful local business into a multi-location enterprise. Your OAuth2 provider starts as a focused identity service solving the core authentication and authorization problem, much like a local bank serving basic checking and savings accounts. But as your organization grows and integrates with partners, you need additional services: international transfers (federation), business credit lines (advanced security features), automated teller networks (additional grant types), and corporate reporting (operational features).

Each extension builds upon the solid foundation established in the core implementation, adding specialized capabilities without disrupting the essential services that existing clients depend upon. The key insight is that these extensions follow the same architectural patterns and security principles as the core system—they're not bolted-on afterthoughts, but natural evolutions that leverage the existing infrastructure.

Just as a bank's expansion into investment services still relies on the same customer identification and account management systems, OAuth2 extensions reuse the same client registration, token generation, and validation mechanisms while adding new endpoints, flows, and integration points. The mental model helps us understand that each extension should feel like a natural part of the ecosystem, not a foreign addition.

Additional Grant Types

Beyond the authorization code flow and client credentials grant implemented in the core milestones, the OAuth2 ecosystem includes several specialized grant types designed for specific use cases and device capabilities. Each grant type represents a different approach to the fundamental OAuth2 challenge: how do we securely establish trust between a client application and the authorization server when the capabilities and constraints of the client environment vary significantly?

The **device authorization grant** (RFC 8628) addresses the challenge of OAuth2 authentication on input-constrained devices like smart TVs, gaming consoles, or IoT devices that lack a web browser or have limited text input capabilities. Think of this like trying to log into Netflix on your TV—you don't want to type your password with a remote control, so instead the TV displays a code that you enter on your phone or computer where typing is easier.

The device flow introduces a polling-based mechanism where the device requests a device code and user code from a device authorization endpoint, displays the user code to the user, and then polls the token endpoint while the user completes authentication on a secondary device. This requires new endpoints and state management beyond the core implementation.

Device Flow Data Structures:

Structure	Field	Type	Description
<code>DeviceAuthorizationRequest</code>	ClientID	string	Identifying the device application
	Scope	string	Requested permissions for device access
<code>DeviceAuthorizationResponse</code>	DeviceCode	string	Server-generated code for device polling
	UserCode	string	Human-readable code for user entry
	VerificationURI	string	URL where user enters the user code
	VerificationURIComplete	string	Optional URL with user code pre-filled
	ExpiresIn	int	Lifetime of the device code in seconds
	Interval	int	Minimum polling interval in seconds
<code>DeviceTokenRequest</code>	GrantType	string	Must be "urn:ietf:params:oauth:grant-type:device_code"
	DeviceCode	string	The device code from authorization response
	ClientID	string	Device client identifier

The **implicit grant** was historically used for single-page applications but is now deprecated in favor of authorization code with PKCE due to security concerns. However, understanding its mechanics helps illustrate why PKCE was necessary. The implicit flow returns access tokens directly from the authorization endpoint in the URL fragment, bypassing the token endpoint entirely. This eliminates the need for client authentication but creates security vulnerabilities around token exposure in browser history and referer headers.

Security Insight: The deprecation of the implicit grant demonstrates how OAuth2 security best practices evolve. What seemed like a reasonable trade-off for browser limitations in 2012 became a security liability as we better understood browser security models and as CORS enabled cross-origin requests from JavaScript.

Custom grant types allow organizations to implement domain-specific authentication flows while maintaining OAuth2 token compatibility. Examples include integrating with legacy authentication systems, implementing step-up authentication for sensitive operations, or creating specialized flows for partner integrations. Custom grants follow the same token endpoint pattern but with organization-specific grant type identifiers and validation logic.

Custom Grant Implementation Patterns:

Pattern	Use Case	Validation Requirements	Token Response
Legacy Integration	Bridge to SAML/LDAP systems	Validate external assertion	Standard OAuth2 tokens
Step-up Authentication	Additional security for sensitive ops	Verify primary token + secondary auth	Enhanced scope tokens
Partner Exchange	B2B integration flows	Validate partner credentials + user consent	Audience-specific tokens
Certificate-based	PKI client authentication	Validate client certificate chain	Client-bound tokens

Decision: Device Flow State Management

- Context:** Device flows require managing long-lived authorization requests while users complete authentication on secondary devices, creating new state tracking requirements
- Options Considered:** In-memory maps, database tables with TTL, Redis with expiration
- Decision:** Database table with TTL and background cleanup, similar to authorization code storage
- Rationale:** Reuses existing database patterns, survives server restarts, enables horizontal scaling with shared database
- Consequences:** Adds one new table but leverages existing cleanup mechanisms; polling load requires rate limiting

The device authorization endpoint introduces new security considerations around code generation and polling behavior. Device codes must be cryptographically random and sufficiently long to prevent brute force attacks, while user codes should be short and human-readable but still unpredictable. The polling mechanism requires rate limiting to prevent abuse while ensuring responsive user experience.

Device Flow Security Requirements:

Component	Security Requirement	Implementation Approach	Threat Mitigated
Device Code	Cryptographically random, 128+ bits	Same generator as authorization codes	Code prediction attacks
User Code	Short but unpredictable, 6-8 characters	Base32 encoding of random bytes	Code guessing attacks
Polling Rate	Respect interval parameter	Rate limiting by client_id + device_code	Resource exhaustion attacks
Code Expiration	Short-lived, 10-15 minutes	TTL in storage, background cleanup	Credential exposure

Identity Federation

Identity federation extends the OAuth2 provider to trust and integrate with external identity sources, enabling single sign-on across organizational boundaries and reducing user friction through existing identity relationships. Think of federation

like a hotel accepting multiple forms of ID—whether you present a driver's license, passport, or military ID, the hotel can verify your identity through different trusted authorities while providing the same room access.

SAML integration allows the OAuth2 provider to accept SAML assertions from enterprise identity providers as authentication evidence, bridging the gap between SAML-based enterprise SSO systems and OAuth2-based cloud applications. This typically involves implementing a SAML assertion grant type where clients can exchange valid SAML assertions for OAuth2 tokens.

