MCP Server Implementation Guide for ServiceNow Integration

Part 2: Server Foundation - Core Infrastructure Setup

Document Series

This implementation guide is organized into five parts:

1. Part 1: Introduction and Overview
2. **Part 2: Server Foundation - Core Infrastructure Setup** (This Document)
3. Part 3: MCP Protocol Implementation and Tools
4. Part 4: OAuth 2.1 Implementation
5. Part 5: Appendices and Recommendations

Introduction

This document covers the foundational infrastructure required before implementing OAuth authentication or MCP protocol handlers. Think of this as building the "concrete slab and framing" of a house - everything else depends on a solid foundation.

What You'll Build:

- HTTP server with proper lifecycle management
- Middleware stack for request processing
- Configuration and secret management
- Data storage for clients, codes, and tokens

Why Foundation First:

The foundation components must be in place before OAuth and MCP implementations because:

- Middleware processes requests before they reach route handlers
- Configuration must be validated before any services initialize
- Storage must be ready before OAuth client registration
- Proper error handling catches issues from all layers

By completing this foundation, you'll have a production-ready server infrastructure that can support OAuth and MCP protocol implementations.

Section Organization:

The foundation is organized in the order you should implement it:

1. **Basic HTTP Server Setup** - Core server initialization and lifecycle
2. **Core Middleware Stack** - Request processing pipeline
3. **Configuration Management** - Environment variables and secrets
4. **Data Storage Initialization** - Persistent storage for OAuth data

This order ensures each component builds upon the previous ones.

Server Foundation - Core Infrastructure Setup

Overview

Before implementing OAuth endpoints or MCP protocol handlers, you must establish the foundational server infrastructure. This foundation provides the essential capabilities that all subsequent components will build upon.

1. Basic HTTP Server Setup

Server Initialization

Your MCP server requires an HTTP server capable of handling POST requests with JSON payloads. The server must:

- Listen on a configurable port (default: 3000 for local development)
- Support HTTPS in production environments (required for ServiceNow integration)
- Handle graceful startup and shutdown
- Support health check endpoints for monitoring

Port Configuration

Define your server's listening port through environment variables to enable environment-specific configurations:

- Development: Typically port 3000 or 8080
- Production: Port 443 (HTTPS) behind reverse proxy, or custom port with TLS termination
- ServiceNow Requirement: Must be accessible via HTTPS URL

Request/Response Handling

The server must handle:

- HTTP POST requests (primary method for MCP and OAuth endpoints)
- HTTP GET requests (OAuth authorization, health checks, metadata endpoints)
- HTTP OPTIONS requests (CORS preflight)

Server Lifecycle Management

Implement proper startup and shutdown procedures:

- **Startup**: Initialize connections (database, Redis), load configuration, verify dependencies
- **Shutdown**: Close connections gracefully, finish in-flight requests, save state if necessary
- **Signal Handling**: Respond to SIGTERM and SIGINT for container orchestration compatibility

Implementation Approach

Now that we've outlined the requirements, let's see how these elements come together in a working server foundation. The following examples demonstrate initializing an HTTP server with proper configuration, lifecycle management, and graceful shutdown handling.

The key aspects to notice:

- Port configuration loaded from environment variables with sensible defaults
- Server listens only after all initialization is complete
- Graceful shutdown ensures in-flight requests complete before server stops
- Error handling during startup prevents the server from running in an invalid state

Pseudocode:

```
FUNCTION initialize_server():
    // Load configuration
    port = GET_ENV('PORT') OR 3000

    // Create HTTP server
    server = CREATE_HTTP_SERVER()

    // Configure server to handle requests
    server.ON_REQUEST(handle_request)

    // Start listening
    TRY:
        server.LISTEN(port)
        LOG("Server listening on port " + port)
    CATCH error:
        LOG_ERROR("Failed to start server: " + error)
        EXIT(1)
    END TRY

    // Setup graceful shutdown
    ON_SIGNAL('SIGTERM', graceful_shutdown)
    ON_SIGNAL('SIGINT', graceful_shutdown)
END FUNCTION

FUNCTION graceful_shutdown():
    LOG("Shutdown signal received, closing server gracefully...")

    // Stop accepting new connections
    server.CLOSE()

    // Close external connections (database, Redis, etc.)
    CLOSE_DATABASE_CONNECTIONS()
    CLOSE_REDIS_CONNECTION()

    // Save any pending state if necessary
    SAVE_CLIENT_REGISTRATIONS()

    LOG("Server shutdown complete")
    EXIT(0)
END FUNCTION

FUNCTION handle_request(request, response):
    // Basic request handling - will be expanded with routes/middleware
```

```
    IF request.path == "/health":
        response.STATUS(200)
        response.JSON({ status: "healthy" })
    ELSE:
        response.STATUS(404)
        response.JSON({ error: "Not found" })
    END IF
END FUNCTION
```

JavaScript:

```javascript
```

```
const express = require('express');
const app = express();

// Load configuration
const PORT = process.env.PORT || 3000;

// Create HTTP server instance
let server;

// Initialize and start server
function initializeServer() {
  try {
    // Start listening
    server = app.listen(PORT, () => {
      console.log(`[SERVER] Listening on port ${PORT}`);
      console.log(`[SERVER] Environment: ${process.env.NODE_ENV || 'development'}`);
    });

    // Handle server errors
    server.on('error', (error) => {
      if (error.code === 'EADDRINUSE') {
        console.error(`[ERROR] Port ${PORT} is already in use`);
      } else {
        console.error('[ERROR] Server error:', error);
      }
      process.exit(1);
    });

  } catch (error) {
    console.error('[ERROR] Failed to start server:', error);
    process.exit(1);
  }
}

// Graceful shutdown handler
function gracefulShutdown(signal) {
  console.log(`[SHUTDOWN] ${signal} received, closing server gracefully...`);

  // Stop accepting new connections
  server.close(() => {
    console.log('[SHUTDOWN] HTTP server closed');

    // Close external connections here
```

```
    // e.g., database.close(), redisClient.quit()

    // Save any pending state
    // e.g., saveClientRegistrations()

    console.log('[SHUTDOWN] Cleanup complete');
    process.exit(0);
  });

  // Force shutdown after 10 seconds if graceful shutdown hangs
  setTimeout(() => {
    console.error('[SHUTDOWN] Forcing shutdown after timeout');
    process.exit(1);
  }, 10000);
}

// Setup signal handlers
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));
process.on('SIGINT', () => gracefulShutdown('SIGINT'));

// Basic health check endpoint (will be expanded)
app.get('/health', (req, res) => {
  res.status(200).json({
    status: 'healthy',
    timestamp: new Date().toISOString()
  });
});

// Initialize server
initializeServer();
```

TypeScript:

```
typescript
```

```
import express, { Application, Request, Response } from 'express';
import { Server } from 'http';

// Load configuration
const PORT: number = parseInt(process.env.PORT || '3000', 10);

// Create Express application
const app: Application = express();

// Server instance
let server: Server;

// Initialize and start server
function initializeServer(): void {
  try {
    // Start listening
    server = app.listen(PORT, () => {
      console.log(`[SERVER] Listening on port ${PORT}`);
      console.log(`[SERVER] Environment: ${process.env.NODE_ENV || 'development'}`);
    });

    // Handle server errors
    server.on('error', (error: NodeJS.ErrnoException) => {
      if (error.code === 'EADDRINUSE') {
        console.error(`[ERROR] Port ${PORT} is already in use`);
      } else {
        console.error('[ERROR] Server error:', error);
      }
      process.exit(1);
    });

  } catch (error) {
    console.error('[ERROR] Failed to start server:', error);
    process.exit(1);
  }
}

// Graceful shutdown handler
function gracefulShutdown(signal: string): void {
  console.log(`[SHUTDOWN] ${signal} received, closing server gracefully...`);

  // Stop accepting new connections
  server.close(() => {
```

```
    console.log('[SHUTDOWN] HTTP server closed');

    // Close external connections here
    // e.g., database.close(), redisClient.quit()

    // Save any pending state
    // e.g., saveClientRegistrations()

    console.log('[SHUTDOWN] Cleanup complete');
    process.exit(0);
  });

  // Force shutdown after 10 seconds if graceful shutdown hangs
  setTimeout(() => {
    console.error('[SHUTDOWN] Forcing shutdown after timeout');
    process.exit(1);
  }, 10000);
}

// Setup signal handlers
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));
process.on('SIGINT', () => gracefulShutdown('SIGINT'));

// Basic health check endpoint (will be expanded)
app.get('/health', (req: Request, res: Response) => {
  res.status(200).json({
    status: 'healthy',
    timestamp: new Date().toISOString()
  });
});

// Initialize server
initializeServer();
```

---

2. Core Middleware Stack

Middleware provides cross-cutting functionality that applies to all or most endpoints. Configure middleware in the correct order for proper request processing.

Body Parsing

Configure two body parsers to handle different content types:

1. **JSON Parser**: For MCP protocol messages and most OAuth responses
    - Maximum payload size: Consider limiting to prevent DOS attacks (e.g., 1MB)
    - Strict parsing: Reject malformed JSON
2. **URL-Encoded Parser**: For OAuth token endpoint requests
    - Required: OAuth 2.1 token endpoint accepts `application/x-www-form-urlencoded`
    - Extended parsing: Enable to support nested objects if needed

Implementation Approach

Body parsing middleware must be configured early in your middleware stack, before any route handlers that need to access request data. Two parsers are required: one for JSON payloads (MCP protocol messages) and one for URL-encoded data (OAuth token endpoint). Configure size limits to prevent denial-of-service attacks through oversized payloads.

Key considerations:

- JSON parser handles MCP protocol messages (initialize, tools/list, tools/call)
- URL-encoded parser handles OAuth token requests (required by OAuth 2.1 specification)
- Set reasonable size limits (1MB for JSON, 100KB for URL-encoded is typical)
- Apply parsers globally so all routes benefit

Pseudocode:

```
FUNCTION configure_body_parsing(app):
  // Configure JSON parser for MCP protocol messages
  json_parser_options = {
    limit: '1mb',          // Maximum payload size
    strict: true           // Reject malformed JSON
  }
  app.USE_MIDDLEWARE(json_parser(json_parser_options))

  // Configure URL-encoded parser for OAuth token endpoint
  urlencoded_parser_options = {
    limit: '100kb',        // Smaller limit for form data
    extended: true         // Support nested objects
  }
  app.USE_MIDDLEWARE(urlencoded_parser(urlencoded_parser_options))

  LOG("Body parsers configured: JSON (1MB limit), URL-encoded (100KB limit)")
END FUNCTION
```

JavaScript:

javascript

```javascript
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

// Configure JSON parser for MCP protocol messages
app.use(bodyParser.json({
  limit: '1mb',    // Maximum payload size
  strict: true     // Reject malformed JSON
}));

// Configure URL-encoded parser for OAuth token endpoint
app.use(bodyParser.urlencoded({
  limit: '100kb',  // Smaller limit for form data
  extended: true   // Support nested objects
}));

console.log('[MIDDLEWARE] Body parsers configured: JSON (1MB limit), URL-encoded (100KB limit)');
```

TypeScript:

typescript

```
import express, { Application } from 'express';
import bodyParser from 'body-parser';

const app: Application = express();

// Configure JSON parser for MCP protocol messages
app.use(bodyParser.json({
  limit: '1mb',   // Maximum payload size
  strict: true    // Reject malformed JSON
}));

// Configure URL-encoded parser for OAuth token endpoint
app.use(bodyParser.urlencoded({
  limit: '100kb',  // Smaller limit for form data
  extended: true   // Support nested objects
}));

console.log('[MIDDLEWARE] Body parsers configured: JSON (1MB limit), URL-encoded (100KB limit)');
```

CORS Configuration

ServiceNow instances may make cross-origin requests to your MCP server. Configure CORS to allow these requests while maintaining security.

