MCP Server Implementation - Pseudocode Template

Overview

This pseudocode template provides a complete, language-agnostic reference implementation for building a production-ready MCP (Model Context Protocol) server with OAuth 2.1 + PKCE authentication. The template expands on the pseudocode examples in the presentation and implementation documentation, providing comprehensive coverage of all server components.

Purpose: Serve as a blueprint for translating core MCP server logic into any programming language.

Target Audience: Developers and architects building ServiceNow MCP integrations or other enterprise MCP server implementations.

Table of Contents

---

1. Configuration and Constants

pseudocode

```
// Server Configuration
CONSTANT SERVER_PORT = 3000
CONSTANT SERVER_HOST = "0.0.0.0"
CONSTANT JWT_ISSUER = "https://mcp-server.example.com"

// Token Lifetimes (in seconds)
CONSTANT AUTHORIZATION_CODE_LIFETIME = 300  // 5 minutes
CONSTANT ACCESS_TOKEN_LIFETIME = 3600       // 1 hour
CONSTANT REFRESH_TOKEN_LIFETIME = 2592000   // 30 days

// Security Configuration
CONSTANT JWT_ALGORITHM = "HS256"
CONSTANT JWT_SECRET = getEnvironmentVariable("JWT_SECRET")  // Load from secure environment
CONSTANT DCR_REGISTRATION_TOKEN = getEnvironmentVariable("DCR_TOKEN")  // Optional

// Rate Limiting Configuration
CONSTANT RATE_LIMIT_WINDOW_MS = 900000     // 15 minutes
CONSTANT RATE_LIMIT_MAX_REQUESTS = 100     // Max requests per window
CONSTANT OAUTH_RATE_LIMIT_MAX = 10         // More restrictive for OAuth endpoints

// MCP Protocol Configuration
CONSTANT MCP_PROTOCOL_VERSION = "2025-06-18"
CONSTANT MCP_SERVER_NAME = "mcp-oauth21-server"
CONSTANT MCP_SERVER_VERSION = "3.3.0"

// PKCE Configuration
CONSTANT SUPPORTED_PKCE_METHODS = ["S256", "plain"]
CONSTANT RECOMMENDED_PKCE_METHOD = "S256"
```

---

2. Data Structures

pseudocode

```
// In-memory storage structures (replace with persistent storage in production)
GLOBAL registeredClients = Map()      // Map<client_id, ClientRecord>
GLOBAL authorizationCodes = Map()     // Map<code, AuthCodeRecord>
GLOBAL activeRefreshTokens = Set()    // Set<refresh_token_jti>
GLOBAL revokedTokens = Set()          // Set<token_jti> - Token blacklist

STRUCTURE ClientRecord:
    client_id: string
    client_secret: string
    client_name: string
    redirect_uris: array of string
    grant_types: array of string
    response_types: array of string
    use_pkce: boolean
    created_at: timestamp
END STRUCTURE

STRUCTURE AuthCodeRecord:
    code: string
    client_id: string
    redirect_uri: string
    scope: string
    code_challenge: string
    code_challenge_method: string
    user_id: string
    expires_at: timestamp
    used: boolean
END STRUCTURE

STRUCTURE JWTAccessTokenPayload:
    sub: string          // user_id
    client_id: string
    scope: string
    type: "access"
    iat: timestamp
    exp: timestamp
    iss: string
    jti: string          // Unique token identifier for revocation
END STRUCTURE

STRUCTURE JWTRefreshTokenPayload:
    sub: string          // user_id
    client_id: string
    scope: string
    type: "refresh"
    rotation_count: integer
    iat: timestamp
    exp: timestamp
    iss: string
    jti: string          // Unique token identifier for revocation
END STRUCTURE
```

---

3. Server Initialization

```
pseudocode
```

```
FUNCTION initializeServer()
    LOG "Starting MCP OAuth 2.1 Server"
    LOG "Issuer: " + JWT_ISSUER
    LOG "Protocol Version: " + MCP_PROTOCOL_VERSION

    // Validate required configuration
    IF JWT_SECRET is empty THEN
        ERROR "JWT_SECRET environment variable is required"
        EXIT
    END IF

    // Initialize storage connections (if using persistent storage)
    initializeTokenBlacklist()

    // Register routes
    registerOAuthRoutes()
    registerMCPRoutes()
    registerMetadataRoutes()
    registerHealthCheckRoute()

    // Start HTTP server
    startHTTPServer(SERVER_HOST, SERVER_PORT)

    LOG "Server listening on " + SERVER_HOST + ":" + SERVER_PORT
END FUNCTION

FUNCTION initializeTokenBlacklist()
    // Initialize connection to token blacklist storage
    // This could be Redis, database, or in-memory for development
    LOG "Initializing token blacklist storage"

    // Example: Connect to storage backend
    // revokedTokens = connectToStorageBackend()
END FUNCTION
```

