National College of Ireland

A novel workflow for improved access to microservices using ZKP-based methods

MSc Research Project
Cloud Computing

Abdul Wasee
Student ID: 23388790

School of Computing
National College of Ireland

Supervisor: Dr Giovani Estrada

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Abdul Wasee
Student ID:	23388790
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	09/08/2025
Project Title:	Configureation Manual
Word Count:	1744
Page Count:	13

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Abdul Wasee
Date:	12th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A novel workflow for improved access to microservices using ZKP-based methods

Abdul Wasee
23388790

1 NestJS Configuration

This section provides a comprehensive guide for configuring and setting up the NestJS-based Zero-Knowledge Identity Provider (IdP) system. The application demonstrates practical implementation of zero-knowledge proofs for privacy-preserving authentication using ZK-SNARKs and the Groth16 proving system.

1.1 Project Structure

The NestJS application follows a modular architecture optimized for cryptographic operations and zero-knowledge proof handling:

```
IdP/
  src/
    main.ts                     # Application entry point
    app.module.ts               # Root module configuration
    app.controller.ts           # Root controller
    app.service.ts              # Root service
    auth/                       # Authentication module
      auth.module.ts            # Auth module configuration
      auth.controller.ts        # API endpoints (register/proof/verify)
      auth.service.ts           # Business logic implementation
      dto/                      # Data Transfer Objects
        create-auth.dto.ts      # User registration DTO
        proof.dto.ts            # Proof generation DTO
        verify.dto.ts           # Proof verification DTO
      entities/
        auth.entity.ts          # Database entity definitions
    crypto/                     # Cryptographic utilities   <-- fixed typo
      generate-proof.ts         # ZK proof generation logic
      secret-generator.ts       # Secret and commitment generation
  circuits/                     # ZK circuit implementation
    secret-proof.circom         # Main ZK circuit (Poseidon hash)
    secret-proof.r1cs           # Compiled constraint system
    secret-proof.sym            # Symbol mapping
    secret-proof_js/            # WASM artifacts and witness calculator
  contracts/                    # Smart contract verifiers
    SecretProofVerifier.sol     # Generated Groth16 verifier contract
  prisma/                       # Database configuration
```

```
  schema.prisma                # PostgreSQL schema definition
scripts/                       # Build and utility scripts
  build-circuits.sh            # Circuit compilation automation
bench-mark/                    # Performance testing suite
  bench.js                     # Comprehensive benchmarking tool
```

Listing 1: Project Directory Structure

1.2 Core Technologies and Libraries

This section explains the key technologies and libraries used in the system, providing context for readers unfamiliar with zero-knowledge cryptography and modern web development frameworks.

1.2.1 Zero-Knowledge Proof Technologies

Zero-Knowledge SNARKs (ZK-SNARKs) are cryptographic proofs that allow one party to prove knowledge of certain information without revealing the information itself. In our system, users can prove they possess valid credentials without exposing their personal data.

- **Circom**: A domain-specific language for writing arithmetic circuits used in zero-knowledge proofs. It compiles high-level constraint logic into efficient mathematical representations.
- **SnarkJS**: A JavaScript library that enables the generation and verification of ZK-SNARK proofs in web applications. It handles the complex cryptographic operations required for proof creation.
- **Groth16**: A specific ZK-SNARK construction that provides efficient proof generation and verification. It's currently one of the most practical ZK-SNARK systems for real-world applications.
- **Poseidon Hash**: A cryptographic hash function specifically designed for efficiency in zero-knowledge circuits. Unlike traditional hash functions, Poseidon requires fewer constraints in ZK circuits.

1.2.2 Web Framework and Backend Technologies

NestJS is a progressive Node.js framework for building efficient server-side applications. It uses TypeScript by default and combines elements of functional programming, object-oriented programming, and reactive programming.

- **TypeScript**: A typed superset of JavaScript that compiles to plain JavaScript, providing better code quality and developer experience through static type checking.
- **Express.js**: The underlying HTTP server framework that NestJS builds upon, handling web requests and responses.
- **Swagger/OpenAPI**: Provides interactive API documentation, making it easier for developers to understand and test the endpoints.