Recommended Headers:

- `Access-Control-Allow-Origin`: Your ServiceNow instance URL (e.g., `https://instance.service-now.com`)
- `Access-Control-Allow-Methods`: `GET, POST, OPTIONS`
- `Access-Control-Allow-Headers`: `Content-Type, Authorization`
- `Access-Control-Allow-Credentials`: `true`

Platform Considerations:

- Some platforms (Google Cloud Run, AWS API Gateway, Azure API Management) handle CORS through platform configuration
- Verify your deployment platform's CORS handling before implementing custom middleware
- If your platform doesn't handle CORS automatically, implement CORS middleware as shown below

Security Considerations:

- Never use wildcard (`*`) for origin when credentials are enabled
- Explicitly list your ServiceNow instance(s)

- Consider maintaining a configuration list for multiple instances

Implementation Approach

CORS configuration enables ServiceNow to make cross-origin requests to your MCP server. While some deployment platforms handle CORS automatically (Google Cloud Run, AWS API Gateway), implementing CORS middleware ensures compatibility across all platforms. The middleware must execute early in the request pipeline (before route handlers) to properly handle preflight OPTIONS requests. Notice how we explicitly whitelist the ServiceNow instance URL rather than using wildcards, and enable credentials to support OAuth token transmission.

Key considerations:

- Check if your deployment platform handles CORS automatically before implementing
- Origin must match your ServiceNow instance URL exactly (no wildcards when credentials enabled)
- Credentials flag must be true to allow Authorization headers
- Preflight OPTIONS requests must return 200 OK
- CORS middleware should be among the first middleware configured if needed

Pseudocode:

```
FUNCTION configure_cors(app):
    // Load ServiceNow instance URL from configuration
    servicenow_url = GET_ENV('SERVICENOW_INSTANCE')

    IF servicenow_url IS NULL:
        LOG_ERROR("SERVICENOW_INSTANCE not configured")
        FAIL_STARTUP("Missing required configuration")
    END IF

    // Configure CORS options
    cors_options = {
        origin: servicenow_url,                    // Explicit origin (no wildcards)
        methods: ['GET', 'POST', 'OPTIONS'],       // Allowed HTTP methods
        allowedHeaders: ['Content-Type', 'Authorization'],   // Allowed request headers
        credentials: true                          // Allow cookies and Authorization header
    }

    // Apply CORS middleware
    app.USE_MIDDLEWARE(cors(cors_options))

    LOG("CORS configured for ServiceNow instance: " + servicenow_url)
END FUNCTION
```

JavaScript:

```
javascript
```

```
const express = require('express');
const cors = require('cors');

const app = express();

// Load ServiceNow instance URL from configuration
const SERVICENOW_INSTANCE = process.env.SERVICENOW_INSTANCE;

if (!SERVICENOW_INSTANCE) {
  console.error('[ERROR] SERVICENOW_INSTANCE not configured');
  process.exit(1);
}

// Configure CORS options
const corsOptions = {
  origin: SERVICENOW_INSTANCE,                    // Explicit origin (no wildcards)
  methods: ['GET', 'POST', 'OPTIONS'],            // Allowed HTTP methods
  allowedHeaders: ['Content-Type', 'Authorization'], // Allowed request headers
  credentials: true                               // Allow cookies and Authorization header
};

// Apply CORS middleware
app.use(cors(corsOptions));

console.log(`[MIDDLEWARE] CORS configured for ServiceNow instance: ${SERVICENOW_INSTANCE}`);
```

TypeScript:

```
typescript
```

```
import express, { Application } from 'express';
import cors, { CorsOptions } from 'cors';

const app: Application = express();

// Load ServiceNow instance URL from configuration
const SERVICENOW_INSTANCE: string | undefined = process.env.SERVICENOW_INSTANCE;

if (!SERVICENOW_INSTANCE) {
  console.error('[ERROR] SERVICENOW_INSTANCE not configured');
  process.exit(1);
}

// Configure CORS options
const corsOptions: CorsOptions = {
  origin: SERVICENOW_INSTANCE,                    // Explicit origin (no wildcards)
  methods: ['GET', 'POST', 'OPTIONS'],            // Allowed HTTP methods
  allowedHeaders: ['Content-Type', 'Authorization'], // Allowed request headers
  credentials: true                               // Allow cookies and Authorization header
};

// Apply CORS middleware
app.use(cors(corsOptions));

console.log(`[MIDDLEWARE] CORS configured for ServiceNow instance: ${SERVICENOW_INSTANCE}`);
```

Request Logging

Implement request logging middleware for debugging, monitoring, and audit purposes:

What to Log:

- Timestamp (ISO 8601 format)
- HTTP method and path
- Request ID (generated UUID for request tracking)
- Source IP address
- Response status code
- Response time (milliseconds)

What NOT to Log:

- Full Authorization headers (log only first 10-15 characters for debugging)

- Request/response bodies containing sensitive data
- Complete tokens or secrets

Recommended Log Levels:

- Development: Verbose (all requests, headers, body previews)
- Production: Minimal (timestamp, method, path, status, response time)

Implementation Approach

Request logging middleware captures essential information about every request for debugging, monitoring, and audit purposes. Position this middleware early in the stack (after CORS, before route handlers) to ensure all requests are logged, including those that result in errors. The middleware should log minimal information in production while providing verbose details in development.

Key considerations:

- Log enough information for debugging without exposing sensitive data
- Generate unique request IDs for tracing requests through the system
- Never log full Authorization headers or request bodies with secrets
- Use structured logging format for easier parsing and analysis
- Consider different log levels for development vs production

Pseudocode:

```
FUNCTION configure_request_logging(app):
    environment = GET_ENV('NODE_ENV') OR 'development'

    // Create logging middleware
    logging_middleware = FUNCTION(request, response, next):
        request_id = GENERATE_UUID()
        start_time = CURRENT_TIME()

        // Attach request ID for tracing
        request.id = request_id

        // Log request received
        IF environment == 'development':
            LOG("Request received: " + request.method + " " + request.path)
            LOG("  Request ID: " + request_id)
            LOG("  Headers: " + SANITIZE_HEADERS(request.headers))
        ELSE:
            LOG(request.method + " " + request.path + " | ID: " + request_id)
        END IF

        // Log response when finished
        response.ON_FINISH(FUNCTION():
            duration = CURRENT_TIME() - start_time
            status = response.statusCode

            IF environment == 'development':
                LOG("Response sent: " + status + " | Duration: " + duration + "ms")
            ELSE:
                LOG(request.method + " " + request.path + " | " + status + " | " + duration + "ms")
            END IF
        END FUNCTION)

        next()
    END FUNCTION

    app.USE_MIDDLEWARE(logging_middleware)
    LOG("Request logging middleware configured")
END FUNCTION

FUNCTION SANITIZE_HEADERS(headers):
    sanitized = COPY(headers)

    // Redact sensitive headers
```

```
  IF sanitized.authorization EXISTS:
    sanitized.authorization = sanitized.authorization.SUBSTRING(0, 15) + "..."
  END IF

  RETURN sanitized
END FUNCTION
```

JavaScript:

```javascript
```

```
const express = require('express');
const { v4: uuidv4 } = require('uuid');

const app = express();
const NODE_ENV = process.env.NODE_ENV || 'development';

// Request logging middleware
app.use((req, res, next) => {
  const requestId = uuidv4();
  const startTime = Date.now();

  // Attach request ID for tracing
  req.id = requestId;

  // Log request received
  if (NODE_ENV === 'development') {
    console.log(`[REQUEST] ${req.method} ${req.path}`);
    console.log(`  Request ID: ${requestId}`);
    console.log(`  IP: ${req.ip}`);

    // Sanitize and log headers
    const sanitizedHeaders = { ...req.headers };
    if (sanitizedHeaders.authorization) {
      sanitizedHeaders.authorization = sanitizedHeaders.authorization.substring(0, 15) + '...';
    }
    console.log(`  Headers:`, sanitizedHeaders);
  } else {
    console.log(`[REQUEST] ${req.method} ${req.path} | ID: ${requestId} | IP: ${req.ip}`);
  }

  // Log response when finished
  res.on('finish', () => {
    const duration = Date.now() - startTime;
    const status = res.statusCode;

    if (NODE_ENV === 'development') {
      console.log(`[RESPONSE] ${status} | Duration: ${duration}ms | Request ID: ${requestId}`);
    } else {
      console.log(`[RESPONSE] ${req.method} ${req.path} | ${status} | ${duration}ms | ID: ${requestId}`);
    }
  });

  next();
```

```
});

console.log('[MIDDLEWARE] Request logging configured');
```

TypeScript:

```typescript
```

```typescript
import express, { Application, Request, Response, NextFunction } from 'express';
import { v4 as uuidv4 } from 'uuid';

const app: Application = express();
const NODE_ENV: string = process.env.NODE_ENV || 'development';

// Extend Request type to include id property
declare global {
  namespace Express {
    interface Request {
      id?: string;
    }
  }
}

// Request logging middleware
app.use((req: Request, res: Response, next: NextFunction) => {
  const requestId: string = uuidv4();
  const startTime: number = Date.now();

  // Attach request ID for tracing
  req.id = requestId;

  // Log request received
  if (NODE_ENV === 'development') {
    console.log(`[REQUEST] ${req.method} ${req.path}`);
    console.log(`  Request ID: ${requestId}`);
    console.log(`  IP: ${req.ip}`);

    // Sanitize and log headers
    const sanitizedHeaders = { ...req.headers };
    if (sanitizedHeaders.authorization) {
      sanitizedHeaders.authorization = sanitizedHeaders.authorization.substring(0, 15) + '...';
    }
    console.log(`  Headers:`, sanitizedHeaders);
  } else {
    console.log(`[REQUEST] ${req.method} ${req.path} | ID: ${requestId} | IP: ${req.ip}`);
  }

  // Log response when finished
  res.on('finish', () => {
    const duration: number = Date.now() - startTime;
    const status: number = res.statusCode;
```

```
    if (NODE_ENV === 'development') {
      console.log(`[RESPONSE] ${status} | Duration: ${duration}ms | Request ID: ${requestId}`);
    } else {
      console.log(`[RESPONSE] ${req.method} ${req.path} | ${status} | ${duration}ms | ID: ${requestId}`);
    }
  });

  next();
});

console.log('[MIDDLEWARE] Request logging configured');
```