---

4. OAuth 2.1 Endpoints

4.1 Dynamic Client Registration (DCR)

pseudocode

```
ROUTE POST "/register"
  FUNCTION handleDynamicClientRegistration(request, response)
    LOG "=== DCR REQUEST ==="

    // Optional: Validate DCR authorization token
    IF DCR_REGISTRATION_TOKEN is configured THEN
      authHeader = request.headers["authorization"]

      IF NOT authHeader THEN
        RETURN response(401, {
          error: "invalid_token",
          error_description: "Missing authorization header"
        })
      END IF

      IF NOT authHeader.startsWith("Bearer ") THEN
        RETURN response(401, {
          error: "invalid_token",
          error_description: "Invalid authorization format"
        })
      END IF

      providedToken = authHeader.substring(7)  // Remove "Bearer "

      IF providedToken != DCR_REGISTRATION_TOKEN THEN
        RETURN response(401, {
          error: "invalid_token",
          error_description: "Invalid registration token"
        })
      END IF

      LOG "âœ“ DCR token validated"
    END IF

    // Extract client registration details
    client_name = request.body.client_name OR "Unnamed Client"
    redirect_uris = request.body.redirect_uris OR []
    grant_types = request.body.grant_types OR ["authorization_code", "refresh_token"]
    use_pkce = request.body.use_pkce OR false

    // Validate redirect URIs (required)
    IF redirect_uris.length = 0 THEN
      RETURN response(400, {
            error: "invalid_redirect_uri",
            error_description: "At least one redirect_uri is required"
         })
      END IF

      // Generate unique client credentials
      client_id = generateUUID()
      client_secret = generateSecureToken(32)

      // Create client record
      clientRecord = ClientRecord {
         client_id: client_id,
         client_secret: client_secret,
         client_name: client_name,
         redirect_uris: redirect_uris,
         grant_types: grant_types,
         response_types: ["code"],
         use_pkce: use_pkce,
         created_at: currentTimestamp()
      }

      // Store client (persistent storage in production)
      registeredClients.set(client_id, clientRecord)

      LOG "âœ“ Client registered: " + client_id
      LOG "Client name: " + client_name
      LOG "PKCE enabled: " + use_pkce

      // Return credentials (RFC 7591 format)
      RETURN response(201, {
         client_id: client_id,
         client_secret: client_secret,
         client_name: client_name,
         redirect_uris: redirect_uris,
         grant_types: grant_types,
         response_types: ["code"],
         token_endpoint_auth_method: "client_secret_post"
      })
   END FUNCTION
END ROUTE
```

4.2 Authorization Endpoint

```
pseudocode
```

```
ROUTE GET "/oauth/authorize"
  FUNCTION handleAuthorization(request, response)
    LOG "=== AUTHORIZATION REQUEST ==="

    // Extract OAuth 2.1 parameters
    client_id = request.query.client_id
    redirect_uri = request.query.redirect_uri
    response_type = request.query.response_type
    scope = request.query.scope OR "openid email profile"
    state = request.query.state
    code_challenge = request.query.code_challenge
    code_challenge_method = request.query.code_challenge_method

    // Validate response_type
    IF response_type != "code" THEN
      errorParams = "error=unsupported_response_type"
      IF state THEN
        errorParams = errorParams + "&state=" + state
      END IF
      RETURN redirect(redirect_uri + "?" + errorParams)
    END IF

    // Validate required parameters
    IF NOT client_id OR NOT redirect_uri THEN
      RETURN response(400, {
        error: "invalid_request",
        error_description: "Missing required parameters: client_id and redirect_uri"
      })
    END IF

    // PKCE validation (mandatory in OAuth 2.1)
    IF NOT code_challenge OR NOT code_challenge_method THEN
      RETURN response(400, {
        error: "invalid_request",
        error_description: "PKCE required: code_challenge and code_challenge_method must be provided"
      })
    END IF

    // Validate PKCE method
    IF code_challenge_method NOT IN SUPPORTED_PKCE_METHODS THEN
      RETURN response(400, {
        error: "invalid_request",
        error_description: "code_challenge_method must be S256 or plain"
    })
END IF

// Validate client exists
client = registeredClients.get(client_id)
IF NOT client THEN
    RETURN response(400, {
        error: "invalid_client",
        error_description: "Client not found"
    })
END IF

LOG "âœ“ Client validated: " + client_id

// Validate redirect URI is registered
IF NOT client.redirect_uris.includes(redirect_uri) THEN
    RETURN response(400, {
        error: "invalid_request",
        error_description: "redirect_uri not registered for this client"
    })
END IF

LOG "âœ“ Redirect URI validated"
LOG "âœ“ PKCE parameters validated"
LOG "Method: " + code_challenge_method

// NOTE: In production, display user consent screen here
// For service-to-service integration, auto-approve
// ServiceNow handles user authentication, so simulate approved user
user_id = generateUUID()  // Simulated authenticated user

LOG "âš ï¸  SIMULATED USER AUTHENTICATION - user_id: " + user_id
LOG "âš ï¸  In production, implement real user authentication here"

// Generate authorization code
authCode = generateSecureToken(32)

// Store authorization code with PKCE parameters
authCodeRecord = AuthCodeRecord {
    code: authCode,
    client_id: client_id,
    redirect_uri: redirect_uri,
    scope: scope,
    code_challenge: code_challenge,
        code_challenge_method: code_challenge_method,
        user_id: user_id,
        expires_at: currentTimestamp() + AUTHORIZATION_CODE_LIFETIME,
        used: false
      }

      authorizationCodes.set(authCode, authCodeRecord)

      LOG "âœ“ Authorization code generated"
      LOG "Code expires in " + AUTHORIZATION_CODE_LIFETIME + " seconds"

      // Redirect back to client with authorization code
      redirectParams = "code=" + authCode
      IF state THEN
        redirectParams = redirectParams + "&state=" + state
      END IF

      redirectURL = redirect_uri + "?" + redirectParams

      LOG "Redirecting to: " + redirectURL

      RETURN redirect(redirectURL)
  END FUNCTION
END ROUTE
```