SAML Integration Components:

Component	Responsibility	Key Operations	Security Requirements
SAMLAssertionValidator	Validate incoming SAML assertions	Signature verification, condition checking	Trust store management, XML signature validation
SAMLGrantHandler	Process SAML assertion grant requests	Assertion validation, user mapping	Audience restriction, time window validation
IdentityMapper	Map SAML attributes to local user profiles	Attribute extraction, profile creation/update	Attribute validation, conflict resolution
TrustManager	Manage trusted SAML identity providers	Certificate validation, metadata processing	Certificate chain validation, revocation checking

Social login integration enables users to authenticate using existing accounts from providers like Google, GitHub, or Microsoft, reducing registration friction and leveraging established identity relationships. This involves implementing OAuth2 client capabilities within your authorization server, where your provider acts as a client to upstream identity providers during user authentication.

Social Login Flow Components:

Phase	Component Action	External Provider Interaction	User Experience
Discovery	Present provider options	Fetch provider metadata	Choose authentication method
Upstream Authentication	Redirect to social provider	Standard OAuth2/OIDC flow	Authenticate with existing account
Token Exchange	Receive upstream tokens	Validate tokens, fetch profile	Transparent to user
Local Account Creation	Create or link local profile	Store identity linkage	Optional profile completion

The social login flow creates identity linkage challenges where users may have multiple upstream identities that should map to a single local account, or where users may want to unlink social accounts while maintaining local account access. This requires careful user experience design and robust account management capabilities.

Decision: Identity Linkage Strategy

- **Context:** Users may authenticate via multiple social providers or want to link social accounts to existing local accounts
- **Options Considered:** Automatic email-based linking, manual account linking, separate accounts per provider
- **Decision:** Manual linking with email verification for automatic suggestions
- **Rationale:** Prevents account takeover via email compromise while providing user convenience; gives users control over identity consolidation
- **Consequences:** Requires additional UI for account management but provides strongest security properties

Enterprise SSO integration extends beyond simple SAML support to include full enterprise identity workflows like just-in-time provisioning, group-based access control, and integration with enterprise directories. This often involves implementing SCIM (System for Cross-domain Identity Management) endpoints for automated user and group synchronization.

Enterprise SSO Extended Features:

Feature	Purpose	Implementation Components	Business Value
JIT Provisioning	Automatic user creation on first login	User creation pipeline, default role assignment	Reduces admin overhead
Group Synchronization	Mirror enterprise groups in OAuth provider	SCIM endpoint, group mapping rules	Enables group-based authorization
Directory Integration	Real-time user status updates	LDAP connector, user deactivation workflows	Maintains security during role changes
Audit Integration	Enterprise audit trail requirements	Audit event formatting, SIEM integration	Compliance and security monitoring

Federation introduces complex trust management requirements where the OAuth2 provider must maintain and validate relationships with multiple external identity sources. Each federated identity provider requires its own trust configuration, including certificate management, endpoint validation, and metadata synchronization.

Advanced Security Features

Modern OAuth2 deployments face sophisticated attack vectors that require security enhancements beyond the core specification. These advanced security features implement cutting-edge OAuth2 security research and industry best practices to protect against evolving threats while maintaining protocol compatibility and user experience.

Pushed Authorization Requests (PAR, RFC 9126) addresses the security limitations of passing authorization parameters via browser redirects by requiring clients to push authorization request parameters directly to the authorization server before initiating the front-channel authorization flow. Think of PAR like submitting a loan application at the bank before meeting with a loan officer—sensitive details are communicated securely through a direct channel rather than discussed in the public lobby.

PAR Implementation Architecture:

Component	Responsibility	Security Benefit	Implementation Requirements
PAREndpoint	Accept and validate pushed authorization requests	Eliminates parameter tampering in authorization URL	Client authentication, parameter validation
RequestObjectStore	Store authorization parameters with unique identifier	Prevents replay attacks through one-time use	Secure storage, TTL management
AuthorizationEndpoint (enhanced)	Accept PAR request URLs instead of full parameters	Reduces authorization URL size and complexity	Request URI validation, parameter retrieval
ParameterValidator	Validate authorization parameters server-side	Prevents client-side parameter manipulation	Schema validation, business rule enforcement

PAR transforms the authorization flow by introducing a preliminary step where clients authenticate and submit their authorization request parameters via a secure back-channel POST request. The authorization server validates the parameters, stores them with a unique request URI, and returns the request URI to the client. The client then initiates the normal authorization flow using only the request URI instead of embedding all parameters in the authorization URL.

PAR Security Benefits:

Attack Vector	Traditional Authorization	PAR Protection	Additional Security
Parameter Tampering	Parameters visible in URL, browser history	Parameters hidden in server-side storage	Request integrity verification
Authorization URL Length	Limited by browser URL length constraints	Unlimited parameter complexity support	Complex policy expression
Phishing Attacks	Malicious sites can construct convincing auth URLs	Request URIs meaningless without server context	Server-side validation enforcement
Traffic Analysis	Authorization parameters visible to network observers	Only opaque request URI visible	Parameter confidentiality

Rich Authorization Requests (RAR, RFC 9396) extends OAuth2 scopes with structured authorization request details that enable fine-grained, context-aware authorization decisions. Instead of simple scope strings like "payments:read", RAR allows clients to request specific permissions with detailed context like "access account 12345 for payments between \$1-\$1000 during business hours".

RAR Data Structures:

Structure	Field	Type	Description
AuthorizationDetail	Type	string	The authorization detail type identifier
	Locations	[]string	Resource server locations for this authorization
	Actions	[]string	Specific actions permitted on the resource
	DataTypes	[]string	Types of data that can be accessed
	Privileges	map[string]interface{}	Type-specific privilege details
RichAuthorizationRequest	AuthorizationDetails	[]AuthorizationDetail	List of detailed authorization requests
	Scope	string	Traditional scopes for backward compatibility
AuthorizationDetailResponse	AuthorizationDetails	[]AuthorizationDetail	Granted authorization details (may differ from requested)

RAR enables sophisticated authorization scenarios like time-bounded access, amount-limited transactions, or resource-specific permissions while maintaining backward compatibility with scope-based authorization. The authorization server can modify requested authorization details during the consent process, granting subset permissions or adding additional constraints based on policy rules.