Error Handling

Implement global error handling middleware to catch and properly format errors:

Error Response Structure:

- Use JSON-RPC 2.0 error format for MCP endpoint errors
- Use OAuth 2.1 error format for authentication endpoint errors
- Use standard HTTP error responses for other endpoints

Error Categories:

- 400 Bad Request: Invalid request format or parameters
- 401 Unauthorized: Missing or invalid authentication
- 403 Forbidden: Valid auth but insufficient permissions
- 405 Method Not Allowed: Unsupported HTTP method
- 429 Too Many Requests: Rate limit exceeded
- 500 Internal Server Error: Unexpected server errors

Security Consideration:

- Never expose internal error details (stack traces, file paths) in production
- Log detailed errors server-side
- Return generic error messages to clients

Implementation Approach

Global error handling middleware catches all errors that occur during request processing and formats them appropriately based on the endpoint type. This middleware must be positioned last in your middleware stack

(after all route handlers) to catch errors from any previous middleware or route. Different endpoints require different error formats: JSON-RPC 2.0 for MCP, OAuth 2.1 format for authentication endpoints, and standard HTTP errors for others.

Key considerations:

- Position error handler as the last middleware (after all routes)
- Distinguish between MCP, OAuth, and general HTTP errors
- Never expose internal error details (stack traces, file paths) in production
- Log detailed errors server-side for debugging
- Return appropriate HTTP status codes with error responses

Pseudocode:

```
FUNCTION configure_error_handling(app):
  environment = GET_ENV('NODE_ENV') OR 'development'

  // Global error handling middleware (must be last)
  error_handler = FUNCTION(error, request, response, next):
    // Log error details server-side
    LOG_ERROR("Error occurred:")
    LOG_ERROR("  Path: " + request.path)
    LOG_ERROR("  Method: " + request.method)
    LOG_ERROR("  Request ID: " + request.id)
    LOG_ERROR("  Error: " + error.message)

    IF environment == 'development':
      LOG_ERROR("  Stack trace: " + error.stack)
    END IF

    // Determine error format based on endpoint
    IF request.path == '/mcp':
      // JSON-RPC 2.0 error format for MCP endpoint
      response.STATUS(200)  // JSON-RPC uses 200 with error object
      response.JSON({
        jsonrpc: "2.0",
        error: {
          code: -32603,
          message: "Internal server error",
          data: environment == 'development' ? error.message : undefined
        },
        id: request.body.id OR null
      })
    ELSE IF request.path.STARTS_WITH('/oauth/'):
      // OAuth 2.1 error format
      response.STATUS(error.statusCode OR 500)
      response.JSON({
        error: error.oauthError OR "server_error",
        error_description: error.message OR "An internal error occurred"
      })
    ELSE:
      // Standard HTTP error format
      response.STATUS(error.statusCode OR 500)
      response.JSON({
        error: error.message OR "Internal server error",
        request_id: request.id,
        timestamp: CURRENT_TIMESTAMP()
```

```
        })
    END IF
END FUNCTION

// Apply error handler (must be after all routes)
app.USE_ERROR_HANDLER(error_handler)

LOG("Error handling middleware configured")
END FUNCTION
```

JavaScript:

```javascript
```

```
const express = require('express');

const app = express();
const NODE_ENV = process.env.NODE_ENV || 'development';

// Global error handling middleware (must be defined after all routes)
app.use((err, req, res, next) => {
  // Log error details server-side
  console.error('[ERROR] Error occurred:');
  console.error(`  Path: ${req.path}`);
  console.error(`  Method: ${req.method}`);
  console.error(`  Request ID: ${req.id}`);
  console.error(`  Error: ${err.message}`);

  if (NODE_ENV === 'development') {
    console.error(`  Stack trace: ${err.stack}`);
  }

  // Determine error format based on endpoint
  if (req.path === '/mcp') {
    // JSON-RPC 2.0 error format for MCP endpoint
    return res.status(200).json({
      jsonrpc: '2.0',
      error: {
        code: -32603,
        message: 'Internal server error',
        data: NODE_ENV === 'development' ? err.message : undefined
      },
      id: req.body?.id || null
    });
  } else if (req.path.startsWith('/oauth/')) {
    // OAuth 2.1 error format
    return res.status(err.statusCode || 500).json({
      error: err.oauthError || 'server_error',
      error_description: err.message || 'An internal error occurred'
    });
  } else {
    // Standard HTTP error format
    return res.status(err.statusCode || 500).json({
      error: err.message || 'Internal server error',
      request_id: req.id,
      timestamp: new Date().toISOString()
    });
```

```
  }
});

console.log('[MIDDLEWARE] Error handling configured');
```

TypeScript:

```
typescript
```

```typescript
import express, { Application, Request, Response, NextFunction } from 'express';

const app: Application = express();
const NODE_ENV: string = process.env.NODE_ENV || 'development';

// Define custom error type
interface CustomError extends Error {
  statusCode?: number;
  oauthError?: string;
}

// Global error handling middleware (must be defined after all routes)
app.use((err: CustomError, req: Request, res: Response, next: NextFunction) => {
  // Log error details server-side
  console.error('[ERROR] Error occurred:');
  console.error(`  Path: ${req.path}`);
  console.error(`  Method: ${req.method}`);
  console.error(`  Request ID: ${req.id}`);
  console.error(`  Error: ${err.message}`);

  if (NODE_ENV === 'development') {
    console.error(`  Stack trace: ${err.stack}`);
  }

  // Determine error format based on endpoint
  if (req.path === '/mcp') {
    // JSON-RPC 2.0 error format for MCP endpoint
    return res.status(200).json({
      jsonrpc: '2.0',
      error: {
        code: -32603,
        message: 'Internal server error',
        data: NODE_ENV === 'development' ? err.message : undefined
      },
      id: req.body?.id || null
    });
  } else if (req.path.startsWith('/oauth/')) {
    // OAuth 2.1 error format
    return res.status(err.statusCode || 500).json({
      error: err.oauthError || 'server_error',
      error_description: err.message || 'An internal error occurred'
    });
  } else {
```

```
    // Standard HTTP error format
    return res.status(err.statusCode || 500).json({
      error: err.message || 'Internal server error',
      request_id: req.id,
      timestamp: new Date().toISOString()
    });
  }
});

console.log('[MIDDLEWARE] Error handling configured');
```

---

3. Configuration Management

Centralize all configuration values to enable environment-specific deployments and maintain security.

Environment Variables

Use environment variables for all configuration that varies between environments:

Server Configuration:

- `PORT`: Server listening port
- `NODE_ENV`: Environment identifier (development, production)
- `SERVER_URL`: Public-facing server URL (e.g., `https://mcp.yourdomain.com`)

Authentication Configuration:

- `JWT_SECRET`: Secret key for signing JWT tokens (32+ bytes, cryptographically secure)
- `JWT_ISSUER`: Token issuer identifier (typically your server URL)
- `DCR_AUTH_TOKEN`: Authorization token for Dynamic Client Registration endpoint

Token Lifetime Configuration:

- `ACCESS_TOKEN_LIFETIME`: Access token expiration (seconds, recommended: 3600 = 1 hour)
- `REFRESH_TOKEN_LIFETIME`: Refresh token expiration (seconds, recommended: 2592000 = 30 days)
- `AUTHORIZATION_CODE_LIFETIME`: Auth code expiration (seconds, recommended: 600 = 10 minutes)

ServiceNow Configuration:

- `SERVICENOW_INSTANCE`: ServiceNow instance URL for CORS configuration

External Service Configuration:

- `REDIS_HOST`: Redis server hostname (if using Redis)
- `REDIS_PORT`: Redis server port (if using Redis)
- Database connection strings (if using database storage)

Implementation Approach

Environment variables provide the foundation for configuration management, enabling your MCP server to adapt to different deployment environments without code changes. Load environment variables at the very start of your application, before any other initialization occurs. Use default values for optional settings while requiring explicit values for security-critical configuration like JWT secrets.

Key considerations:

- Load environment variables before initializing any services
- Use descriptive variable names with consistent prefixes (e.g., JWT_, REDIS_)
- Provide sensible defaults for non-critical settings (ports, hostnames)
- Never hardcode secrets or environment-specific values in code
- Document all environment variables in a .env.example file

Pseudocode:

```
// Load environment configuration at application start
LOAD_ENVIRONMENT_VARIABLES_FROM_FILE('.env')

// Server Configuration
PORT = GET_ENV('PORT') OR 3000
NODE_ENV = GET_ENV('NODE_ENV') OR 'development'
SERVER_URL = GET_ENV('SERVER_URL') OR 'http://localhost:3000'

// Authentication Configuration (no defaults - must be provided)
JWT_SECRET = GET_ENV('JWT_SECRET')
JWT_ISSUER = GET_ENV('JWT_ISSUER') OR SERVER_URL
DCR_AUTH_TOKEN = GET_ENV('DCR_AUTH_TOKEN')
MCP_API_KEY = GET_ENV('MCP_API_KEY')

// Token Lifetime Configuration (defaults provided)
ACCESS_TOKEN_LIFETIME = GET_ENV('ACCESS_TOKEN_LIFETIME') OR 3600
REFRESH_TOKEN_LIFETIME = GET_ENV('REFRESH_TOKEN_LIFETIME') OR 2592000
AUTHORIZATION_CODE_LIFETIME = GET_ENV('AUTHORIZATION_CODE_LIFETIME') OR 600

// ServiceNow Configuration (no default - must be provided)
SERVICENOW_INSTANCE = GET_ENV('SERVICENOW_INSTANCE')

// External Service Configuration (defaults for local development)
REDIS_HOST = GET_ENV('REDIS_HOST') OR 'localhost'
REDIS_PORT = GET_ENV('REDIS_PORT') OR 6379

LOG("Environment variables loaded successfully")
```