4.3 Token Endpoint

```
pseudocode
```

```
ROUTE POST "/oauth/token"
  APPLY rateLimitMiddleware(OAUTH_RATE_LIMIT_MAX)

  FUNCTION handleTokenExchange(request, response)
    LOG "=== TOKEN REQUEST ==="
    LOG "Grant type: " + request.body.grant_type

    grant_type = request.body.grant_type

    // Validate grant_type
    IF NOT grant_type THEN
      RETURN response(400, {
        error: "invalid_request",
        error_description: "grant_type is required"
      })
    END IF

    // Route to appropriate grant handler
    IF grant_type = "authorization_code" THEN
      RETURN handleAuthorizationCodeGrant(request, response)
    ELSE IF grant_type = "refresh_token" THEN
      RETURN handleRefreshTokenGrant(request, response)
    ELSE
      RETURN response(400, {
        error: "unsupported_grant_type",
        error_description: "Supported grant types: authorization_code, refresh_token"
      })
    END IF
  END FUNCTION

  // Authorization Code Grant Handler
  FUNCTION handleAuthorizationCodeGrant(request, response)
    LOG "=== AUTHORIZATION CODE GRANT ==="

    // Extract parameters
    code = request.body.code
    redirect_uri = request.body.redirect_uri
    client_id = request.body.client_id
    client_secret = request.body.client_secret
    code_verifier = request.body.code_verifier

    // Validate required parameters
    IF NOT code OR NOT redirect_uri OR NOT client_id OR NOT client_secret THEN
    RETURN response(400, {
        error: "invalid_request",
        error_description: "Missing required parameters"
    })
END IF

// Validate client credentials
client = registeredClients.get(client_id)
IF NOT client OR client.client_secret != client_secret THEN
    RETURN response(401, {
        error: "invalid_client",
        error_description: "Invalid client credentials"
    })
END IF

LOG "âœ“ Client authenticated: " + client_id

// Retrieve authorization code
authCodeRecord = authorizationCodes.get(code)
IF NOT authCodeRecord THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Authorization code not found"
    })
END IF

// Check if code has been used (prevent replay attacks)
IF authCodeRecord.used THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Authorization code already used"
    })
END IF

// Check if code has expired
IF currentTimestamp() > authCodeRecord.expires_at THEN
    authorizationCodes.delete(code)
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Authorization code expired"
    })
END IF

// Validate client_id matches
```

```
IF authCodeRecord.client_id != client_id THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Authorization code was issued to a different client"
    })
END IF

// Validate redirect_uri matches
IF authCodeRecord.redirect_uri != redirect_uri THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "redirect_uri does not match"
    })
END IF

LOG "âœ“ Authorization code validated"

// PKCE VALIDATION - code_verifier is MANDATORY
IF NOT code_verifier THEN
    RETURN response(400, {
        error: "invalid_request",
        error_description: "code_verifier is required (PKCE)"
    })
END IF

LOG "=== PKCE VALIDATION ==="
LOG "Method: " + authCodeRecord.code_challenge_method
LOG "Stored challenge: " + authCodeRecord.code_challenge

// Validate code_verifier against stored challenge
isValid = validatePKCE(
    code_verifier,
    authCodeRecord.code_challenge,
    authCodeRecord.code_challenge_method
)

IF NOT isValid THEN
    LOG "âœ— PKCE validation failed"
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Invalid code_verifier"
    })
END IF
```

```
    LOG "âœ“ PKCE validation successful"

    // Mark authorization code as used and delete
    authCodeRecord.used = true
    authorizationCodes.delete(code)

    // Issue access token and refresh token
    access_token = createAccessToken(
        authCodeRecord.user_id,
        client_id,
        authCodeRecord.scope
    )

    refresh_token = createRefreshToken(
        authCodeRecord.user_id,
        client_id,
        authCodeRecord.scope,
        0  // Initial rotation_count
    )

    LOG "âœ“ Tokens issued successfully"

    // Return token response (RFC 6749 format)
    RETURN response(200, {
        access_token: access_token,
        token_type: "Bearer",
        expires_in: ACCESS_TOKEN_LIFETIME,
        refresh_token: refresh_token,
        scope: authCodeRecord.scope
    })
END FUNCTION

// Refresh Token Grant Handler
FUNCTION handleRefreshTokenGrant(request, response)
    LOG "=== REFRESH TOKEN GRANT ==="

    // Extract parameters
    refresh_token = request.body.refresh_token
    client_id = request.body.client_id
    client_secret = request.body.client_secret
    scope = request.body.scope  // Optional - can request reduced scope

    // Validate required parameters
    IF NOT refresh_token OR NOT client_id OR NOT client_secret THEN
    RETURN response(400, {
        error: "invalid_request",
        error_description: "Missing required parameters"
    })
END IF

// Validate client credentials
client = registeredClients.get(client_id)
IF NOT client OR client.client_secret != client_secret THEN
    RETURN response(401, {
        error: "invalid_client",
        error_description: "Invalid client credentials"
    })
END IF

LOG "âœ“ Client authenticated: " + client_id

// Validate refresh token
TRY
    decoded = verifyJWT(refresh_token, JWT_SECRET, {
        algorithms: [JWT_ALGORITHM],
        issuer: JWT_ISSUER
    })

    LOG "âœ“ Refresh token signature valid"
CATCH error
    LOG "âœ— Refresh token validation failed: " + error.message
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Invalid refresh token"
    })
END TRY

// Verify token type
IF decoded.type != "refresh" THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Token is not a refresh token"
    })
END IF

// Check if token has been revoked
IF isTokenRevoked(decoded.jti) THEN
    LOG "âœ— Refresh token has been revoked"
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Refresh token has been revoked"
    })
END IF

// Validate client_id matches token
IF decoded.client_id != client_id THEN
    RETURN response(400, {
        error: "invalid_grant",
        error_description: "Refresh token was issued to a different client"
    })
END IF

LOG "âœ“ Refresh token validated"

// Revoke old refresh token (token rotation)
addTokenToBlacklist(decoded.jti, decoded.exp)

LOG "âœ“ Old refresh token revoked"

// Issue new tokens with incremented rotation count
new_access_token = createAccessToken(
    decoded.sub,
    client_id,
    scope OR decoded.scope
)

new_refresh_token = createRefreshToken(
    decoded.sub,
    client_id,
    scope OR decoded.scope,
    decoded.rotation_count + 1
)

LOG "âœ“ New tokens issued (rotation count: " + (decoded.rotation_count + 1) + ")"

// Return token response
RETURN response(200, {
    access_token: new_access_token,
    token_type: "Bearer",
    expires_in: ACCESS_TOKEN_LIFETIME,
    refresh_token: new_refresh_token,
    scope: scope OR decoded.scope
    })
  END FUNCTION
END ROUTE
```