Decision: RAR Processing Strategy

- **Context:** Rich authorization requests require complex validation, policy evaluation, and consent presentation beyond simple scope checking
- **Options Considered:** Inline processing in authorization endpoint, separate policy engine, rule-based configuration
- **Decision:** Pluggable policy engine with rule-based configuration and extensible evaluation context
- **Rationale:** Enables complex authorization logic without hardcoding business rules; supports compliance requirements and custom policies
- **Consequences:** Adds complexity to authorization endpoint but provides necessary flexibility for enterprise requirements

FAPI (Financial-grade API) compliance implements the security requirements defined by the OpenID Foundation for financial services, including strong authentication, request object requirements, and advanced threat protection. FAPI represents the highest security standard for OAuth2 deployments and serves as a model for other high-security industries.

FAPI Security Requirements:

FAPI Level	Authentication Requirement	Authorization Requirement	Token Requirement	Network Requirement
FAPI 1.0 Baseline	Multi-factor authentication	Signed request objects	Short-lived access tokens	TLS 1.2 with perfect forward secrecy
FAPI 1.0 Advanced	Certificate-bound tokens	Encrypted request objects	MTLS or token binding	TLS 1.2 with specific cipher suites
FAPI 2.0	Strong authentication	PAR + signed request objects	DPoP or MTLS	TLS 1.3 with approved cipher suites

FAPI implementation requires extensive security enhancements including request object validation, certificate-based client authentication, token binding mechanisms, and advanced threat detection. These requirements often necessitate significant architectural changes to support certificate management, cryptographic operations, and real-time security monitoring.

Certificate-bound tokens (mutual TLS) create a binding between OAuth2 tokens and client certificates, ensuring that tokens can only be used by clients possessing the corresponding private key. This prevents token theft attacks even if access tokens are intercepted, as attackers cannot present the required client certificate.

Operational Features

Production OAuth2 deployments require comprehensive operational capabilities for monitoring, management, and maintenance that extend beyond the core protocol functionality. These operational features transform a functional OAuth2 implementation into a production-ready service capable of supporting enterprise-scale deployments with appropriate observability, security monitoring, and administrative capabilities.

Metrics and monitoring provide visibility into OAuth2 provider performance, usage patterns, and security events through comprehensive instrumentation and alerting. Effective monitoring enables proactive identification of performance bottlenecks, security anomalies, and operational issues before they impact users or security posture.

OAuth2 Metrics Categories:

Category	Key Metrics	Collection Method	Alert Conditions
Performance	Request latency, token generation time, database query duration	Histogram metrics with percentiles	P99 latency > 500ms, error rate > 1%
Usage	Requests per second, active tokens, client activity	Counter and gauge metrics	Unusual traffic spikes, dormant client patterns
Security	Failed authentication attempts, token validation errors, suspicious patterns	Counter metrics with labels	Brute force patterns, token enumeration attacks
Business	New client registrations, user consent decisions, scope usage	Counter and gauge metrics	Unusual consent patterns, scope escalation attempts

OAuth2-specific monitoring requires tracking token lifecycles, authorization flow completion rates, and client behavior patterns. Unlike traditional web application monitoring, OAuth2 monitoring must account for multi-step flows where failures at any stage can indicate security issues or integration problems.

Token Lifecycle Monitoring:

Lifecycle Stage	Metrics	Security Indicators	Performance Indicators
Authorization Request	Request rate, validation failures	Parameter tampering attempts	Consent screen load times
Code Exchange	Exchange success rate, code expiration	Code replay attempts	Token generation latency
Token Usage	Validation requests, scope checks	Invalid token presentations	Validation cache hit rates
Token Expiration	Natural expiration vs revocation	Premature revocation patterns	Cleanup operation efficiency

Administrative interfaces enable OAuth2 provider operators to manage clients, monitor system health, and respond to security incidents through web-based dashboards and API endpoints. Administrative capabilities must balance operational efficiency with security controls, ensuring that administrative access follows the same security principles as the OAuth2 flows themselves.

Administrative Interface Components:

Component	Functionality	Security Requirements	User Experience Requirements
ClientManagementUI	Create, modify, delete OAuth2 clients	Role-based access control, audit logging	Bulk operations, search and filtering
UserManagementUI	View user profiles, consent history	User privacy controls, data minimization	Efficient user lookup, consent visualization
SecurityDashboard	Monitor threats, investigate incidents	Real-time alerting, correlation analysis	Interactive threat timeline, drill-down capabilities
SystemHealthUI	View metrics, configure alerts	System administration roles	Real-time dashboards, historical trending

Administrative interfaces require careful authorization design to prevent privilege escalation while enabling efficient operations. Administrative users need different permission levels—from read-only monitoring access to full client management capabilities—and these permissions must integrate with the organization's existing identity and access management systems.

Decision: Administrative Authentication Strategy

- **Context:** Administrative interfaces need strong authentication while integrating with existing enterprise identity systems
- **Options Considered:** Separate admin accounts, federated SSO integration, OAuth2 self-authentication
- **Decision:** Federated SSO with OAuth2 self-authentication as fallback, plus mandatory MFA for admin operations
- **Rationale:** Leverages existing enterprise authentication while providing OAuth2 provider independence; MFA requirement addresses high-privilege access risks
- **Consequences:** Requires federation implementation but provides strongest security and operational integration

Audit and compliance capabilities ensure that OAuth2 provider operations meet regulatory requirements and security standards through comprehensive logging, reporting, and data retention policies. Audit systems must capture all security-relevant events while protecting user privacy and enabling compliance demonstration.

Audit Event Categories:

Event Type	Required Information	Retention Period	Compliance Standards
Authentication Events	User ID, client ID, timestamp, outcome, IP address	7 years	SOX, GDPR, PCI DSS
Authorization Events	User consent decisions, scope grants, expiration	7 years	Financial regulations, healthcare compliance
Administrative Actions	Admin user, action type, affected resources	10 years	Security frameworks, audit standards
Security Events	Threat detection, incident response, remediation	7 years	Incident reporting requirements

Audit logging must balance comprehensive event capture with performance and storage costs. High-volume events like token validation require sampling or aggregation strategies, while security events require complete capture with tamper-evident storage.

Backup and disaster recovery ensure OAuth2 provider availability and data integrity through systematic backup procedures, recovery testing, and disaster response planning. OAuth2 providers contain critical security state that must be protected against data loss while enabling rapid service restoration.