JavaScript:

```
javascript
```

```
require('dotenv').config();

// Server Configuration
const PORT = process.env.PORT || 3000;
const NODE_ENV = process.env.NODE_ENV || 'development';
const SERVER_URL = process.env.SERVER_URL || `http://localhost:${PORT}`;

// Authentication Configuration (no defaults - must be provided)
const JWT_SECRET = process.env.JWT_SECRET;
const JWT_ISSUER = process.env.JWT_ISSUER || SERVER_URL;
const DCR_AUTH_TOKEN = process.env.DCR_AUTH_TOKEN;
const MCP_API_KEY = process.env.MCP_API_KEY;

// Token Lifetime Configuration (defaults provided)
const ACCESS_TOKEN_LIFETIME = parseInt(process.env.ACCESS_TOKEN_LIFETIME || '3600', 10);
const REFRESH_TOKEN_LIFETIME = parseInt(process.env.REFRESH_TOKEN_LIFETIME || '2592000', 10);
const AUTHORIZATION_CODE_LIFETIME = parseInt(process.env.AUTHORIZATION_CODE_LIFETIME || '600', 10);

// ServiceNow Configuration (no default - must be provided)
const SERVICENOW_INSTANCE = process.env.SERVICENOW_INSTANCE;

// External Service Configuration (defaults for local development)
const REDIS_HOST = process.env.REDIS_HOST || 'localhost';
const REDIS_PORT = parseInt(process.env.REDIS_PORT || '6379', 10);

console.log('[CONFIG] Environment variables loaded successfully');
```

TypeScript:

typescript

```
import dotenv from 'dotenv';
dotenv.config();

// Server Configuration
const PORT: number = parseInt(process.env.PORT || '3000', 10);
const NODE_ENV: string = process.env.NODE_ENV || 'development';
const SERVER_URL: string = process.env.SERVER_URL || `http://localhost:${PORT}`;

// Authentication Configuration (no defaults - must be provided)
const JWT_SECRET: string | undefined = process.env.JWT_SECRET;
const JWT_ISSUER: string = process.env.JWT_ISSUER || SERVER_URL;
const DCR_AUTH_TOKEN: string | undefined = process.env.DCR_AUTH_TOKEN;
const MCP_API_KEY: string | undefined = process.env.MCP_API_KEY;

// Token Lifetime Configuration (defaults provided)
const ACCESS_TOKEN_LIFETIME: number = parseInt(process.env.ACCESS_TOKEN_LIFETIME || '3600', 10);
const REFRESH_TOKEN_LIFETIME: number = parseInt(process.env.REFRESH_TOKEN_LIFETIME || '2592000', 10);
const AUTHORIZATION_CODE_LIFETIME: number = parseInt(process.env.AUTHORIZATION_CODE_LIFETIME || '60

// ServiceNow Configuration (no default - must be provided)
const SERVICENOW_INSTANCE: string | undefined = process.env.SERVICENOW_INSTANCE;

// External Service Configuration (defaults for local development)
const REDIS_HOST: string = process.env.REDIS_HOST || 'localhost';
const REDIS_PORT: number = parseInt(process.env.REDIS_PORT || '6379', 10);

console.log('[CONFIG] Environment variables loaded successfully');
```

Secret Management

Development:

- Use `.env` files with dotenv or similar library
- Never commit `.env` files to version control (add to `.gitignore`)
- Provide `.env.example` template with dummy values

Production:

- Use cloud provider secret managers (GCP Secret Manager, AWS Secrets Manager, Azure Key Vault)
- Use environment variables injected at container runtime
- Implement secret rotation capabilities

- Restrict access to secrets using IAM policies

Implementation Approach

Secret management separates security-critical values from your codebase, reducing the risk of accidental exposure through version control or logs. In development, store secrets in a local .env file that never gets committed. In production, leverage cloud provider secret management services that provide encryption, access controls, and audit logging. Always treat secrets as sensitive data that requires special handling throughout their lifecycle.

Key considerations:

- Never commit .env files containing real secrets to version control
- Provide .env.example with dummy values as a template for developers
- Use cloud secret managers (GCP Secret Manager, AWS Secrets Manager, Azure Key Vault) in production
- Implement secret rotation capabilities for long-lived deployments
- Restrict access to secrets using IAM policies and role-based access controls
- Log secret access for audit purposes (but never log the secret values themselves)

Pseudocode:

```
// Development Environment Setup
CREATE_FILE('.env.example'):
  CONTENT = "
  # Server Configuration
  PORT=3000
  NODE_ENV=development
  SERVER_URL=http://localhost:3000

  # Authentication (REPLACE WITH SECURE VALUES)
  JWT_SECRET=your-32-character-secret-here
  DCR_AUTH_TOKEN=your-32-character-dcr-token

  # ServiceNow Configuration
  SERVICENOW_INSTANCE=https://your-instance.service-now.com
  "
  WRITE_TO_FILE('.env.example', CONTENT)
END CREATE_FILE

// Add to .gitignore
APPEND_TO_FILE('.gitignore', '.env')

// Production Environment Setup
FUNCTION load_production_secrets():
  IF CLOUD_PROVIDER == 'GCP':
    JWT_SECRET = GET_SECRET_FROM_GCP('jwt-secret')
    DCR_AUTH_TOKEN = GET_SECRET_FROM_GCP('dcr-auth-token')
  ELSE IF CLOUD_PROVIDER == 'AWS':
    JWT_SECRET = GET_SECRET_FROM_AWS('jwt-secret')
    DCR_AUTH_TOKEN = GET_SECRET_FROM_AWS('dcr-auth-token')
  ELSE IF CLOUD_PROVIDER == 'AZURE':
    JWT_SECRET = GET_SECRET_FROM_AZURE('jwt-secret')
    DCR_AUTH_TOKEN = GET_SECRET_FROM_AZURE('dcr-auth-token')
  ELSE:
    // Fallback to environment variables injected at runtime
    JWT_SECRET = GET_ENV('JWT_SECRET')
    DCR_AUTH_TOKEN = GET_ENV('DCR_AUTH_TOKEN')
  END IF

  RETURN { JWT_SECRET, DCR_AUTH_TOKEN }
END FUNCTION
```

JavaScript:

javascript

```
// Development: .env.example file
// Create this file and commit it to version control
/*
# Server Configuration
PORT=3000
NODE_ENV=development
SERVER_URL=http://localhost:3000

# Authentication (REPLACE WITH SECURE VALUES)
JWT_SECRET=your-32-character-secret-here-replace-me
DCR_AUTH_TOKEN=your-32-character-dcr-token-replace-me
MCP_API_KEY=your-api-key-here

# ServiceNow Configuration
SERVICENOW_INSTANCE=https://your-instance.service-now.com

# Redis Configuration
REDIS_HOST=localhost
REDIS_PORT=6379
*/

// Development: Load from .env file
require('dotenv').config();

// Production: Load from cloud secret manager (example with GCP)
async function loadProductionSecrets() {
  const isProduction = process.env.NODE_ENV === 'production';

  if (!isProduction) {
    // Development: use environment variables from .env
    return {
      JWT_SECRET: process.env.JWT_SECRET,
      DCR_AUTH_TOKEN: process.env.DCR_AUTH_TOKEN
    };
  }

  // Production: load from secret manager
  // Example for GCP Secret Manager
  try {
    const { SecretManagerServiceClient } = require('@google-cloud/secret-manager');
    const client = new SecretManagerServiceClient();

    const projectId = process.env.GCP_PROJECT_ID;
```

```
    // Access secrets
    const [jwtSecretVersion] = await client.accessSecretVersion({
      name: `projects/${projectId}/secrets/jwt-secret/versions/latest`
    });

    const [dcrTokenVersion] = await client.accessSecretVersion({
      name: `projects/${projectId}/secrets/dcr-auth-token/versions/latest`
    });

    return {
      JWT_SECRET: jwtSecretVersion.payload.data.toString(),
      DCR_AUTH_TOKEN: dcrTokenVersion.payload.data.toString()
    };
  } catch (error) {
    console.error('[ERROR] Failed to load secrets from Secret Manager:', error);
    throw error;
  }
}

// .gitignore file (ensure .env is never committed)
/*
.env
node_modules/
*.log
*/

console.log('[CONFIG] Secret management configured');
```

TypeScript:

typescript

```
// Development: .env.example file
// Create this file and commit it to version control
/*
# Server Configuration
PORT=3000
NODE_ENV=development
SERVER_URL=http://localhost:3000

# Authentication (REPLACE WITH SECURE VALUES)
JWT_SECRET=your-32-character-secret-here-replace-me
DCR_AUTH_TOKEN=your-32-character-dcr-token-replace-me
MCP_API_KEY=your-api-key-here

# ServiceNow Configuration
SERVICENOW_INSTANCE=https://your-instance.service-now.com

# Redis Configuration
REDIS_HOST=localhost
REDIS_PORT=6379
*/

import dotenv from 'dotenv';

// Development: Load from .env file
dotenv.config();

// Define secret structure
interface Secrets {
  JWT_SECRET: string;
  DCR_AUTH_TOKEN: string;
}

// Production: Load from cloud secret manager (example with GCP)
async function loadProductionSecrets(): Promise<Secrets> {
  const isProduction: boolean = process.env.NODE_ENV === 'production';

  if (!isProduction) {
    // Development: use environment variables from .env
    return {
      JWT_SECRET: process.env.JWT_SECRET || '',
      DCR_AUTH_TOKEN: process.env.DCR_AUTH_TOKEN || ''
    };
  }
```

```
  // Production: load from secret manager
  // Example for GCP Secret Manager
  try {
    const { SecretManagerServiceClient } = require('@google-cloud/secret-manager');
    const client = new SecretManagerServiceClient();

    const projectId: string = process.env.GCP_PROJECT_ID || '';

    // Access secrets
    const [jwtSecretVersion] = await client.accessSecretVersion({
      name: `projects/${projectId}/secrets/jwt-secret/versions/latest`
    });

    const [dcrTokenVersion] = await client.accessSecretVersion({
      name: `projects/${projectId}/secrets/dcr-auth-token/versions/latest`
    });

    return {
      JWT_SECRET: jwtSecretVersion.payload.data.toString(),
      DCR_AUTH_TOKEN: dcrTokenVersion.payload.data.toString()
    };
  } catch (error) {
    console.error('[ERROR] Failed to load secrets from Secret Manager:', error);
    throw error;
  }
}

// .gitignore file (ensure .env is never committed)
/*
.env
node_modules/
*.log
*/

console.log('[CONFIG] Secret management configured');
```

Configuration Validation

Validate required configuration at server startup:

- Check that all required environment variables are present
- Validate format and length of secrets (e.g., JWT_SECRET must be 32+ bytes)
- Verify URLs are properly formatted

- Fail fast if configuration is invalid (don't start server with missing/invalid config)

Implementation Approach

Configuration validation ensures your server never starts in an invalid state that could cause runtime failures or security vulnerabilities. Validate all required configuration at startup, before initializing any services or accepting connections. This "fail fast" approach prevents hard-to-debug issues that arise from missing or malformed configuration discovered only when a specific code path executes.