4.4 Token Revocation Endpoint

```
pseudocode
```

```
ROUTE POST "/oauth/revoke"
  FUNCTION handleTokenRevocation(request, response)
    LOG "=== TOKEN REVOCATION REQUEST ==="

    // Extract parameters
    token = request.body.token
    client_id = request.body.client_id
    client_secret = request.body.client_secret

    // RFC 7009: token parameter is required
    IF NOT token THEN
      RETURN response(400, {
        error: "invalid_request",
        error_description: "token is required"
      })
    END IF

    // Optional: Validate client credentials
    IF client_id AND client_secret THEN
      client = registeredClients.get(client_id)
      IF NOT client OR client.client_secret != client_secret THEN
        RETURN response(401, {
          error: "invalid_client",
          error_description: "Invalid client credentials"
        })
      END IF
      LOG "âœ“ Client authenticated"
    END IF

    // Decode token to get jti (token identifier)
    TRY
      decoded = decodeJWT(token)  // Decode without verification

      IF decoded.jti THEN
        // Add token to blacklist
        addTokenToBlacklist(decoded.jti, decoded.exp)
        LOG "âœ“ Token revoked: " + decoded.jti
      END IF
    CATCH error
      // Token might be malformed, but still return 200 per RFC 7009
      LOG "âš ï¸  Could not decode token: " + error.message
    END TRY
```

```
        // RFC 7009: Always return 200 OK (even if token not found)
        RETURN response(200)
    END FUNCTION
END ROUTE
```

---

5. JWT Token Management

```
pseudocode
```

```
FUNCTION createAccessToken(user_id, client_id, scope)
    LOG "Creating access token for user: " + user_id

    // Generate unique token identifier
    token_jti = generateUUID()

    // Create token payload
    payload = JWTAccessTokenPayload {
        sub: user_id,
        client_id: client_id,
        scope: scope,
        type: "access",
        iat: currentTimestamp(),
        exp: currentTimestamp() + ACCESS_TOKEN_LIFETIME,
        iss: JWT_ISSUER,
        jti: token_jti
    }

    // Sign token
    access_token = signJWT(payload, JWT_SECRET, JWT_ALGORITHM)

    RETURN access_token
END FUNCTION

FUNCTION createRefreshToken(user_id, client_id, scope, rotation_count)
    LOG "Creating refresh token for user: " + user_id

    // Generate unique token identifier
    token_jti = generateUUID()

    // Create token payload
    payload = JWTRefreshTokenPayload {
        sub: user_id,
        client_id: client_id,
        scope: scope,
        type: "refresh",
        rotation_count: rotation_count,
        iat: currentTimestamp(),
        exp: currentTimestamp() + REFRESH_TOKEN_LIFETIME,
        iss: JWT_ISSUER,
        jti: token_jti
    }
    // Sign token
    refresh_token = signJWT(payload, JWT_SECRET, JWT_ALGORITHM)

    RETURN refresh_token
END FUNCTION

FUNCTION validateToken(token)
    TRY
        // Verify JWT signature and decode claims
        payload = verifyJWT(token, JWT_SECRET, {
            algorithms: [JWT_ALGORITHM],
            issuer: JWT_ISSUER
        })

        // Check if token has been revoked
        IF isTokenRevoked(payload.jti) THEN
            LOG "âœ— Token has been revoked"
            RETURN null
        END IF

        // Token is valid
        LOG "âœ“ Token validated for client: " + payload.client_id
        RETURN payload

    CATCH error
        // Invalid signature, expired, or malformed
        LOG "âœ— Token validation failed: " + error.message
        RETURN null
    END TRY
END FUNCTION
```