- **Class Validator**: Ensures incoming data meets specified criteria through decorators, providing robust input validation.

1.2.3 Cryptographic Libraries

- **Noble Hashes**: A lightweight, security-focused cryptographic library providing various hash functions including SHA-3 and Keccak.
- **CircomLibJS**: JavaScript implementation of circuit libraries, providing building blocks for common cryptographic operations in ZK circuits.
- **Ethers.js**: A library for interacting with Ethereum blockchain, used here primarily for its BigNumber utilities and cryptographic functions.

1.2.4 Benchmark Tools

- **Autocannon & K6**: Performance testing tools for measuring HTTP server performance under various load conditions.
- **pnpm**: A fast, disk-space efficient package manager that uses symlinks to avoid duplication of dependencies.

1.3 Dependencies

The application utilizes specialized dependencies for cryptographic operations, zero-knowledge proofs, and high-performance web services:

1.3.1 Core Framework Dependencies

```
{
  "@nestjs/common": "^11.0.1",           // Core NestJS framework
  "@nestjs/core": "^11.0.1",             // NestJS application core
  "@nestjs/platform-express": "^11.0.1", // Express platform adapter
  "@nestjs/swagger": "^11.2.0",          // API documentation generation
  "swagger-ui-express": "^5.0.1",        // Swagger UI integration
  "class-transformer": "^0.5.1",         // DTO transformation
  "class-validator": "^0.14.2",          // Input validation
  "reflect-metadata": "^0.2.2",          // Metadata reflection support
  "rxjs": "^7.8.1"                       // Reactive programming utilities
}
```

Listing 2: NestJS Framework Dependencies

1.3.2 Cryptographic and ZK Dependencies

```
{
  "snarkjs": "^0.7.5",                   // ZK-SNARK proof system
  "circomlibjs": "^0.1.7",               // Circom circuit library (JS)
  "@noble/hashes": "^1.8.0",             // Cryptographic hash functions
  "sha3": "^2.1.4",                      // SHA-3 (Keccak) implementation
```

```
  "ethers": "^6.14.4"                    // Ethereum utilities and
   BigNumber
}
```

Listing 3: Zero-Knowledge and Cryptographic Libraries

1.3.3 Database and Infrastructure

```
{
  "@prisma/client": "^6.10.1",           // Prisma ORM client
  "prisma": "^6.10.1",                   // Prisma CLI and schema tools
  "lru-cache": "^11.1.0"                 // Memory caching for performance
}
```

Listing 4: Database and Infrastructure Dependencies

1.3.4 Development and Testing Dependencies

```
{
  "@nestjs/cli": "^11.0.0",              // NestJS command-line interface
  "@nestjs/testing": "^11.0.1",          // Testing utilities
  "jest": "^29.7.0",                     // Testing framework
  "ts-jest": "^29.2.5",                  // TypeScript Jest integration
  "typescript": "^5.7.3",                // TypeScript compiler
  "eslint": "^9.18.0",                   // Code linting
  "prettier": "^3.4.2",                  // Code formatting
  "autocannon": "^8.0.0",                // HTTP load testing
  "tinybench": "^4.0.1",                 // Micro-benchmarking utilities
  "pidusage": "^4.0.0"                   // Process monitoring
}
```

Listing 5: Development Tools and Testing Framework

1.4 Installation Commands

Follow these sequential installation steps to configure the complete NestJS environment with zero-knowledge capabilities:

1.4.1 Prerequisites Installation

```
# Add NodeSource repo for Node 22 LTS and install
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
sudo apt-get install -y nodejs

npm install --global corepack@latest

# Enable Corepack and activate pnpm
corepack enable
corepack use pnpm@latest-10
```

```
# Verify
node -v && pnpm -v
```

Listing 6: Ubuntu/Debian: Install Node.js 22 LTS

```
# Install Node.js 22 LTS
brew install node@22

# (Optional) Make Node 22 the default "node"
brew link --force --overwrite node@22

# Enable Corepack and activate pnpm
corepack enable
corepack use pnpm@latest-10

# Verify
node -v && pnpm -v
```