Key considerations:

- Validate configuration immediately after loading, before any service initialization
- Check that all required environment variables are present
- Validate format and constraints (e.g., JWT_SECRET length, URL format, numeric ranges)
- Exit immediately with clear error messages if validation fails
- Log successful validation to confirm configuration is loaded correctly
- In production, validation failures should trigger alerting/monitoring

Pseudocode:

```
FUNCTION validate_configuration():
  errors = []

  // Validate JWT_SECRET
  IF JWT_SECRET IS NULL:
    errors.ADD("JWT_SECRET is required but not set")
  ELSE IF LENGTH(JWT_SECRET) < 32:
    errors.ADD("JWT_SECRET must be at least 32 characters (current: " + LENGTH(JWT_SECRET) + ")")
  END IF

  // Validate ServiceNow instance
  IF SERVICENOW_INSTANCE IS NULL:
    errors.ADD("SERVICENOW_INSTANCE is required but not set")
  ELSE IF NOT SERVICENOW_INSTANCE.STARTS_WITH("https://"):
    errors.ADD("SERVICENOW_INSTANCE must be an HTTPS URL")
  END IF

  // Validate DCR auth token
  IF DCR_AUTH_TOKEN IS NULL:
    errors.ADD("DCR_AUTH_TOKEN is required but not set")
  ELSE IF LENGTH(DCR_AUTH_TOKEN) < 32:
    errors.ADD("DCR_AUTH_TOKEN must be at least 32 characters")
  END IF

  // Validate token lifetimes (must be positive numbers)
  IF ACCESS_TOKEN_LIFETIME <= 0:
    errors.ADD("ACCESS_TOKEN_LIFETIME must be a positive number")
  END IF

  IF REFRESH_TOKEN_LIFETIME <= 0:
    errors.ADD("REFRESH_TOKEN_LIFETIME must be a positive number")
  END IF

  IF AUTHORIZATION_CODE_LIFETIME <= 0:
    errors.ADD("AUTHORIZATION_CODE_LIFETIME must be a positive number")
  END IF

  // If errors exist, log and exit
  IF errors.LENGTH > 0:
    LOG_ERROR("Configuration validation failed:")
    FOR EACH error IN errors:
      LOG_ERROR("  ❌ " + error)
    END FOR
```

```
    LOG_ERROR("")
    LOG_ERROR("Please check your environment variables and try again.")
    EXIT(1)  // Exit with error code
  END IF

  // Validation passed
  LOG("✅ Configuration validation passed")
  LOG("  - JWT_SECRET: configured (" + LENGTH(JWT_SECRET) + " characters)")
  LOG("  - SERVICENOW_INSTANCE: " + SERVICENOW_INSTANCE)
  LOG("  - DCR_AUTH_TOKEN: configured")
  LOG("  - Token lifetimes: access=" + ACCESS_TOKEN_LIFETIME + "s, refresh=" +
REFRESH_TOKEN_LIFETIME + "s")
END FUNCTION

// Call validation before any other initialization
validate_configuration()
```

JavaScript:

```
javascript
```

```
function validateConfiguration() {
  const errors = [];

  // Validate JWT_SECRET
  if (!JWT_SECRET) {
    errors.push('JWT_SECRET is required but not set');
  } else if (JWT_SECRET.length < 32) {
    errors.push(`JWT_SECRET must be at least 32 characters (current: ${JWT_SECRET.length})`);
  }

  // Validate ServiceNow instance
  if (!SERVICENOW_INSTANCE) {
    errors.push('SERVICENOW_INSTANCE is required but not set');
  } else if (!SERVICENOW_INSTANCE.startsWith('https://')) {
    errors.push('SERVICENOW_INSTANCE must be an HTTPS URL');
  }

  // Validate DCR auth token
  if (!DCR_AUTH_TOKEN) {
    errors.push('DCR_AUTH_TOKEN is required but not set');
  } else if (DCR_AUTH_TOKEN.length < 32) {
    errors.push(`DCR_AUTH_TOKEN must be at least 32 characters (current: ${DCR_AUTH_TOKEN.length})`);
  }

  // Validate token lifetimes (must be positive numbers)
  if (ACCESS_TOKEN_LIFETIME <= 0) {
    errors.push('ACCESS_TOKEN_LIFETIME must be a positive number');
  }

  if (REFRESH_TOKEN_LIFETIME <= 0) {
    errors.push('REFRESH_TOKEN_LIFETIME must be a positive number');
  }

  if (AUTHORIZATION_CODE_LIFETIME <= 0) {
    errors.push('AUTHORIZATION_CODE_LIFETIME must be a positive number');
  }

  // If errors exist, log and exit
  if (errors.length > 0) {
    console.error('[CONFIG] Configuration validation failed:');
    errors.forEach(error => {
      console.error(`  ❌ ${error}`);
    });
```

```
    console.error('');
    console.error('Please check your environment variables and try again.');
    process.exit(1);
  }

  // Validation passed
  console.log('[CONFIG] ✅ Configuration validation passed');
  console.log(`[CONFIG]   - JWT_SECRET: configured (${JWT_SECRET.length} characters)`);
  console.log(`[CONFIG]   - SERVICENOW_INSTANCE: ${SERVICENOW_INSTANCE}`);
  console.log(`[CONFIG]   - DCR_AUTH_TOKEN: configured`);
  console.log(`[CONFIG]   - Token lifetimes: access=${ACCESS_TOKEN_LIFETIME}s, refresh=${REFRESH_TOKEN_LI
}

// Call validation before any other initialization
validateConfiguration();
```

TypeScript:

typescript

```
function validateConfiguration(): void {
  const errors: string[] = [];

  // Validate JWT_SECRET
  if (!JWT_SECRET) {
    errors.push('JWT_SECRET is required but not set');
  } else if (JWT_SECRET.length < 32) {
    errors.push(`JWT_SECRET must be at least 32 characters (current: ${JWT_SECRET.length})`);
  }

  // Validate ServiceNow instance
  if (!SERVICENOW_INSTANCE) {
    errors.push('SERVICENOW_INSTANCE is required but not set');
  } else if (!SERVICENOW_INSTANCE.startsWith('https://')) {
    errors.push('SERVICENOW_INSTANCE must be an HTTPS URL');
  }

  // Validate DCR auth token
  if (!DCR_AUTH_TOKEN) {
    errors.push('DCR_AUTH_TOKEN is required but not set');
  } else if (DCR_AUTH_TOKEN.length < 32) {
    errors.push(`DCR_AUTH_TOKEN must be at least 32 characters (current: ${DCR_AUTH_TOKEN.length})`);
  }

  // Validate token lifetimes (must be positive numbers)
  if (ACCESS_TOKEN_LIFETIME <= 0) {
    errors.push('ACCESS_TOKEN_LIFETIME must be a positive number');
  }

  if (REFRESH_TOKEN_LIFETIME <= 0) {
    errors.push('REFRESH_TOKEN_LIFETIME must be a positive number');
  }

  if (AUTHORIZATION_CODE_LIFETIME <= 0) {
    errors.push('AUTHORIZATION_CODE_LIFETIME must be a positive number');
  }

  // If errors exist, log and exit
  if (errors.length > 0) {
    console.error('[CONFIG] Configuration validation failed:');
    errors.forEach((error: string) => {
      console.error(`  ❌ ${error}`);
    });
```

```
    console.error('');
    console.error('Please check your environment variables and try again.');
    process.exit(1);
  }

  // Validation passed
  console.log('[CONFIG] ✅ Configuration validation passed');
  console.log(`[CONFIG]   - JWT_SECRET: configured (${JWT_SECRET.length} characters)`);
  console.log(`[CONFIG]   - SERVICENOW_INSTANCE: ${SERVICENOW_INSTANCE}`);
  console.log(`[CONFIG]   - DCR_AUTH_TOKEN: configured`);
  console.log(`[CONFIG]   - Token lifetimes: access=${ACCESS_TOKEN_LIFETIME}s, refresh=${REFRESH_TOKEN_LI
}

// Call validation before any other initialization
validateConfiguration();
```

---

4. Data Storage Initialization

Determine and initialize your storage approach for client registrations, authorization codes, and token revocations.

Important: The storage solutions presented in this section are **recommended examples**, not mandatory requirements. Choose storage mechanisms that align with your deployment architecture, scalability needs, and operational expertise. The examples use Redis and file-based storage, but you can substitute PostgreSQL, MongoDB, or other solutions that meet the same functional requirements.

Storage Architecture Decision

Choose storage strategy based on deployment architecture:

Single-Server Deployment:

- In-memory structures with file-based persistence acceptable
- Simpler implementation
- Lower operational complexity
- Suitable for POC and small-scale deployments

Multi-Server Deployment:

- Shared external storage required (Redis, PostgreSQL, etc.)
- Enables horizontal scaling
- Required for high availability

- Necessary for production at scale

Implementation Approach

The storage architecture decision shapes your MCP server's scalability, reliability, and operational complexity. Single-server deployments can leverage simpler solutions like file-based storage combined with in-memory structures, while multi-server deployments require shared external storage to maintain consistency across instances. Make this decision early, as it impacts all subsequent storage implementations.

Key considerations:

- Single-server deployments prioritize simplicity and minimal dependencies
- Multi-server deployments require shared storage for client registrations and token revocations
- Authorization codes can remain in-memory (short-lived, single-use)
- Consider your team's operational expertise (managing Redis vs PostgreSQL vs file systems)
- Balance persistence requirements with performance needs
- Plan for future scaling even if starting with single-server

Pseudocode:

```
// Decision framework at application initialization
FUNCTION determine_storage_architecture():
    deployment_type = GET_ENV('DEPLOYMENT_TYPE') OR 'single-server'

    IF deployment_type == 'single-server':
        LOG("Using single-server storage architecture")

        // Client registry: File-based with in-memory cache
        client_storage = {
            type: 'file',
            path: './registered_clients.json',
            in_memory_cache: true
        }

        // Authorization codes: In-memory only (short-lived)
        auth_code_storage = {
            type: 'in-memory',
            cleanup_interval: 3600000  // 1 hour
        }

        // Token revocations: File-based or Redis
        token_revocation_storage = {
            type: GET_ENV('TOKEN_STORAGE') OR 'redis',
            fallback: 'in-memory'
        }

    ELSE IF deployment_type == 'multi-server':
        LOG("Using multi-server storage architecture (requires shared storage)")

        // Client registry: Shared database
        client_storage = {
            type: 'database',  // PostgreSQL, MongoDB, etc.
            shared: true
        }

        // Authorization codes: Redis (shared across servers)
        auth_code_storage = {
            type: 'redis',
            shared: true,
            ttl: AUTHORIZATION_CODE_LIFETIME
        }

        // Token revocations: Redis (shared across servers)
```

```
    token_revocation_storage = {
      type: 'redis',
      shared: true
    }
  END IF

  RETURN {
    client_storage,
    auth_code_storage,
    token_revocation_storage
  }
END FUNCTION

// Initialize storage based on architecture decision
storage_config = determine_storage_architecture()
LOG("Storage architecture configured: " + storage_config.deployment_type)
```