---

6. PKCE Validation

```
pseudocode
```

```
FUNCTION validatePKCE(code_verifier, code_challenge, method)
    // Validate using S256 method (RECOMMENDED)
    IF method = "S256" THEN
        // Hash the code_verifier using SHA-256
        computed_challenge = base64URLEncode(sha256(code_verifier))

        // Compare with stored challenge
        IF computed_challenge = code_challenge THEN
            RETURN true
        ELSE
            RETURN false
        END IF

    // Validate using plain method (ALLOWED but discouraged)
    ELSE IF method = "plain" THEN
        // Direct comparison (no hashing)
        IF code_verifier = code_challenge THEN
            RETURN true
        ELSE
            RETURN false
        END IF

    // Invalid method
    ELSE
        LOG "Invalid code_challenge_method: " + method
        RETURN false
    END IF
END FUNCTION

FUNCTION base64URLEncode(data)
    // Encode data in base64url format (RFC 4648)
    // This is URL-safe base64 encoding:
    // - Replace '+' with '-'
    // - Replace '/' with '_'
    // - Remove '=' padding

    base64 = base64Encode(data)
    base64url = base64.replace('+', '-').replace('/', '_').replace('=', '')

    RETURN base64url
END FUNCTION
```

7. Token Blacklist Management

pseudocode

```
FUNCTION addTokenToBlacklist(token_jti, expiration_timestamp)
    // Add token identifier to blacklist
    // Store with expiration time for automatic cleanup

    LOG "Adding token to blacklist: " + token_jti

    // Add to blacklist storage
    revokedTokens.add(token_jti)

    // Optional: Schedule automatic removal after expiration
    // This prevents blacklist from growing indefinitely
    scheduleTokenCleanup(token_jti, expiration_timestamp)

    LOG "âœ“ Token blacklisted until: " + expiration_timestamp
END FUNCTION

FUNCTION isTokenRevoked(token_jti)
    // Check if token identifier exists in blacklist
    RETURN revokedTokens.has(token_jti)
END FUNCTION

FUNCTION scheduleTokenCleanup(token_jti, expiration_timestamp)
    // Calculate delay until token expires
    current_time = currentTimestamp()
    delay = expiration_timestamp - current_time

    IF delay > 0 THEN
        // Schedule removal after expiration
        scheduleTask(delay, FUNCTION()
            revokedTokens.delete(token_jti)
            LOG "âœ“ Expired token removed from blacklist: " + token_jti
        END FUNCTION)
    ELSE
        // Already expired, remove immediately
        revokedTokens.delete(token_jti)
    END IF
END FUNCTION

FUNCTION cleanupExpiredTokens()
    // Periodic cleanup task for token blacklist
    // Run this on a schedule (e.g., every hour)

    LOG "Running blacklist cleanup"
    current_time = currentTimestamp()
    removed_count = 0

    FOR EACH token_jti IN revokedTokens DO
        // Decode token to get expiration
        // In production, store expiration with blacklist entry
        // to avoid needing to decode

        TRY
            decoded = decodeJWT(getTokenByJTI(token_jti))

            IF decoded.exp < current_time THEN
                revokedTokens.delete(token_jti)
                removed_count = removed_count + 1
            END IF
        CATCH error
            // Token no longer valid, remove from blacklist
            revokedTokens.delete(token_jti)
            removed_count = removed_count + 1
        END TRY
    END FOR

    LOG "âœ" Cleanup complete: " + removed_count + " expired tokens removed"
END FUNCTION
```

---

8. Rate Limiting

pseudocode

```
// Rate limit storage
GLOBAL rateLimitStore = Map()  // Map<client_key, RateLimitRecord>

STRUCTURE RateLimitRecord:
    count: integer
    reset_time: timestamp
END STRUCTURE

FUNCTION rateLimitMiddleware(max_requests)
    RETURN FUNCTION(request, response, next)
        // Identify client (by IP address or client_id)
        client_key = getClientIdentifier(request)

        // Get current rate limit record
        record = rateLimitStore.get(client_key)
        current_time = currentTimestamp()

        IF NOT record OR current_time >= record.reset_time THEN
            // Start new window
            record = RateLimitRecord {
                count: 1,
                reset_time: current_time + RATE_LIMIT_WINDOW_MS
            }
            rateLimitStore.set(client_key, record)

            // Add rate limit headers to response
            response.setHeader("X-RateLimit-Limit", max_requests)
            response.setHeader("X-RateLimit-Remaining", max_requests - 1)
            response.setHeader("X-RateLimit-Reset", record.reset_time)

            RETURN next()  // Allow request
        ELSE
            // Increment counter
            record.count = record.count + 1

            IF record.count > max_requests THEN
                // Rate limit exceeded
                LOG "âš ï¸     Rate limit exceeded for: " + client_key

                response.setHeader("X-RateLimit-Limit", max_requests)
                response.setHeader("X-RateLimit-Remaining", 0)
                response.setHeader("X-RateLimit-Reset", record.reset_time)
                response.setHeader("Retry-After", record.reset_time - current_time)
            RETURN response(429, {
                error: "too_many_requests",
                error_description: "Rate limit exceeded"
            })
        ELSE
            // Update record and allow request
            rateLimitStore.set(client_key, record)

            response.setHeader("X-RateLimit-Limit", max_requests)
            response.setHeader("X-RateLimit-Remaining", max_requests - record.count)
            response.setHeader("X-RateLimit-Reset", record.reset_time)

            RETURN next()  // Allow request
        END IF
      END IF
    END FUNCTION
END FUNCTION

FUNCTION getClientIdentifier(request)
    // Identify client by client_id if available, otherwise IP address

    // Try to extract client_id from request
    IF request.body.client_id THEN
        RETURN "client:" + request.body.client_id
    ELSE IF request.query.client_id THEN
        RETURN "client:" + request.query.client_id
    ELSE
        // Fallback to IP address
        ip_address = request.headers["x-forwarded-for"] OR request.connection.remoteAddress
        RETURN "ip:" + ip_address
    END IF
END FUNCTION
```