Backup and Recovery Components:

Component	Backup Frequency	Recovery Priority	Recovery Time Objective
Client Registration Database	Continuous replication	Critical	< 1 hour
User Profile Database	Daily backup with continuous replication	High	< 4 hours
Token Revocation Lists	Real-time replication	Critical	< 15 minutes
Cryptographic Keys	Secure offline backup	Critical	< 30 minutes
Configuration and Secrets	Version-controlled backup	High	< 2 hours

Disaster recovery testing must validate not only data restoration but also security property preservation. Recovered OAuth2 providers must maintain the same trust relationships, cryptographic properties, and security configurations as the original deployment to prevent security gaps during recovery operations.

Implementation Guidance

The implementation of future extensions requires careful architectural planning to ensure that new features integrate seamlessly with the core OAuth2 provider while maintaining security properties and operational stability. Each extension should be designed as a modular component that can be enabled or disabled based on deployment requirements.

Technology Recommendations:

Extension Category	Simple Implementation	Advanced Implementation
Device Flow	In-memory polling state with Redis backup	Distributed state with Kafka event streams
SAML Integration	XML parsing with crypto/xml libraries	Dedicated SAML library (go-saml) with metadata management
Social Login	HTTP clients with provider-specific logic	OAuth2 client library with provider discovery
PAR/RAR	Database storage with JSON parameter handling	Policy engine integration with rule-based evaluation
Monitoring	Prometheus metrics with Grafana dashboards	Full observability stack with distributed tracing
Administration	HTTP handlers with HTML templates	React/Vue SPA with REST API backend

Recommended File Structure:

```

oauth2-provider/
  cmd/
    server/main.go           ← main server entry point
    admin/main.go            ← administrative interface server
    migrate/main.go          ← database migration tool

  internal/
    core/
      auth/                 ← core OAuth2 components from milestones 1-4
      token/                ← token endpoint
      client/               ← client management
      userinfo/             ← OIDC UserInfo

  extensions/
    device/
      endpoint.go           ← device authorization grant
      poller.go              ← device authorization endpoint
      storage.go             ← device polling logic
      storage.go             ← device code storage

    federation/
      saml/
        assertion.go         ← SAML integration
        grant.go              ← SAML assertion validation
        trust.go              ← SAML assertion grant handler
        trust.go              ← trust manager
      social/
        providers/
          linking.go           ← social login integration
          linking.go           ← provider-specific implementations
          linking.go           ← account linking logic

    security/
      par/
        endpoint.go           ← pushed authorization requests
        storage.go             ← PAR endpoint
        storage.go             ← request object storage
      rar/
        parser.go              ← rich authorization requests
        policy.go              ← authorization detail parsing
        policy.go              ← policy engine
      fapi/
        validation.go          ← FAPI compliance
        validation.go          ← FAPI security validation
        mtls.go                ← mutual TLS handling

  operations/
    metrics/
      oauth2.go              ← metrics collection
      instrumentation.go     ← OAuth2-specific metrics
      instrumentation.go     ← instrumentation helpers
    admin/
      handlers.go             ← administrative interfaces
      ui/
        ui.go                  ← admin API handlers
        ui.go                  ← web interface files
      audit/
        logger.go              ← audit and compliance
        retention.go            ← structured audit logging
        retention.go            ← audit data retention

  shared/
    config/                ← configuration management
    storage/               ← database interfaces
    security/              ← shared security utilities

```

Device Flow Infrastructure Starter Code:

```
package device

import (
    "context"
    "crypto/rand"
    "encoding/base32"
    "fmt"
    "strings"
    "time"
)

// DeviceAuthorizationRequest represents a device authorization request

type DeviceAuthorizationRequest struct {

    ClientID string `json:"client_id"`
    Scope     string `json:"scope"`
}

// DeviceAuthorizationResponse represents the response to device authorization

type DeviceAuthorizationResponse struct {

    DeviceCode          string `json:"device_code"`
    UserCode            string `json:"user_code"`
    VerificationURI    string `json:"verification_uri"`
    VerificationURIComplete string `json:"verification_uri_complete"`
    ExpiresIn           int    `json:"expires_in"`
    Interval             int    `json:"interval"`
}

// DeviceCodeRecord represents stored device authorization state

type DeviceCodeRecord struct {

    DeviceCode    string `db:"device_code"`
    UserCode      string `db:"user_code"`
    ClientID     string `db:"client_id"`
}
```

GO

```
Scope      string `db:"scope"`

ExpiresAt  time.Time `db:"expires_at"`

UserApproved *bool `db:"user_approved"`

UserID     *string `db:"user_id"`

CreatedAt  time.Time `db:"created_at"`

}

// DeviceCodeGenerator generates secure device and user codes

type DeviceCodeGenerator struct {

    baseURL string

}

// GenerateDeviceCode creates a cryptographically secure device code

func (g *DeviceCodeGenerator) GenerateDeviceCode() (string, error) {

    // TODO: Generate 32 bytes of random data

    // TODO: Encode as base64url for URL safety

    // TODO: Return the encoded string

    // Hint: Use crypto/rand.Read() and base64.URLEncoding

}

// GenerateUserCode creates a human-readable user code

func (g *DeviceCodeGenerator) GenerateUserCode() (string, error) {

    // TODO: Generate 5 bytes of random data

    // TODO: Encode as base32 for human readability

    // TODO: Remove padding and convert to uppercase

    // TODO: Insert hyphen for readability (e.g., "A1B2-C3D4")

    // Hint: Use base32.StdEncoding and string manipulation

}

// DeviceStore interface for device code persistence

type DeviceStore interface {
```

```
CreateDeviceCode(ctx context.Context, record *DeviceCodeRecord) error  
  
GetDeviceCodeByDeviceCode(ctx context.Context, deviceCode string) (*DeviceCodeRecord, error)  
  
GetDeviceCodeByUserCode(ctx context.Context, userCode string) (*DeviceCodeRecord, error)  
  
ApproveDeviceCode(ctx context.Context, userCode, userID string) error  
  
DenyDeviceCode(ctx context.Context, userCode string) error  
  
CleanupExpiredDeviceCodes(ctx context.Context) (int, error)  
  
}
```

PAR Core Logic Skeleton:

GO

```
package par

import (
    "context"
    "crypto/sha256"
    "encoding/base64"
    "net/http"
    "time"
)

// PAREndpoint handles pushed authorization requests

type PAREndpoint struct {

    store      RequestObjectStore
    validator  *ParameterValidator
    clientAuth ClientAuthenticator
    requestURITTL time.Duration
}

// HandlePushedAuthorizationRequest processes PAR requests

func (p *PAREndpoint) HandlePushedAuthorizationRequest(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Extract and validate client credentials from request
    // TODO 2: Parse authorization parameters from request body
    // TODO 3: Validate authorization parameters according to OAuth2 spec
    // TODO 4: Generate unique request URI with sufficient entropy
    // TODO 5: Store authorization parameters with request URI and TTL
    // TODO 6: Return request URI and expires_in to client
    // Hint: Use client authentication from token endpoint patterns
    // Hint: Request URI should be "urn:ietf:params:oauth:request_uri:" + random
}

// GenerateRequestURI creates a unique request URI for stored parameters

func (p *PAREndpoint) GenerateRequestURI(params map[string]string) (string, error) {
```

```

// TODO 1: Generate 32 bytes of cryptographically secure random data

// TODO 2: Create SHA256 hash of authorization parameters for integrity

// TODO 3: Combine random data and hash for request URI identifier

// TODO 4: Encode as base64url and prefix with standard URN

// TODO 5: Return formatted request URI

// Hint: Use crypto/rand and crypto/sha256 packages

}

// ValidateRequestURI verifies request URI format and retrieves parameters

func (p *PAREndpoint) ValidateRequestURI(requestURI string) (map[string]string, error) {

    // TODO 1: Verify request URI has correct URN prefix

    // TODO 2: Extract request identifier from URI

    // TODO 3: Retrieve stored parameters from storage

    // TODO 4: Verify parameters haven't expired

    // TODO 5: Mark request URI as used (one-time use)

    // Hint: Return error if URI is malformed, expired, or already used

}

// RequestObjectStore interface for PAR parameter storage

type RequestObjectStore interface {

    StoreRequestObject(ctx context.Context, requestURI string, params map[string]string, expiresAt time.Time) error

    RetrieveRequestObject(ctx context.Context, requestURI string) (map[string]string, error)

    MarkRequestObjectUsed(ctx context.Context, requestURI string) error

    CleanupExpiredRequestObjects(ctx context.Context) (int, error)

}

```

Monitoring Infrastructure Starter Code:

GO

```
package metrics

import (
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
    "time"
)

var (
    // OAuth2 request metrics

    oauth2Requests = promauto.NewCounterVec(
        prometheus.CounterOpts{
            Name: "oauth2_requests_total",
            Help: "Total number of OAuth2 requests by endpoint and status",
        },
        []string{"endpoint", "client_id", "status"},
    )

    // OAuth2 request duration metrics

    oauth2Duration = promauto.NewHistogramVec(
        prometheus.HistogramOpts{
            Name:      "oauth2_request_duration_seconds",
            Help:      "OAuth2 request duration in seconds",
            Buckets:  prometheus.DefBuckets,
        },
        []string{"endpoint", "client_id"},
    )

    // Active token metrics

    activeTokens = promauto.NewGaugeVec(
        prometheus.GaugeOpts{
            Name: "oauth2_active_tokens",
        },
    )
)
```

```
        Help: "Number of active OAuth2 tokens by type",
    },
    []string{"token_type", "client_id"},

)

// Security event metrics

securityEvents = promauto.NewCounterVec(
    prometheus.CounterOpts{
        Name: "oauth2_security_events_total",
        Help: "Total security events by type and client",
    },
    []string{"event_type", "client_id", "severity"},

)
)

// OAuth2Instrumentor provides OAuth2-specific metrics collection

type OAuth2Instrumentor struct {
    startTime time.Time
}

// NewOAuth2Instrumentor creates new metrics collector

func NewOAuth2Instrumentor() *OAuth2Instrumentor {
    return &OAuth2Instrumentor{
        startTime: time.Now(),
    }
}

// RecordRequest increments request counter and measures duration

func (i *OAuth2Instrumentor) RecordRequest(endpoint, clientID, status string, duration time.Duration) {
    // TODO: Increment oauth2Requests counter with labels

    // TODO: Record request duration in oauth2Duration histogram
}
```

```

    // Hint: Use .With(prometheus.Labels{}) to set label values
}

// RecordTokenCreated updates active token gauge

func (i *OAuth2Instrumentor) RecordTokenCreated(tokenType, clientID string) {
    // TODO: Increment activeTokens gauge for token creation

    // Hint: Use .With() to specify labels, then .Inc() to increment
}

// RecordSecurityEvent logs security-related events

func (i *OAuth2Instrumentor) RecordSecurityEvent(eventType, clientID, severity string) {
    // TODO: Increment securityEvents counter with event details

    // TODO: Consider rate limiting high-frequency security events

    // Hint: Use severity levels: "low", "medium", "high", "critical"
}

```

Milestone Checkpoints:

After implementing device flow extensions:

- Run `curl -X POST http://localhost:8080/device_authorization -d 'client_id=test&scope=openid'`
- Verify response contains `device_code`, `user_code`, and `verification_uri`
- Navigate to `verification_uri` and enter `user_code`
- Poll token endpoint with `device_code` to receive access token
- Confirm device codes expire after configured TTL

After implementing federation features:

- Configure test SAML IdP or social login provider credentials
- Initiate OAuth2 flow with identity provider parameter
- Verify redirect to external authentication
- Confirm successful token issuance after external authentication
- Test account linking and unlinking functionality

After implementing operational features:

- Access metrics endpoint at `/metrics` and verify OAuth2-specific metrics
- Generate load with multiple clients and verify metric accuracy
- Access admin interface and verify client management capabilities
- Review audit logs for comprehensive event capture

- Test backup and recovery procedures with test data

These extensions transform the core OAuth2 provider into a production-ready identity service capable of supporting diverse client types, enterprise integration requirements, and operational demands while maintaining the security properties and architectural principles established in the foundational milestones.

Glossary

Milestone(s): This glossary provides definitions for all milestones (1-4), establishing clear terminology used throughout the OAuth2/OIDC provider implementation

Mental Model: Technical Dictionary for Authorization Systems

Think of this glossary as a specialized dictionary for authorization systems, similar to how medical professionals use precise terminology to avoid confusion in high-stakes environments. Just as a doctor distinguishes between "acute" and "chronic" conditions because the treatment implications are vastly different, OAuth2 and OpenID Connect use precise terminology where subtle distinctions have significant security implications. For example, an "authorization code" versus an "access token" might sound similar to a newcomer, but they serve completely different purposes in the security model - one is a temporary voucher for obtaining credentials, while the other is the actual credential for accessing resources.