JavaScript:

```
javascript
```

```
// Storage architecture configuration
const DEPLOYMENT_TYPE = process.env.DEPLOYMENT_TYPE || 'single-server';

function determineStorageArchitecture() {
  if (DEPLOYMENT_TYPE === 'single-server') {
    console.log('[STORAGE] Using single-server storage architecture');

    return {
      deploymentType: 'single-server',
      clientStorage: {
        type: 'file',
        path: './registered_clients.json',
        inMemoryCache: true
      },
      authCodeStorage: {
        type: 'in-memory',
        cleanupInterval: 3600000  // 1 hour
      },
      tokenRevocationStorage: {
        type: process.env.TOKEN_STORAGE || 'redis',
        fallback: 'in-memory'
      }
    };
  } else if (DEPLOYMENT_TYPE === 'multi-server') {
    console.log('[STORAGE] Using multi-server storage architecture (requires shared storage)');

    return {
      deploymentType: 'multi-server',
      clientStorage: {
        type: 'database',  // PostgreSQL, MongoDB, etc.
        shared: true
      },
      authCodeStorage: {
        type: 'redis',
        shared: true,
        ttl: AUTHORIZATION_CODE_LIFETIME
      },
      tokenRevocationStorage: {
        type: 'redis',
        shared: true
      }
    };
  } else {
```

```
    throw new Error(`Unknown deployment type: ${DEPLOYMENT_TYPE}`);
  }
}

// Initialize storage architecture
const storageConfig = determineStorageArchitecture();
console.log(`[STORAGE] Storage architecture configured: ${storageConfig.deploymentType}`);
```

TypeScript:

```
typescript
```

```typescript
// Storage architecture types
type DeploymentType = 'single-server' | 'multi-server';
type StorageType = 'file' | 'in-memory' | 'redis' | 'database';

interface StorageConfig {
  type: StorageType;
  path?: string;
  inMemoryCache?: boolean;
  cleanupInterval?: number;
  shared?: boolean;
  ttl?: number;
  fallback?: StorageType;
}

interface StorageArchitecture {
  deploymentType: DeploymentType;
  clientStorage: StorageConfig;
  authCodeStorage: StorageConfig;
  tokenRevocationStorage: StorageConfig;
}

// Storage architecture configuration
const DEPLOYMENT_TYPE: DeploymentType = (process.env.DEPLOYMENT_TYPE as DeploymentType) || 'single-server

function determineStorageArchitecture(): StorageArchitecture {
  if (DEPLOYMENT_TYPE === 'single-server') {
    console.log('[STORAGE] Using single-server storage architecture');

    return {
      deploymentType: 'single-server',
      clientStorage: {
        type: 'file',
        path: './registered_clients.json',
        inMemoryCache: true
      },
      authCodeStorage: {
        type: 'in-memory',
        cleanupInterval: 3600000  // 1 hour
      },
      tokenRevocationStorage: {
        type: (process.env.TOKEN_STORAGE as StorageType) || 'redis',
        fallback: 'in-memory'
      }
```

```
    };
  } else if (DEPLOYMENT_TYPE === 'multi-server') {
    console.log('[STORAGE] Using multi-server storage architecture (requires shared storage)');

    return {
      deploymentType: 'multi-server',
      clientStorage: {
        type: 'database',  // PostgreSQL, MongoDB, etc.
        shared: true
      },
      authCodeStorage: {
        type: 'redis',
        shared: true,
        ttl: AUTHORIZATION_CODE_LIFETIME
      },
      tokenRevocationStorage: {
        type: 'redis',
        shared: true
      }
    };
  } else {
    throw new Error(`Unknown deployment type: ${DEPLOYMENT_TYPE}`);
  }
}

// Initialize storage architecture
const storageConfig: StorageArchitecture = determineStorageArchitecture();
console.log(`[STORAGE] Storage architecture configured: ${storageConfig.deploymentType}`);
```

Client Registry Setup

Initialize storage for registered OAuth clients (from Dynamic Client Registration):

Required Data:

- Client ID (unique identifier)
- Client Secret (hashed or encrypted)
- Client Name (human-readable identifier)
- Redirect URIs (array of authorized callback URLs)
- Grant Types (supported OAuth flows)
- Creation timestamp

Storage Options:

- **File-based**: JSON file persisted to disk, loaded at startup
- **Database**: PostgreSQL, MongoDB, MySQL
- **Redis**: Key-value storage with optional persistence

Persistence Requirement:

- Client registrations MUST survive server restarts
- Losing client data forces ServiceNow re-registration

Implementation Approach

The client registry stores OAuth client credentials registered through Dynamic Client Registration (DCR). This data must persist across server restarts, as losing client registrations forces ServiceNow to re-register, disrupting active integrations. Implement both in-memory storage for fast access and persistent storage (file or database) for durability. Load registered clients at startup and save after each registration.

Key considerations:

- Client registrations must survive server restarts
- In-memory Map provides fast lookup during OAuth flows
- File-based storage acceptable for single-server deployments
- Database storage required for multi-server deployments
- Client secrets should be stored securely (consider hashing in production)
- Include timestamps for audit and cleanup purposes

Pseudocode:

```
// Initialize client registry storage
registered_clients = NEW_MAP()  // In-memory for fast access
CLIENTS_FILE = './registered_clients.json'

FUNCTION load_clients_from_storage():
    IF FILE_EXISTS(CLIENTS_FILE):
        TRY:
            file_content = READ_FILE(CLIENTS_FILE)
            clients = PARSE_JSON(file_content)

            FOR EACH (client_id, client_data) IN clients:
                registered_clients.SET(client_id, client_data)
            END FOR

            LOG("Loaded " + registered_clients.SIZE + " client(s) from storage")
        CATCH error:
            LOG_ERROR("Failed to load clients: " + error.message)
            // Continue with empty registry - don't fail startup
        END TRY
    ELSE:
        LOG("No existing client registry found, starting with empty registry")
    END IF
END FUNCTION

FUNCTION save_clients_to_storage():
    TRY:
        // Convert Map to plain object for JSON serialization
        clients_object = CONVERT_MAP_TO_OBJECT(registered_clients)
        json_string = STRINGIFY_JSON(clients_object, indent=2)

        WRITE_FILE(CLIENTS_FILE, json_string)
        LOG("Saved " + registered_clients.SIZE + " client(s) to storage")
    CATCH error:
        LOG_ERROR("Failed to save clients: " + error.message)
        // Don't throw - log and continue
    END TRY
END FUNCTION

FUNCTION register_new_client(client_data):
    client_id = client_data.clientId

    // Store in-memory
    registered_clients.SET(client_id, client_data)
```

```
    // Persist to storage
    save_clients_to_storage()

    LOG("Registered new client: " + client_id)
    RETURN client_data
END FUNCTION

// Load clients at startup
load_clients_from_storage()
```

JavaScript:

```javascript
```

```
const fs = require('fs');
const path = require('path');

// Initialize client registry storage
const registeredClients = new Map();
const CLIENTS_FILE = path.join(__dirname, 'registered_clients.json');

function loadClientsFromStorage() {
  if (fs.existsSync(CLIENTS_FILE)) {
    try {
      const fileContent = fs.readFileSync(CLIENTS_FILE, 'utf8');
      const clients = JSON.parse(fileContent);

      // Populate in-memory Map
      for (const [clientId, clientData] of Object.entries(clients)) {
        registeredClients.set(clientId, clientData);
      }

      console.log(`[STORAGE] Loaded ${registeredClients.size} client(s) from storage`);
    } catch (error) {
      console.error('[STORAGE] Failed to load clients:', error.message);
      // Continue with empty registry - don't fail startup
    }
  } else {
    console.log('[STORAGE] No existing client registry found, starting with empty registry');
  }
}

function saveClientsToStorage() {
  try {
    // Convert Map to plain object for JSON serialization
    const clientsObject = Object.fromEntries(registeredClients);
    const jsonString = JSON.stringify(clientsObject, null, 2);

    fs.writeFileSync(CLIENTS_FILE, jsonString, 'utf8');
    console.log(`[STORAGE] Saved ${registeredClients.size} client(s) to storage`);
  } catch (error) {
    console.error('[STORAGE] Failed to save clients:', error.message);
    // Don't throw - log and continue
  }
}

function registerNewClient(clientData) {
```

```
  const clientId = clientData.clientId;

  // Store in-memory
  registeredClients.set(clientId, clientData);

  // Persist to storage
  saveClientsToStorage();

  console.log(`[STORAGE] Registered new client: ${clientId}`);
  return clientData;
}

// Load clients at startup
loadClientsFromStorage();
```

TypeScript:

```
typescript
```

```
import fs from 'fs';
import path from 'path';

// Client data structure
interface ClientData {
  clientId: string;
  clientSecret: string;
  clientName: string;
  redirectUris: string[];
  grantTypes: string[];
  responseTypes: string[];
  tokenEndpointAuthMethod: string;
  usePkce: boolean;
  createdAt: string;
}

// Initialize client registry storage
const registeredClients = new Map<string, ClientData>();
const CLIENTS_FILE: string = path.join(__dirname, 'registered_clients.json');

function loadClientsFromStorage(): void {
  if (fs.existsSync(CLIENTS_FILE)) {
    try {
      const fileContent: string = fs.readFileSync(CLIENTS_FILE, 'utf8');
      const clients: Record<string, ClientData> = JSON.parse(fileContent);

      // Populate in-memory Map
      for (const [clientId, clientData] of Object.entries(clients)) {
        registeredClients.set(clientId, clientData);
      }

      console.log(`[STORAGE] Loaded ${registeredClients.size} client(s) from storage`);
    } catch (error) {
      console.error('[STORAGE] Failed to load clients:', (error as Error).message);
      // Continue with empty registry - don't fail startup
    }
  } else {
    console.log('[STORAGE] No existing client registry found, starting with empty registry');
  }
}

function saveClientsToStorage(): void {
  try {
```

```
    // Convert Map to plain object for JSON serialization
    const clientsObject: Record<string, ClientData> = Object.fromEntries(registeredClients);
    const jsonString: string = JSON.stringify(clientsObject, null, 2);

    fs.writeFileSync(CLIENTS_FILE, jsonString, 'utf8');
    console.log(`[STORAGE] Saved ${registeredClients.size} client(s) to storage`);
  } catch (error) {
    console.error('[STORAGE] Failed to save clients:', (error as Error).message);
    // Don't throw - log and continue
  }
}

function registerNewClient(clientData: ClientData): ClientData {
  const clientId: string = clientData.clientId;

  // Store in-memory
  registeredClients.set(clientId, clientData);

  // Persist to storage
  saveClientsToStorage();

  console.log(`[STORAGE] Registered new client: ${clientId}`);
  return clientData;
}

// Load clients at startup
loadClientsFromStorage();
```

Authorization Code Storage

Temporary storage for authorization codes during OAuth flow:

Required Data:

- Authorization code (random string)
- Client ID (which client requested the code)
- User ID (simulated or actual user identifier)
- Redirect URI (must match token exchange request)
- Code Challenge (PKCE challenge for validation)
- Scope (requested OAuth scopes)
- Creation timestamp

Storage Characteristics:

- Short-lived (10 minutes maximum)
- Single-use (deleted after token exchange)
- Can be in-memory (no persistence required)

Cleanup Strategy:

- Implement periodic cleanup of expired codes (e.g., hourly)
- Prevent memory leaks from unused codes
- Log cleanup operations for monitoring

Implementation Approach

Authorization codes are short-lived, single-use tokens that bridge the OAuth authorization and token exchange steps. Since they expire in 10 minutes and are deleted immediately after use, in-memory storage is sufficient and optimal. Implement periodic cleanup to prevent memory leaks from unused codes. No persistence is required—if the server restarts during an OAuth flow, the user simply re-authenticates.