---

9. Authentication Middleware

```
pseudocode
```

```
FUNCTION authenticationMiddleware(request, response, next)
    LOG "=== AUTHENTICATION CHECK ==="

    // Extract Authorization header
    authHeader = request.headers["authorization"]

    IF NOT authHeader THEN
        LOG "âœ— Missing Authorization header"
        RETURN response(401, {
            error: "Unauthorized",
            message: "Missing Authorization header"
        })
    END IF

    // Extract token from "Bearer <token>" format
    IF NOT authHeader.startsWith("Bearer ") THEN
        LOG "âœ— Invalid Authorization format"
        RETURN response(401, {
            error: "Unauthorized",
            message: "Invalid Authorization header format"
        })
    END IF

    token = authHeader.substring(7)  // Remove "Bearer "

    // Validate token
    payload = validateToken(token)

    IF NOT payload THEN
        LOG "âœ— Token validation failed"
        RETURN response(401, {
            error: "Unauthorized",
            message: "Invalid or expired token"
        })
    END IF

    // Token is valid - attach user info to request
    request.user = {
        user_id: payload.sub,
        client_id: payload.client_id,
        scope: payload.scope
    }
    LOG "âœ“ Authentication successful"
    LOG "User: " + payload.sub
    LOG "Client: " + payload.client_id

    // Proceed to route handler
    RETURN next()
END FUNCTION
```

---

10. MCP Protocol Handlers

10.1 Main MCP Endpoint

pseudocode

```
ROUTE POST "/mcp"
  APPLY authenticationMiddleware  // Require authentication
  APPLY rateLimitMiddleware(RATE_LIMIT_MAX_REQUESTS)

  FUNCTION handleMCPRequest(request, response)
    LOG "=== INCOMING MCP REQUEST ==="
    LOG "Method: " + request.body.method
    LOG "Request ID: " + request.body.id

    // Extract JSON-RPC parameters
    jsonrpc = request.body.jsonrpc
    method = request.body.method
    params = request.body.params
    request_id = request.body.id

    // Validate JSON-RPC version
    IF jsonrpc != "2.0" THEN
      RETURN response(400, {
        jsonrpc: "2.0",
        id: request_id,
        error: {
          code: -32600,
          message: "Invalid JSON-RPC version"
        }
      })
    END IF

    TRY
      // Handle notifications (no response needed)
      IF NOT request_id AND method.startsWith("notifications/") THEN
        LOG "âœ" Notification received: " + method
        RETURN response(200)
      END IF

      // Route to appropriate handler
      IF method = "initialize" THEN
        result = handleInitialize(params)
      ELSE IF method = "tools/list" THEN
        result = handleToolsList(params)
      ELSE IF method = "tools/call" THEN
        result = handleToolsCall(params)
      ELSE IF method = "resources/list" THEN
        result = handleResourcesList(params)
        ELSE IF method = "resources/read" THEN
            result = handleResourcesRead(params)
        ELSE IF method = "prompts/list" THEN
            result = handlePromptsList(params)
        ELSE IF method = "prompts/get" THEN
            result = handlePromptsGet(params)
        ELSE
            // Unknown method
            RETURN response(400, {
                jsonrpc: "2.0",
                id: request_id,
                error: {
                    code: -32601,
                    message: "Method not found: " + method
                }
            })
        END IF

        // Return success response
        RETURN response(200, {
            jsonrpc: "2.0",
            id: request_id,
            result: result
        })

    CATCH error
        LOG "âœ— MCP request error: " + error.message
        RETURN response(500, {
            jsonrpc: "2.0",
            id: request_id,
            error: {
                code: -32603,
                message: "Internal error: " + error.message
            }
        })
    END TRY
  END FUNCTION
END ROUTE
```

10.2 Initialize Handler

pseudocode

```
FUNCTION handleInitialize(params)
  LOG "=== INITIALIZE ==="
  LOG "Client protocol version: " + params.protocolVersion
  LOG "Client capabilities: " + params.capabilities

  // Return server capabilities and version info
  RETURN {
    protocolVersion: MCP_PROTOCOL_VERSION,
    capabilities: {
      tools: {
        listChanged: false
      },
      resources: {
        subscribe: false,
        listChanged: false
      },
      prompts: {
        listChanged: false
      },
      logging: {}
    },
    serverInfo: {
      name: MCP_SERVER_NAME,
      version: MCP_SERVER_VERSION
    }
  }
END FUNCTION
```