This glossary serves as your reference guide for understanding the precise meaning of terms used throughout this design document. Each definition includes not just what the term means, but why the distinction matters for implementation and security. Understanding these terms precisely is crucial because OAuth2 and OIDC are security protocols where ambiguous language can lead to vulnerabilities.

OAuth2 Core Protocol Terms

The foundational OAuth2 specification defines a specific vocabulary that must be understood precisely for secure implementation. These terms form the building blocks of all OAuth2 flows and security mechanisms.

Term	Definition	Implementation Significance
OAuth2	An authorization framework enabling third-party applications to obtain limited access to user accounts without exposing user credentials. Defined in RFC 6749.	The core protocol that all our components implement. Understanding OAuth2 versus authentication protocols is crucial for security boundaries.
Authorization Server	The server that issues access tokens after successfully authenticating the resource owner and obtaining authorization. In our implementation, this is the complete OAuth2 provider system.	This is what we're building - the central security component that all clients trust for token issuance and validation.
Resource Server	The server hosting protected resources, capable of accepting and responding to protected resource requests using access tokens.	External systems that will validate our JWT tokens and enforce scope-based access control.
Client Application	An application making protected resource requests on behalf of the resource owner and with the resource owner's authorization. Also called "OAuth2 client."	Third-party applications that register with our system and request user authorization through standardized flows.
Resource Owner	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."	The user who owns the data and grants permission for client applications to access it on their behalf.
Authorization Grant	A credential representing the resource owner's authorization to access protected resources. OAuth2 defines several grant types including authorization code, client credentials, and refresh token.	The security proof that a client has been authorized to access specific resources with specific scopes.
Authorization Code	A temporary credential that represents the resource owner's authorization, designed to be exchanged for access tokens. Short-lived and single-use.	Critical security token that must be bound to client and redirect URI, expire quickly (max 10 minutes), and be used exactly once.
Access Token	A credential used to access protected resources. In our implementation, these are signed JWT tokens containing claims about the authorization.	The actual credential that resource servers validate. Must contain scope, expiration, and subject information for proper access control.
Refresh Token	A credential used to obtain new access tokens without requiring resource owner re-authorization. Long-lived but revocable.	Enables long-term access while maintaining access token security through short lifetimes. Requires family tracking for theft detection.

Token and Credential Management Terms

Token lifecycle management is central to OAuth2 security. Understanding the precise meaning of token-related terms is essential for implementing secure token handling, validation, and revocation.

Term	Definition	Security Implications
JWT (JSON Web Token)	A compact, URL-safe means of representing claims between two parties. Consists of header, payload, and signature components separated by dots. Defined in RFC 7519.	Our access tokens are JWTs, enabling stateless validation by resource servers. Signature verification and claims validation are critical for security.
Token Introspection	An OAuth2 extension (RFC 7662) that allows resource servers to query the authorization server about the state of an access token.	Enables real-time token validation and revocation checking for resource servers that cannot validate JWTs locally.
Token Revocation	An OAuth2 extension (RFC 7009) that allows clients to notify the authorization server that a token should no longer be considered valid.	Critical for security incidents and user-initiated logout. Requires tracking revoked token IDs until expiration.
Client ID	A public identifier for OAuth2 client applications, generated during client registration. Not confidential and can appear in redirect URIs.	Used to identify clients in authorization requests. Must be unpredictable to prevent client enumeration attacks.
Client Secret	A confidential credential known only to the client and authorization server, used for client authentication.	Must be stored as bcrypt hash, never in plaintext. Required for confidential clients but not public clients (mobile/SPA).
Redirect URI	The URI where the authorization server redirects the user after completing authorization. Must be registered during client registration.	Critical security boundary - exact string matching prevents authorization code theft through redirect manipulation.
Scope	A parameter that defines the specific permissions being requested by the client application. Maps to specific resource access or user claims.	Determines what data access is granted. Must be validated against client's registered scopes and user consent decisions.
State Parameter	An unguessable random string used to maintain state between authorization request and callback, providing CSRF protection.	Prevents cross-site request forgery attacks in authorization flows. Must be unpredictable and verified on callback.

OpenID Connect Terms

OpenID Connect adds an identity layer on top of OAuth2, introducing additional concepts for user authentication and profile information sharing.

Term	Definition	OIDC-Specific Purpose
OpenID Connect (OIDC)	An identity layer on top of OAuth2, enabling clients to verify the identity of end-users and obtain basic profile information. Defined in OpenID Connect Core 1.0.	Adds standardized user authentication and profile sharing to OAuth2's authorization framework.
ID Token	A security token that contains claims about the authentication of an end-user by an authorization server. Always a signed JWT in OIDC.	Provides proof of user authentication to client applications. Contains authentication event information, not for API access.
UserInfo Endpoint	An OIDC endpoint that returns claims about the authenticated end-user, protected by an OAuth2 access token.	Provides user profile information based on granted scopes. Must validate access token and filter claims appropriately.
Claims	Pieces of information about an entity (typically the end-user). OIDC defines standard claims like <code>sub</code> , <code>name</code> , <code>email</code> , and <code>picture</code> .	User profile data returned by UserInfo endpoint. Must be filtered based on granted scopes and consent decisions.
Standard Claims	OIDC-defined user profile fields such as <code>sub</code> (subject identifier), <code>name</code> , <code>given_name</code> , <code>family_name</code> , <code>email</code> , <code>picture</code> , etc.	Provides interoperability between OIDC providers and clients. Implementation must support core claims for compliance.
Scope to Claims Mapping	The system that determines which user profile claims are returned based on the OAuth2 scopes granted to the client.	Critical for privacy - ensures clients only receive user data they're authorized to access based on granted scopes.
Discovery Document	A JSON document describing the OIDC provider's configuration, available at the well-known configuration endpoint.	Enables automatic client configuration. Must accurately reflect provider capabilities and endpoint URLs.
Well-Known Configuration Endpoint	A standardized URL (<code>/well-known/openid-configuration</code>) where OIDC providers publish their discovery document.	Provides automated discovery of provider capabilities and endpoints for OIDC clients.

PKCE and Security Extension Terms

Proof Key for Code Exchange (PKCE) and other OAuth2 security extensions provide additional protection against various attack vectors.