Key considerations:

- Authorization codes are short-lived (10 minutes) and single-use
- In-memory storage is sufficient and provides best performance
- No persistence required—restart forces re-authentication (acceptable)
- Implement cleanup of expired codes to prevent memory leaks
- Store PKCE challenge with the code for later validation
- Include timestamp to enable expiration checking

Pseudocode:

```
// Initialize authorization code storage
authorization_codes = NEW_MAP()  // In-memory only

FUNCTION store_authorization_code(auth_code, code_data):
    // Add creation timestamp
    code_data.created = CURRENT_TIMESTAMP()

    // Store code with associated data
    authorization_codes.SET(auth_code, code_data)

    LOG("Authorization code generated for client: " + code_data.clientId)
END FUNCTION

FUNCTION retrieve_and_delete_code(auth_code):
    // Get code data
    code_data = authorization_codes.GET(auth_code)

    IF code_data IS NULL:
        RETURN null  // Code doesn't exist or already used
    END IF

    // Check expiration
    age = CURRENT_TIMESTAMP() - code_data.created
    IF age > AUTHORIZATION_CODE_LIFETIME * 1000:
        authorization_codes.DELETE(auth_code)
        RETURN null  // Code expired
    END IF

    // Delete code (single-use)
    authorization_codes.DELETE(auth_code)

    LOG("Authorization code consumed for client: " + code_data.clientId)
    RETURN code_data
END FUNCTION

FUNCTION cleanup_expired_codes():
    current_time = CURRENT_TIMESTAMP()
    expiration_time = AUTHORIZATION_CODE_LIFETIME * 1000
    cleaned_count = 0

    FOR EACH (code, code_data) IN authorization_codes:
        age = current_time - code_data.created
        IF age > expiration_time:
```

```
        authorization_codes.DELETE(code)
        cleaned_count = cleaned_count + 1
      END IF
    END FOR

    IF cleaned_count > 0:
      LOG("Cleaned up " + cleaned_count + " expired authorization code(s)")
    END IF
  END FUNCTION

  // Schedule periodic cleanup (every hour)
  SET_INTERVAL(cleanup_expired_codes, 3600000)
```

JavaScript:

```
javascript
```

```
// Initialize authorization code storage
const authorizationCodes = new Map();

function storeAuthorizationCode(authCode, codeData) {
  // Add creation timestamp
  codeData.created = Date.now();

  // Store code with associated data
  authorizationCodes.set(authCode, codeData);

  console.log(`[OAUTH] Authorization code generated for client: ${codeData.clientId}`);
}

function retrieveAndDeleteCode(authCode) {
  // Get code data
  const codeData = authorizationCodes.get(authCode);

  if (!codeData) {
    return null;  // Code doesn't exist or already used
  }

  // Check expiration
  const age = Date.now() - codeData.created;
  if (age > AUTHORIZATION_CODE_LIFETIME * 1000) {
    authorizationCodes.delete(authCode);
    return null;  // Code expired
  }

  // Delete code (single-use)
  authorizationCodes.delete(authCode);

  console.log(`[OAUTH] Authorization code consumed for client: ${codeData.clientId}`);
  return codeData;
}

function cleanupExpiredCodes() {
  const currentTime = Date.now();
  const expirationTime = AUTHORIZATION_CODE_LIFETIME * 1000;
  let cleanedCount = 0;

  for (const [code, codeData] of authorizationCodes.entries()) {
    const age = currentTime - codeData.created;
    if (age > expirationTime) {
```

```
      authorizationCodes.delete(code);
      cleanedCount++;
    }
  }

  if (cleanedCount > 0) {
    console.log(`[CLEANUP] Cleaned up ${cleanedCount} expired authorization code(s)`);
  }
}

// Schedule periodic cleanup (every hour)
setInterval(cleanupExpiredCodes, 3600000);
```

TypeScript:

```
typescript
```

```
// Authorization code data structure
interface AuthCodeData {
  clientId: string;
  redirectUri: string;
  scope: string;
  codeChallenge: string;
  userId: string;
  created: number;
}

// Initialize authorization code storage
const authorizationCodes = new Map<string, AuthCodeData>();

function storeAuthorizationCode(authCode: string, codeData: Omit<AuthCodeData, 'created'>): void {
  // Add creation timestamp
  const codeDataWithTimestamp: AuthCodeData = {
    ...codeData,
    created: Date.now()
  };

  // Store code with associated data
  authorizationCodes.set(authCode, codeDataWithTimestamp);

  console.log(`[OAUTH] Authorization code generated for client: ${codeData.clientId}`);
}

function retrieveAndDeleteCode(authCode: string): AuthCodeData | null {
  // Get code data
  const codeData: AuthCodeData | undefined = authorizationCodes.get(authCode);

  if (!codeData) {
    return null;  // Code doesn't exist or already used
  }

  // Check expiration
  const age: number = Date.now() - codeData.created;
  if (age > AUTHORIZATION_CODE_LIFETIME * 1000) {
    authorizationCodes.delete(authCode);
    return null;  // Code expired
  }

  // Delete code (single-use)
  authorizationCodes.delete(authCode);
```

```
  console.log(`[OAUTH] Authorization code consumed for client: ${codeData.clientId}`);
  return codeData;
}

function cleanupExpiredCodes(): void {
  const currentTime: number = Date.now();
  const expirationTime: number = AUTHORIZATION_CODE_LIFETIME * 1000;
  let cleanedCount: number = 0;

  for (const [code, codeData] of authorizationCodes.entries()) {
    const age: number = currentTime - codeData.created;
    if (age > expirationTime) {
      authorizationCodes.delete(code);
      cleanedCount++;
    }
  }

  if (cleanedCount > 0) {
    console.log(`[CLEANUP] Cleaned up ${cleanedCount} expired authorization code(s)`);
  }
}

// Schedule periodic cleanup (every hour)
setInterval(cleanupExpiredCodes, 3600000);
```

Token Revocation Storage

Storage for revoked token identifiers (JWT jti claims):

Required Data:

- Token ID (jti from JWT)
- Expiration timestamp (when token would naturally expire)

Storage Options (Choose Based on Your Architecture):

1. **Redis** (Recommended for Production - Used in Examples)
    - Fast in-memory lookup
    - Automatic expiration via TTL
    - Simple key-value pattern: `revoked:{jti}` with TTL
    - Scales well for distributed deployments
2. **PostgreSQL/MySQL** (Good for Centralized Data)

- Table with indexed jti column
- Requires periodic cleanup of expired tokens
- Leverages existing database infrastructure

3. **MongoDB** (Good for Document-Oriented Architectures)
    - Collection with TTL index on expiration field
    - Automatic cleanup of expired documents
4. **File-Based** (Acceptable for Single-Server POC)
    - JSON file with revoked token list
    - Simple implementation
    - Requires manual cleanup logic
5. **In-Memory Set** (Development Only)
    - Fastest performance
    - Lost on server restart
    - NOT suitable for production

Key Requirement:

- Revoked tokens SHOULD persist across server restarts to prevent replay attacks

Example Pattern (Redis):

- Store as key: `revoked:{jti}`, value: `"revoked"`
- Set TTL to token's remaining lifetime
- Automatic cleanup when token expires naturally

Note: The code examples in this guide use Redis, but the patterns can be adapted to any of the storage options listed above.

Implementation Note: Redis is shown in the code examples below for its simplicity and automatic TTL support. For alternative storage implementations (PostgreSQL, MongoDB, file-based), see Part 5: Appendices for additional patterns and examples.

Implementation Approach

Token revocation storage maintains a blacklist of revoked JWT token IDs to prevent replay attacks with previously-revoked tokens. This storage must persist across server restarts to maintain security—a restarted server without the blacklist would accept revoked tokens. Redis is the recommended solution for production, providing fast lookups, automatic expiration via TTL, and scalability. For single-server POCs, file-based or in-memory storage (with documented limitations) are acceptable.

Key considerations:

- Revoked tokens MUST be checked on every authentication attempt
- Persistence across restarts prevents replay attacks
- Store only the JWT ID (jti claim), not the entire token
- Set expiration matching the token's natural lifetime (no need to store expired tokens)
- Redis TTL handles automatic cleanup—no manual expiration logic needed
- In-memory storage acceptable for development but loses data on restart

Pseudocode:

```
// Redis-based token revocation storage (recommended for production)

FUNCTION initialize_redis_connection():
    redis_client = CONNECT_TO_REDIS({
        host: REDIS_HOST,
        port: REDIS_PORT
    })

    redis_client.ON_CONNECT(FUNCTION():
        LOG("Redis connected for token blacklist")
    END FUNCTION)

    redis_client.ON_ERROR(FUNCTION(error):
        LOG_ERROR("Redis connection error: " + error.message)
    END FUNCTION)

    RETURN redis_client
END FUNCTION

FUNCTION is_token_revoked(jti):
    IF redis_client IS NOT CONNECTED:
        LOG_WARN("Redis not connected - cannot check revocation (failing open)")
        RETURN false  // Fail open if Redis unavailable
    END IF

    TRY:
        exists = REDIS_EXISTS("revoked:" + jti)
        RETURN exists == 1
    CATCH error:
        LOG_ERROR("Error checking token revocation: " + error.message)
        RETURN false  // Fail open on error
    END TRY
END FUNCTION

FUNCTION revoke_token(jti, expires_in_seconds):
    IF redis_client IS NOT CONNECTED:
        LOG_WARN("Redis not connected - revocation not persistent")
        RETURN false
    END IF

    TRY:
        // Store with automatic expiration
        REDIS_SETEX("revoked:" + jti, expires_in_seconds, "revoked")
```

```
        LOG("Token revoked: " + jti.SUBSTRING(0, 10) + "...")
        RETURN true
    CATCH error:
        LOG_ERROR("Error revoking token: " + error.message)
        RETURN false
    END TRY
END FUNCTION

// Alternative: In-memory storage (development only)
revoked_tokens = NEW_SET()  // Lost on restart

FUNCTION is_token_revoked_memory(jti):
    RETURN revoked_tokens.HAS(jti)
END FUNCTION

FUNCTION revoke_token_memory(jti):
    revoked_tokens.ADD(jti)
    LOG("Token revoked (in-memory): " + jti.SUBSTRING(0, 10) + "...")
END FUNCTION

// Initialize appropriate storage
redis_client = initialize_redis_connection()
```