10.3 Tools Handlers

pseudocode

```
FUNCTION handleToolsList(params)
  LOG "=== TOOLS/LIST ==="

  // Define available tools
  // Replace with actual tool definitions for your use case
  tools = [
    {
      name: "get_weather",
      description: "Get current weather for a location",
      inputSchema: {
        type: "object",
        properties: {
          location: {
            type: "string",
            description: "City name or coordinates"
          }
        },
        required: ["location"]
      }
    },
    {
      name: "send_email",
      description: "Send an email message",
      inputSchema: {
        type: "object",
        properties: {
          to: {
            type: "string",
            description: "Recipient email address"
          },
          subject: {
            type: "string",
            description: "Email subject"
          },
          body: {
            type: "string",
            description: "Email body content"
          }
        },
        required: ["to", "subject", "body"]
      }
    }
  ]
```

```
    LOG "Returning " + tools.length + " tools"

    RETURN {
        tools: tools
    }
END FUNCTION

FUNCTION handleToolsCall(params)
    LOG "=== TOOLS/CALL ==="

    tool_name = params.name
    tool_args = params.arguments

    LOG "Executing tool: " + tool_name
    LOG "Arguments: " + tool_args

    // Route to appropriate tool implementation
    IF tool_name = "get_weather" THEN
        result = executeGetWeather(tool_args)
    ELSE IF tool_name = "send_email" THEN
        result = executeSendEmail(tool_args)
    ELSE
        THROW Error("Unknown tool: " + tool_name)
    END IF

    LOG "âœ“ Tool execution complete"

    // Return result in MCP format
    RETURN {
        content: [
            {
                type: "text",
                text: result
            }
        ]
    }
END FUNCTION

// Example tool implementations
FUNCTION executeGetWeather(args)
    location = args.location

    // Call weather API or service
    // This is a placeholder - replace with actual implementation
    LOG "Fetching weather for: " + location

    weather_data = callWeatherAPI(location)

    RETURN "Weather in " + location + ": " + weather_data.description +
           ", Temperature: " + weather_data.temperature + "Â°C"
END FUNCTION

FUNCTION executeSendEmail(args)
    to = args.to
    subject = args.subject
    body = args.body

    // Call email service
    // This is a placeholder - replace with actual implementation
    LOG "Sending email to: " + to

    success = callEmailService(to, subject, body)

    IF success THEN
        RETURN "Email sent successfully to " + to
    ELSE
        THROW Error("Failed to send email")
    END IF
END FUNCTION
```

10.4 Resources Handlers (Optional)

```
pseudocode
```

```
FUNCTION handleResourcesList(params)
    LOG "=== RESOURCES/LIST ==="

    // Define available resources
    // Resources are read-only data sources
    resources = [
        {
            uri: "file:///config/settings.json",
            name: "Server Configuration",
            description: "Current server configuration settings",
            mimeType: "application/json"
        },
        {
            uri: "db://customers",
            name: "Customer Database",
            description: "Customer records",
            mimeType: "application/json"
        }
    ]

    LOG "Returning " + resources.length + " resources"

    RETURN {
        resources: resources
    }
END FUNCTION

FUNCTION handleResourcesRead(params)
    LOG "=== RESOURCES/READ ==="

    resource_uri = params.uri

    LOG "Reading resource: " + resource_uri

    // Route to appropriate resource handler
    IF resource_uri.startsWith("file://") THEN
        content = readFileResource(resource_uri)
    ELSE IF resource_uri.startsWith("db://") THEN
        content = readDatabaseResource(resource_uri)
    ELSE
        THROW Error("Unknown resource URI: " + resource_uri)
    END IF
    LOG "âœ“ Resource read complete"

    // Return resource content
    RETURN {
        contents: [
            {
                uri: resource_uri,
                mimeType: "application/json",
                text: content
            }
        ]
    }
END FUNCTION
```

10.5 Prompts Handlers (Optional)

pseudocode

```
FUNCTION handlePromptsList(params)
    LOG "=== PROMPTS/LIST ==="

    // Define available prompts
    // Prompts are templated workflows
    prompts = [
        {
            name: "code_review",
            description: "Review code for quality and security",
            arguments: [
                {
                    name: "code",
                    description: "Code to review",
                    required: true
                },
                {
                    name: "language",
                    description: "Programming language",
                    required: false
                }
            ]
        }
    ]

    LOG "Returning " + prompts.length + " prompts"

    RETURN {
        prompts: prompts
    }
END FUNCTION

FUNCTION handlePromptsGet(params)
    LOG "=== PROMPTS/GET ==="

    prompt_name = params.name
    prompt_args = params.arguments

    LOG "Getting prompt: " + prompt_name

    // Route to appropriate prompt template
    IF prompt_name = "code_review" THEN
        messages = generateCodeReviewPrompt(prompt_args)
    ELSE
```

```
        THROW Error("Unknown prompt: " + prompt_name)
    END IF

    LOG "âœ“ Prompt generated"

    RETURN {
        messages: messages
    }
END FUNCTION

FUNCTION generateCodeReviewPrompt(args)
    code = args.code
    language = args.language OR "unknown"

    // Generate prompt messages
    RETURN [
        {
            role: "user",
            content: {
                type: "text",
                text: "Please review the following " + language + " code for quality, security, and best practices:\n\n" + code
            }
        }
    ]
END FUNCTION
```