Term	Definition	Security Purpose
PKCE (Proof Key for Code Exchange)	An OAuth2 security extension (RFC 7636) that provides protection against authorization code interception attacks, especially for public clients.	Essential for mobile and single-page applications. Prevents authorization code theft through dynamic client secret generation.
Code Verifier	A cryptographically random string used as input to the PKCE challenge. Generated by the client and never transmitted directly to the authorization server.	Secret proof held by legitimate client. Must be cryptographically random and sufficiently long (43-128 characters).
Code Challenge	A derived value from the code verifier, transmitted during authorization request. Can be plain text or SHA256 hash of verifier.	Public value that allows verification of code verifier without exposing it. SHA256 method required for security.
Code Challenge Method	The method used to derive the code challenge from the code verifier. Either <code>plain</code> or <code>S256</code> (SHA256).	<code>S256</code> is required for security. <code>plain</code> method is deprecated and should not be accepted for public clients.
Client Type	Classification of OAuth2 clients as either "confidential" (can securely store credentials) or "public" (cannot securely store credentials).	Determines authentication requirements. Public clients must use PKCE and cannot have client secrets.
Token Family	A security concept where refresh tokens are grouped into families for tracking and coordinated revocation when theft is detected.	Enables detection of refresh token theft - if old token is reused, entire family can be revoked as security measure.
Token Rotation	Security practice of issuing a new refresh token each time it's used and invalidating the old one.	Limits blast radius of token theft and enables detection of concurrent usage indicating compromise.

Cryptographic and Security Terms

OAuth2 and OIDC rely heavily on cryptographic primitives and security concepts that must be implemented correctly to maintain system security.

Term	Definition	Cryptographic Significance
JWT Signature Verification	The process of validating that a JWT token was issued by the claimed authorization server and has not been tampered with.	Critical for access token security. Resource servers must verify signature before trusting token claims.
Bearer Token Authentication	An OAuth2 token usage method where the access token is included in the <code>Authorization: Bearer <token></code> HTTP header.	Standard method for presenting access tokens to resource servers. Requires HTTPS to prevent token interception.
Constant-Time Comparison	Cryptographic technique for comparing sensitive values (like tokens or secrets) in a way that doesn't leak information through timing differences.	Prevents timing attacks that could be used to guess tokens or secrets through response time analysis.
JTI Claim	The "JWT ID" claim providing a unique identifier for a JWT token, used for revocation tracking and preventing token replay.	Enables efficient revocation checking and prevents token replay attacks by tracking used token IDs.
JWKS (JSON Web Key Set)	A set of cryptographic keys published by the authorization server for verifying JWT signatures. Available at the JWKS endpoint.	Enables distributed JWT validation without sharing private keys. Resource servers fetch public keys for signature verification.
Asymmetric Cryptography	Cryptographic system using key pairs (public/private) where tokens are signed with private key and verified with public key.	Enables stateless token validation - resource servers can verify tokens without contacting authorization server.
Token Binding	Security technique that cryptographically or logically associates tokens with their authorized usage context (client, user, session).	Prevents token misuse by ensuring tokens can only be used in their intended context.
Entropy	Measure of randomness or unpredictability in generated tokens, codes, and secrets. Higher entropy means harder to guess.	Critical for security - insufficient entropy in tokens/codes makes them vulnerable to brute force attacks.

Client and Application Management Terms

Managing OAuth2 clients and their lifecycle involves specific concepts for registration, authentication, and authorization.

Term	Definition	Management Purpose
Client Registration	The process of establishing an OAuth2 client application's identity and configuration with the authorization server.	Creates trusted relationship between client and authorization server. Generates client credentials and stores configuration.
Dynamic Client Registration	OAuth2 extension (RFC 7591) allowing clients to register programmatically via API rather than manual administrative process.	Enables automated client onboarding but requires additional security controls to prevent abuse.
Client Authentication	Process of verifying OAuth2 client identity using client credentials, client assertion, or other authentication methods.	Ensures token requests come from legitimate clients. Required for confidential clients, not applicable to public clients.
Grant Types	The method by which a client obtains an access token. Common types include authorization_code, client_credentials, and refresh_token.	Determines the flow and security requirements for token issuance. Must be validated against client's registered grant types.
Response Type	Parameter in authorization requests indicating the desired response. Typically "code" for authorization code flow.	Determines what the authorization endpoint returns. Must be validated against client's registered response types.
Client Credentials Grant	OAuth2 flow where clients authenticate directly with the authorization server using their credentials to obtain access tokens for machine-to-machine access.	Enables server-to-server authentication without user involvement. No refresh tokens issued since client can re-authenticate.
Redirect URI Validation	Security process ensuring authorization callbacks only go to pre-registered, exact-match URIs.	Prevents authorization code theft through redirect manipulation. Must use exact string matching, not pattern matching.

Consent and User Experience Terms

User consent management and experience concepts are crucial for privacy compliance and user trust in OAuth2 systems.

Term	Definition	User Privacy Purpose
Consent Screen	User interface where resource owners review and approve or deny client applications' requests for access to protected resources.	Provides informed consent opportunity. Must clearly describe requested permissions and data access.
Consent Record	Persistent storage of user authorization decisions, including granted scopes, timestamp, and revocation status.	Enables consent persistence to avoid repeated prompts and supports user consent management and revocation.
Consent Persistence	System capability to remember user consent decisions so repeat authorization requests can skip the consent screen for previously approved scopes.	Improves user experience while maintaining security. Must respect consent expiration and revocation.
Consent Revocation	User action to withdraw previously granted authorization, immediately invalidating associated access and making future requests require new consent.	Critical for user privacy control. Must immediately revoke associated tokens and reset consent state.
Consent Fatigue	User frustration resulting from excessive consent prompts, leading to less careful review of authorization requests.	Design consideration for consent flows. Balance security with user experience through intelligent consent persistence.
Scope Description	Human-readable explanation of what access each OAuth2 scope grants, displayed on consent screens.	Essential for informed consent. Must clearly explain data access implications in user-friendly language.
Claims Filtering	Process of returning only user profile data that the granted scopes authorize, preventing over-disclosure of user information.	Privacy protection ensuring clients only receive data they're authorized to access based on user consent.
Consent Expiration	Automatic invalidation of consent records after a specified time period, requiring fresh user authorization.	Balances user convenience with security by ensuring long-term access requires periodic reauthorization.