Listing 7: macOS (Homebrew): Install Node.js 22 LTS

```
# Install Node.js LTS (22) via winget
winget install -e --id OpenJS.NodeJS.LTS

# Open a new terminal, then enable Corepack and pnpm
corepack enable
corepack use pnpm@latest-10

# Verify
node -v && pnpm -v
```

Listing 8: Windows (PowerShell): Install Node.js 22 LTS

1.4.2 Project Setup and Dependencies

```
# Clone the repository
git clone https://github.com/FocusBT/idP
cd IdP

# Install all project dependencies
pnpm install
```

Listing 9: Project Installation Sequence

1.5 Development Workflow

1.5.1 Development Server Management

```
# Start development server with hot reload
pnpm run start:dev

# Start in debug mode
```

```
pnpm run start:debug

# Build for production
pnpm run build

# Start production server
pnpm run start:prod
```

Listing 10: Development Commands

1.6 Performance Benchmarking

The application includes a comprehensive benchmarking suite for evaluating zero-knowledge proof performance:

1.6.1 Benchmark Execution

```
# Ensure server is running
pnpm run start &

# Execute comprehensive benchmark suite
cd bench-mark
node bench.js
```

Listing 11: Performance Testing Commands

1.7 API Endpoints Documentation

The service exposes three POST endpoints:

- `POST /auth`
- `POST /auth/proof`
- `POST /auth/verify`

1.8 API Access

Local API Base URL

`http://localhost:3000`

Swagger UI (Interactive Docs)

`http://localhost:3000/api`

Tip: All endpoints, request/response schemas, and example payloads are available and testable in Swagger UI.

This NestJS configuration provides a robust foundation for zero-knowledge identity systems with comprehensive tooling for development, testing, and production deployment. The modular architecture ensures maintainability while optimizing for cryptographic performance and security.

2 Rust Configuration

This section provides a comprehensive guide for configuring and setting up the Rust-based Zero-Knowledge Authentication API system. The application demonstrates high-performance implementation of zero-knowledge proofs for privacy-preserving authentication using ZK-SNARKs, Groth16 proving system, and optimized cryptographic operations built with Actix-Web framework.

2.1 Project Structure

The Rust application follows a performance-optimized architecture designed for cryptographic operations and concurrent zero-knowledge proof handling:

```
zk-auth-gpy/
  src/
    main.rs                      # Main application with optimized crypto
   stack
  circuits/                      # ZK circuit implementation
    secret-proof.circom          # Main ZK circuit (Poseidon hash)
    secret-proof.r1cs            # Compiled constraint system
    secret-proof.sym             # Symbol mapping
    secret-proof_js/             # WASM artifacts and witness calculator
      secret-proof.wasm          # WebAssembly witness generator
      witness_calculator.js      # Witness calculation utilities
      generate_witness.js        # Witness generation script
    secret_final.zkey            # Groth16 proving/verifying keys
  bench-mark/                    # Performance testing suite
    bench.js                     # Comprehensive benchmarking tool
  target/                        # Rust compilation artifacts
    release/                     # Optimized production builds
      zk-auth-api               # Compiled binary executable
  Cargo.toml                     # Rust dependencies and configuration
  package.json                   # Node.js dependencies for benchmarking
  pot12_*.ptau                   # Powers of Tau ceremony files
```

Listing 12: Project Directory Structure

2.2 Core Technologies and Libraries

This section explains the key technologies and libraries used in the high-performance Rust implementation, providing context for readers unfamiliar with systems programming and zero-knowledge cryptography.

2.2.1 Rust Systems Programming Technologies

Rust is a systems programming language focused on safety, speed, and concurrency. It prevents segmentation faults and guarantees memory safety without requiring a garbage collector, making it ideal for cryptographic applications requiring both performance and security.

- **Actix-Web**: A powerful, pragmatic, and extremely fast web framework for Rust that provides high-performance HTTP server capabilities with actor-based architecture.
- **Tokio**: An asynchronous runtime for Rust that enables writing reliable, asynchronous, and fast applications with minimal overhead.
- **Serde**: A framework for serializing and deserializing Rust data structures efficiently and generically, crucial for JSON API responses.
- **Cargo**: Rust's build system and package manager that handles dependencies, compilation, and project management.