---

11. Metadata Endpoints

pseudocode

```
ROUTE GET "/.well-known/oauth-authorization-server"
  FUNCTION handleAuthorizationServerMetadata(request, response)
    // RFC 8414 - Authorization Server Metadata
    LOG "=== AUTHORIZATION SERVER METADATA REQUEST ==="

    metadata = {
      issuer: JWT_ISSUER,
      authorization_endpoint: JWT_ISSUER + "/oauth/authorize",
      token_endpoint: JWT_ISSUER + "/oauth/token",
      revocation_endpoint: JWT_ISSUER + "/oauth/revoke",
      registration_endpoint: JWT_ISSUER + "/register",
      response_types_supported: ["code"],
      grant_types_supported: ["authorization_code", "refresh_token"],
      token_endpoint_auth_methods_supported: ["client_secret_post"],
      code_challenge_methods_supported: ["S256", "plain"],
      scopes_supported: ["openid", "email", "profile"],
      service_documentation: "https://github.com/your-org/mcp-server"
    }

    RETURN response(200, metadata)
  END FUNCTION
END ROUTE

ROUTE GET "/.well-known/oauth-protected-resource"
  FUNCTION handleProtectedResourceMetadata(request, response)
    // RFC 8414 - Protected Resource Metadata
    LOG "=== PROTECTED RESOURCE METADATA REQUEST ==="

    metadata = {
      resource: JWT_ISSUER,
      authorization_servers: [JWT_ISSUER],
      scopes_supported: ["openid", "email", "profile"],
      bearer_methods_supported: ["header"],
      resource_documentation: "https://github.com/your-org/mcp-server"
    }

    RETURN response(200, metadata)
  END FUNCTION
END ROUTE
```

---

12. Health Check Endpoint

pseudocode

```
ROUTE GET "/health"
  FUNCTION handleHealthCheck(request, response)
    // Basic health check endpoint
    // Does not require authentication

    health_status = {
      status: "healthy",
      timestamp: currentTimestamp(),
      version: MCP_SERVER_VERSION,
      uptime: getServerUptime()
    }

    RETURN response(200, health_status)
  END FUNCTION
END ROUTE
```

---

13. Audit Logging

pseudocode

```
STRUCTURE AuditLogEntry:
    timestamp: timestamp
    event_type: string
    client_id: string
    user_id: string
    method: string
    status: string
    ip_address: string
    details: object
END STRUCTURE

FUNCTION logAuditEvent(event_type, client_id, user_id, method, status, ip_address, details)
    // Create audit log entry
    entry = AuditLogEntry {
        timestamp: currentTimestamp(),
        event_type: event_type,
        client_id: client_id OR "unknown",
        user_id: user_id OR "unknown",
        method: method,
        status: status,
        ip_address: ip_address,
        details: details
    }

    // Write to audit log
    // This could be written to file, database, or logging service
    writeAuditLog(entry)

    LOG "[AUDIT] " + event_type + " | " + status + " | " + client_id
END FUNCTION

// Example audit logging in endpoints
FUNCTION handleTokenExchangeWithAudit(request, response)
    client_id = request.body.client_id
    ip_address = request.connection.remoteAddress

    TRY
        result = handleTokenExchange(request, response)

        // Log successful token exchange
        logAuditEvent(
            "TOKEN_EXCHANGE",
            client_id,
            null,
            "authorization_code",
            "SUCCESS",
            ip_address,
            { grant_type: request.body.grant_type }
        )

        RETURN result
    CATCH error
        // Log failed token exchange
        logAuditEvent(
            "TOKEN_EXCHANGE",
            client_id,
            null,
            "authorization_code",
            "FAILURE",
            ip_address,
            { error: error.message, grant_type: request.body.grant_type }
        )

        THROW error
    END TRY
END FUNCTION
```

Implementation Notes

Storage Considerations

This pseudocode uses in-memory data structures (`Map`, `Set`) for simplicity. In production:

1. **Client Registry** - Store registered clients in a persistent database
2. **Authorization Codes** - Use Redis or similar with automatic expiration
3. **Token Blacklist** - Use Redis with TTL for automatic cleanup
4. **Rate Limiting** - Use Redis for distributed rate limiting across servers

Security Considerations

1. **JWT_SECRET** - Must be cryptographically secure, minimum 256 bits
2. **HTTPS Required** - All endpoints must use TLS in production
3. **CORS** - Configure CORS headers appropriately for your deployment

4. **Input Validation** - Validate all user inputs before processing
5. **Error Messages** - Don't leak sensitive information in error responses

Performance Optimization

1. **Token Validation** - Cache decoded tokens briefly to reduce CPU usage
2. **Client Lookup** - Index client registry by client_id for fast lookups
3. **Rate Limiting** - Use efficient data structures (sliding window counters)
4. **Blacklist Cleanup** - Run periodic cleanup tasks during off-peak hours

Testing Recommendations

1. **OAuth Flow** - Test complete authorization code flow with PKCE
2. **Token Rotation** - Verify refresh token rotation prevents replay attacks
3. **Rate Limiting** - Verify rate limits are enforced correctly
4. **Token Revocation** - Test revocation immediately blocks access
5. **MCP Protocol** - Test all MCP methods with valid and invalid inputs

---

Conclusion

This pseudocode template provides a complete reference implementation for building a production-ready MCP server with OAuth 2.1 + PKCE authentication. It demonstrates:

- Complete OAuth 2.1 authorization code flow with PKCE
- JWT token management with rotation and revocation
- Rate limiting and audit logging
- Full MCP protocol support (tools, resources, prompts)
- Security best practices and error handling

Translate this pseudocode into your chosen programming language and framework, adapting the data structures and storage mechanisms to your specific infrastructure requirements.