Error Handling and Security Terms

OAuth2 defines specific error handling patterns and security concepts that must be implemented correctly to maintain protocol compliance and security.

Term	Definition	Security Context
OAuth2 Error Response Format	Standardized structure for communicating errors in OAuth2 flows, including error code, description, and optional error URI.	Ensures interoperability while avoiding information disclosure that could aid attackers.
Error Codes	Standardized OAuth2 error identifiers like <code>invalid_request</code> , <code>invalid_client</code> , <code>invalid_grant</code> , <code>unauthorized_client</code> , etc.	Provides precise error communication for client handling while avoiding security-sensitive details.
Timing Attack	Side-channel attack that uses variations in response times to infer information about secrets or internal state.	Critical concern for token validation and client authentication. Must use constant-time comparisons for security operations.
Replay Attack	Attack attempting to reuse intercepted tokens, authorization codes, or other credentials to gain unauthorized access.	Prevented through single-use codes, token expiration, and nonce tracking.
Token Theft	Unauthorized acquisition of valid OAuth2 tokens through interception, storage compromise, or other means.	Mitigated through short token lifetimes, refresh token rotation, and revocation capabilities.
Client Impersonation	Attack where malicious client attempts to masquerade as legitimate client to steal authorization codes or tokens.	Prevented through client authentication, redirect URI validation, and PKCE verification.
Authorization Code Injection	Attack where stolen authorization code is injected into legitimate client session to gain unauthorized access.	Prevented by PKCE, state parameter validation, and authorization code binding to client.
Cross-Site Request Forgery (CSRF)	Attack that tricks users into performing unintended actions on applications where they're authenticated.	Prevented in OAuth2 through state parameter validation and proper session management.

Performance and Operational Terms

Production OAuth2 deployments require understanding of performance, monitoring, and operational concepts.

Term	Definition	Operational Purpose
Rate Limiting	Mechanism to control the frequency of requests to OAuth2 endpoints to prevent abuse and ensure fair resource usage.	Prevents brute force attacks and denial of service. Must be applied to all public endpoints with appropriate limits.
Token Validation Caching	Performance optimization that stores JWT validation results temporarily to reduce cryptographic operations and database queries.	Improves response times for frequently validated tokens while maintaining security through appropriate cache expiration.
Revocation List	Data structure tracking explicitly revoked tokens that haven't yet expired naturally.	Enables immediate token invalidation while avoiding storage growth through automatic cleanup of expired entries.
Circuit Breaker	Reliability pattern that prevents cascade failures by temporarily blocking requests to failing dependencies.	Protects OAuth2 provider from external service failures while providing graceful degradation.
Graceful Degradation	System design approach that maintains core functionality even when some components or dependencies are unavailable.	Ensures OAuth2 provider remains available for critical operations even during partial system failures.
Token Lifecycle Management	Comprehensive approach to managing tokens from creation through expiration, including usage tracking, revocation, and cleanup.	Essential for security, compliance, and system performance in production OAuth2 deployments.
Audit Logging	Comprehensive logging of all OAuth2 operations for security monitoring, compliance, and forensic analysis.	Required for security incident response and compliance with regulations like GDPR, SOC 2, and financial services requirements.
Metrics and Monitoring	Observability system tracking OAuth2 provider performance, security events, and operational health.	Enables proactive incident response and capacity planning for production OAuth2 deployments.

Advanced OAuth2 Extensions

Beyond core OAuth2 and OIDC, several extensions provide additional capabilities for specialized use cases and enhanced security.

Term	Definition	Extension Purpose
Device Authorization Grant	OAuth2 extension (RFC 8628) enabling authorization for devices with limited input capabilities by using a separate device for user interaction.	Enables OAuth2 for smart TVs, IoT devices, and command-line tools that cannot display web interfaces.
Pushed Authorization Requests (PAR)	Security extension (RFC 9126) requiring clients to submit authorization parameters via POST request before redirecting users to authorization endpoint.	Enhances security by preventing parameter manipulation and enables larger request sizes through direct HTTP rather than URL parameters.
Rich Authorization Requests (RAR)	Extension enabling structured, fine-grained authorization requests beyond simple scope strings.	Allows complex permission requests with specific contexts, resources, and action limitations.
Financial-grade API (FAPI)	Security profile for OAuth2 providing enhanced protection for high-risk environments like financial services.	Adds requirements for mutual TLS, certificate-bound tokens, and additional security controls for sensitive financial data access.
Mutual TLS	Authentication method using client certificates for strong client authentication and token binding.	Provides cryptographic client authentication and enables certificate-bound access tokens for enhanced security.
Identity Federation	Integration pattern allowing OAuth2 provider to accept authentication from external identity providers like SAML or social login services.	Enables enterprise SSO and social login while maintaining OAuth2 interface for client applications.

Implementation and Testing Terms

Specific concepts related to implementing and testing OAuth2 providers require precise understanding for successful development.

Term	Definition	Development Context
Protocol Compliance Testing	Systematic validation that OAuth2 implementation correctly follows RFC specifications and handles edge cases appropriately.	Ensures interoperability with standard OAuth2 clients and libraries. Critical for production readiness.
Security Property Testing	Testing approach focused on verifying cryptographic properties and resistance to known attack patterns.	Validates that security mechanisms actually provide intended protection against realistic threats.
Attack Simulation	Testing methodology that systematically attempts common OAuth2 attacks to verify security controls are effective.	Provides confidence that security measures work in practice, not just in theory.
Load Testing	Performance testing that validates OAuth2 provider behavior under realistic traffic loads and stress conditions.	Ensures production readiness and helps identify performance bottlenecks before deployment.
Integration Testing	Testing approach that validates OAuth2 provider functionality with real client libraries and applications.	Verifies interoperability and identifies integration issues that unit tests might miss.
Milestone Checkpoint	Structured verification point during development that validates specific functionality is working correctly before proceeding.	Ensures incremental progress and catches issues early in development process.
Compliance Test Framework	Automated testing infrastructure that validates OAuth2 implementation against specification requirements.	Provides ongoing assurance that code changes don't break protocol compliance.

Understanding these terms precisely is essential for implementing a secure, compliant, and maintainable OAuth2/OIDC provider. Each term represents concepts that have specific security, functional, or operational implications that must be understood for successful implementation.