2.2.2 Zero-Knowledge Proof and Cryptographic Technologies

Arkworks is a comprehensive ecosystem of Rust libraries for zkSNARK programming, providing production-ready implementations of cryptographic primitives.

- **ark-circom**: Rust integration for Circom circuits, enabling seamless proof generation from Circom-compiled artifacts.
- **ark-groth16**: High-performance implementation of the Groth16 zk-SNARK proving system optimized for production use.
- **ark-bn254**: Implementation of the BN254 (alt_bn128) elliptic curve used in Ethereum and many ZK applications.
- **ark-ff**: Finite field arithmetic library providing optimized operations for cryptographic computations.
- **light-poseidon**: Optimized Poseidon hash function implementation designed for zero-knowledge circuits with pre-computed parameters.

2.2.3 Performance Optimization Libraries

- **once_cell**: Provides thread-safe, lazy static initialization for expensive-to-compute static values like cryptographic keys.
- **rand**: Fast and cryptographically secure random number generation with support for various RNG algorithms.
- **hex**: Efficient hexadecimal encoding/decoding for cryptographic data representation.
- **num-bigint**: Arbitrary precision integer arithmetic for handling large cryptographic numbers.
- **sha3**: Implementation of SHA-3 (Keccak) cryptographic hash function.

2.2.4 Concurrency and Resource Management

- **Semaphore**: Used for controlling concurrent access to expensive proof generation operations, preventing resource exhaustion.
- **Mutex**: Thread-safe sharing of cached Poseidon hashers across multiple request handlers.
- **spawn_blocking**: Tokio utility for offloading CPU-intensive cryptographic operations to dedicated thread pools.

2.3 Dependencies

The application utilizes specialized dependencies optimized for high-performance cryptographic operations and concurrent web services:

2.3.1 Web Framework Dependencies

```
[dependencies]
# Web framework and serialization
actix-web = "4"                    # High-performance async web framework
serde = { version = "1", features = ["derive"] }  # Serialization framework
serde_json = "1"                   # JSON serialization support
tokio = "1.46.1"                   # Async runtime
```

Listing 13: Actix-Web Framework Dependencies

2.3.2 Cryptographic and ZK Dependencies

```
[dependencies]
# Arkworks cryptographic stack
ark-bn254 = "0.5"                 # BN254 elliptic curve implementation
ark-ff = "0.5"                     # Finite field arithmetic
ark-circom = "0.5"                 # Circom circuit integration
ark-groth16 = "0.5"                # Groth16 proving system
ark-snark = "0.5"                  # General SNARK traits

# Optimized hash functions
light-poseidon = "0.3"             # BN254 Poseidon parameters
sha3 = "0.10"                      # SHA-3 (Keccak) implementation
```

Listing 14: Zero-Knowledge and Cryptographic Libraries

2.3.3 Performance and Utility Dependencies

```
[dependencies]
# Performance optimization
once_cell = "1.19"                 # Lazy static initialization
rand = { version = "0.8", features = ["small_rng"] }  # Fast RNG
num_cpus = "1.17.0"                # CPU core detection
```

```
# Utility libraries
hex = "0.4"                          # Hexadecimal encoding/decoding
num-bigint = "0.4"                   # Arbitrary precision arithmetic
```

Listing 15: Performance Optimization Libraries

2.3.4 Development and Benchmarking Dependencies

```
{
  "autocannon": "^8.0.0",           // HTTP load testing framework
  "axios": "^1.11.0",               // HTTP client for API testing
  "circomlib": "^2.0.5",            // Circom circuit library
  "pidusage": "^4.0.1"              // Process monitoring utilities
}
```

Listing 16: Node.js Dependencies for Benchmarking

2.4 Installation Commands

Follow these sequential installation steps to configure the complete Rust environment with zero-knowledge capabilities:

2.4.1 Prerequisites Installation

```
# Rust (stable, non-interactive)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"

# Node.js 22 LTS (Homebrew)
brew install node@22
brew link --force --overwrite node@22

# pnpm via Corepack
corepack enable
corepack use pnpm@latest-10

# Verify
node -v && pnpm -v && rustc --version && cargo --version
```