JavaScript:

```
javascript
```

```
const redis = require('redis');

// Redis configuration
const REDIS_HOST = process.env.REDIS_HOST || 'localhost';
const REDIS_PORT = parseInt(process.env.REDIS_PORT || '6379', 10);

let redisClient;
let redisConnected = false;

// Initialize Redis connection
async function initializeRedis() {
  redisClient = redis.createClient({
    socket: {
      host: REDIS_HOST,
      port: REDIS_PORT
    }
  });

  redisClient.on('connect', () => {
    console.log('[REDIS] Connected for token blacklist');
    redisConnected = true;
  });

  redisClient.on('error', (err) => {
    console.error('[REDIS] Connection error:', err.message);
    redisConnected = false;
  });

  try {
    await redisClient.connect();
    console.log(`[REDIS] Initialized at ${REDIS_HOST}:${REDIS_PORT}`);
  } catch (error) {
    console.error('[REDIS] Failed to connect:', error.message);
    redisConnected = false;
  }
}

// Check if token is revoked
async function isTokenRevoked(jti) {
  if (!redisConnected) {
    console.warn('[REDIS] Not connected - cannot check revocation (failing open)');
    return false;  // Fail open if Redis unavailable
  }
```

```
  try {
    const exists = await redisClient.exists(`revoked:${jti}`);
    return exists === 1;
  } catch (error) {
    console.error('[REDIS] Error checking token revocation:', error.message);
    return false;  // Fail open on error
  }
}

// Revoke a token
async function revokeToken(jti, expiresInSeconds) {
  if (!redisConnected) {
    console.warn('[REDIS] Not connected - revocation not persistent');
    return false;
  }

  try {
    // Store with automatic expiration
    await redisClient.setEx(`revoked:${jti}`, expiresInSeconds, 'revoked');
    console.log(`[REDIS] Token revoked: ${jti.substring(0, 10)}...`);
    return true;
  } catch (error) {
    console.error('[REDIS] Error revoking token:', error.message);
    return false;
  }
}

// Alternative: In-memory storage (development only)
const revokedTokens = new Set();  // Lost on restart

function isTokenRevokedMemory(jti) {
  return revokedTokens.has(jti);
}

function revokeTokenMemory(jti) {
  revokedTokens.add(jti);
  console.log(`[MEMORY] Token revoked (in-memory): ${jti.substring(0, 10)}...`);
}

// Initialize Redis at startup
initializeRedis();
```

TypeScript:

typescript

```typescript
import redis, { RedisClientType } from 'redis';

// Redis configuration
const REDIS_HOST: string = process.env.REDIS_HOST || 'localhost';
const REDIS_PORT: number = parseInt(process.env.REDIS_PORT || '6379', 10);

let redisClient: RedisClientType;
let redisConnected: boolean = false;

// Initialize Redis connection
async function initializeRedis(): Promise<void> {
  redisClient = redis.createClient({
    socket: {
      host: REDIS_HOST,
      port: REDIS_PORT
    }
  });

  redisClient.on('connect', () => {
    console.log('[REDIS] Connected for token blacklist');
    redisConnected = true;
  });

  redisClient.on('error', (err: Error) => {
    console.error('[REDIS] Connection error:', err.message);
    redisConnected = false;
  });

  try {
    await redisClient.connect();
    console.log(`[REDIS] Initialized at ${REDIS_HOST}:${REDIS_PORT}`);
  } catch (error) {
    console.error('[REDIS] Failed to connect:', (error as Error).message);
    redisConnected = false;
  }
}

// Check if token is revoked
async function isTokenRevoked(jti: string): Promise<boolean> {
  if (!redisConnected) {
    console.warn('[REDIS] Not connected - cannot check revocation (failing open)');
    return false;  // Fail open if Redis unavailable
  }
```

```
  try {
    const exists: number = await redisClient.exists(`revoked:${jti}`);
    return exists === 1;
  } catch (error) {
    console.error('[REDIS] Error checking token revocation:', (error as Error).message);
    return false;  // Fail open on error
  }
}

// Revoke a token
async function revokeToken(jti: string, expiresInSeconds: number): Promise<boolean> {
  if (!redisConnected) {
    console.warn('[REDIS] Not connected - revocation not persistent');
    return false;
  }

  try {
    // Store with automatic expiration
    await redisClient.setEx(`revoked:${jti}`, expiresInSeconds, 'revoked');
    console.log(`[REDIS] Token revoked: ${jti.substring(0, 10)}...`);
    return true;
  } catch (error) {
    console.error('[REDIS] Error revoking token:', (error as Error).message);
    return false;
  }
}

// Alternative: In-memory storage (development only)
const revokedTokens = new Set<string>();  // Lost on restart

function isTokenRevokedMemory(jti: string): boolean {
  return revokedTokens.has(jti);
}

function revokeTokenMemory(jti: string): void {
  revokedTokens.add(jti);
  console.log(`[MEMORY] Token revoked (in-memory): ${jti.substring(0, 10)}...`);
}

// Initialize Redis at startup
initializeRedis();
```

Foundation Validation Checklist

Before proceeding to implement OAuth or MCP endpoints, verify your foundation. Each item includes testing guidance:

Server Startup:

☐ HTTP server starts successfully and listens on configured port - Test: Run server, check console for "Listening on port X" message - Test: Verify no EADDRINUSE errors

Endpoints:

☐ Server responds to basic health check endpoint (e.g., `GET /health`) - Test: `curl http://localhost:3000/health` - Expected: `{"status":"healthy","timestamp":"..."}`

Body Parsing:

☐ JSON body parsing works for POST requests - Test: `curl -X POST http://localhost:3000/test -H "Content-Type: application/json" -d '{"test":"data"}'`
☐ URL-encoded body parsing works for OAuth compatibility - Test: Send URL-encoded POST request with form data

CORS: (if not handled by platform)

☐ CORS headers configured (if not handled by deployment platform) - Test: Check response headers include Access-Control-Allow-Origin - Test: OPTIONS preflight requests return 200 OK

Logging & Errors:

☐ Request logging captures essential information - Test: Make request, verify log output includes timestamp, method, path
☐ Error handling returns properly formatted responses - Test: Send invalid request, verify error format matches endpoint type

Configuration:

☐ All required environment variables are loaded and validated - Test: Start server, check for configuration validation success message - Test: Remove required variable, verify server exits with error
☐ JWT secret is cryptographically secure (32+ bytes) - Test: Check JWT_SECRET length in validation output

Storage:

☐ Client registry storage initialized (file, database, or Redis) - Test: Check for "clients loaded" message on startup
☐ Authorization code storage initialized - Test: Verify in-memory Map is created (no errors in logs)
☐ Token revocation storage initialized - Test: Check Redis/database connection success message

Lifecycle:

☐ Server handles SIGTERM/SIGINT gracefully (if not handled by platform) - Test: Send CTRL+C, verify graceful shutdown message - Test: Check persistent data saved on shutdown
☐ Configuration can be changed without modifying code - Test: Change PORT in .env, restart server, verify new port used

---

Common Foundation Issues

Practitioners may encounter these common issues during foundation setup:

Issue: Port Already in Use

Symptom: `Error: listen EADDRINUSE: address already in use :::3000`

Solutions:

- Check if another process is using the port: `lsof -i :3000` (macOS/Linux) or `netstat -ano | findstr :3000` (Windows)
- Change PORT in .env file to an available port
- Kill the existing process using the port
- On Cloud platforms: Port conflicts are rare (platform manages ports)

Issue: Missing Environment Variables

Symptom: Server starts but configuration validation fails with specific error messages

Solutions:

- Verify .env file exists in the correct directory
- Check that dotenv is loading .env file (require('dotenv').config())
- Compare .env against .env.example for required variables
- Ensure no typos in variable names (case-sensitive)
- For Cloud deployments: Verify environment variables set in platform console

Issue: CORS Preflight Failures

Symptom: ServiceNow connection test fails with CORS policy error in browser console

Solutions:

- Verify SERVICENOW_INSTANCE URL matches exactly (including https://)
- Ensure CORS middleware is configured before route handlers
- Check that OPTIONS requests return 200 OK
- Verify Access-Control-Allow-Credentials is true
- For Cloud platforms: Check if platform CORS settings conflict with middleware

Issue: Redis Connection Failures

Symptom: "Redis connection error" or "ECONNREFUSED" in logs

Solutions:

- Verify Redis server is running: `redis-cli ping` (should return PONG)
- Check REDIS_HOST and REDIS_PORT in .env match your Redis configuration
- Ensure Redis is accessible from server (check firewall rules, network security groups)
- For local development: Install Redis if not present
- For Cloud deployments: Verify Redis instance is in same VPC/region

Issue: Client Persistence File Errors

Symptom: "Failed to save clients" or "Failed to load clients" errors

Solutions:

- Check file permissions on registered_clients.json (should be readable/writable)
- Verify directory is writable by the server process
- Check available disk space
- Ensure file path is correct (relative vs absolute paths)
- For Cloud deployments: Verify persistent volume is mounted (if using file-based storage)

Issue: JWT Secret Too Short

Symptom: Configuration validation fails with "JWT_SECRET must be at least 32 characters"

Solutions:

- Generate a secure secret: `node -e "console.log(require('crypto').randomBytes(32).toString('base64url'))"`
- Update JWT_SECRET in .env with the generated value
- Never use simple strings or passwords as JWT secrets

Issue: Body Parser Rejecting Requests

Symptom: "Request entity too large" or "Payload too large" errors

Solutions:

- Check if request payload exceeds configured limits (1MB for JSON, 100KB for URL-encoded)
- For legitimate large payloads: Increase limits in body parser configuration
- For unexpected large payloads: Investigate potential abuse or misconfiguration

Issue: Middleware Order Problems

Symptom: Requests not being logged, CORS not working, authentication failing unexpectedly

Solutions:

- Verify middleware order: CORS → Body Parsers → Logging → Routes → Error Handler
- Ensure middleware is added with app.use() before route definitions
- Check that error handler middleware has 4 parameters (err, req, res, next)
- Verify error handler is added AFTER all routes

Next Steps

Proceed to **Part 3: MCP Protocol Implementation and Tools** to implement the core MCP protocol handlers.

Document Status

- **Part:** 2 of 5
- **Version:** 1.0
- **Last Updated:** January 29, 2026
- **Status:** Complete