Listing 17: macOS: Rust

```
# Rust (stable, non-interactive)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"

# Node.js 22 LTS (NodeSource)
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
sudo apt-get install -y nodejs
```

```
# pnpm via Corepack
corepack enable
corepack use pnpm@latest-10

# Verify
node -v && pnpm -v && rustc --version && cargo --version
```

Listing 18: Linux/Ubuntu: Rust

```
# Run in an elevated PowerShell

# Rustup and Node.js 22 LTS
winget install -e --id Rustlang.Rustup
winget install -e --id OpenJS.NodeJS.LTS

# Open a new terminal so PATH updates apply, then:
corepack enable
corepack use pnpm@latest-10

# Verify
node -v; pnpm -v; rustc --version; cargo --version;
```

Listing 19: Windows (PowerShell): Rust

2.4.2 Project Setup and Dependencies

```
# Clone the repository
git clone https://github.com/FocusBT/zk-auth-rust
cd zk-auth-rust

# Build Rust dependencies (debug mode)
cargo build

# Build optimized release version (recommended for performance)
cargo build --release

# Install Node.js dependencies for benchmarking
pnpm install

# Verify installation
ls -la target/release/zk-auth-api
```

Listing 20: Project Installation Sequence

2.5 Development Workflow

2.5.1 Development Server Management

```
# Start development server with debug logging
RUST_LOG=debug cargo run
```

```
# Start development server
cargo run

# Build and start optimized release server
cargo build --release
./target/release/zk-auth-api

# Enable verbose logging for debugging
RUST_LOG=trace ./target/release/zk-auth-api

# Background server execution
./target/release/zk-auth-api &
```

Listing 21: Development Commands

2.5.2 Benchmark Execution

```
# Build and start optimized server
cargo build --release
./target/release/zk-auth-api &

# Execute comprehensive benchmark suite
cd bench-mark
node bench.js

# Monitor system resources during benchmarking
htop    # or your preferred system monitor
```

Listing 22: Performance Testing Commands

2.6 API Endpoints Documentation

The Rust application exposes three high-performance endpoints for zero-knowledge authentication.

2.6.1 User Registration Endpoint

```
curl -X POST http://localhost:8080/register \
  -H "Content-Type: application/json" \
  -d '{
    "email": "alice@example.com",
    "name": "Alice Smith",
    "age": 25,
    "country": "US",
    "dob": "1998-01-15"
  }'
```

Listing 23: Registration API Request

```
{
  "secret": "0x1a2b3c4d5e6f7890abcdef1234567890...",
  "nonce": "0x9876543210fedcba",
  "commitment": "12345678901234567890123456789012345678"
}
```

Listing 24: Registration Sample Response (JSON)

2.6.2 Zero-Knowledge Proof Generation

```
curl -X POST http://localhost:8080/generate-proof \
  -H "Content-Type: application/json" \
  -d '{
    "secret_hex": "0x1a2b3c4d5e6f7890abcdef1234567890...",
    "commitment": "12345678901234567890123456789012345678"
  }'
```

Listing 25: Proof Generation API Request

```
{
  "proof": {
    "a": ["0x...", "0x..."],
    "b": [["0x...", "0x..."], ["0x...", "0x..."]],
    "c": ["0x...", "0x..."]
  }
}
```

Listing 26: Proof Generation Sample Response (Groth16 JSON)

2.6.3 High-Speed Proof Verification

```
curl -X POST http://localhost:8080/verify-proof \
  -H "Content-Type: application/json" \
  -d '{
    "commitment": "12345678901234567890123456789012345678",
    "proof": {
      "a": ["0x...", "0x..."],
      "b": [["0x...", "0x..."], ["0x...", "0x..."]],
      "c": ["0x...", "0x..."]
    }
  }'
```

Listing 27: Verification API Request

```
{
  "valid": true
}
```

Listing 28: Verification Sample Response (JSON)

HTTP semantics: 200 OK for valid proofs; 401 Unauthorized for invalid proofs.