Project Report: Local Identity & Access Management (IAM) Implementation using Keycloak and Docker

Author: [Your Name]
Date: September 29, 2025

Executive Summary

This report documents the process of designing, implementing, and testing a local Identity and Access Management (IAM) server. The project's primary objective was to create a self-contained, developer-friendly authentication environment using Keycloak, a leading open-source IAM solution. The entire system was containerized using Docker to ensure portability, consistency, and ease of setup. The project successfully configured a new authentication realm, registered a sample client application, created a test user, and validated the end-to-end OpenID Connect (OIDC) authentication flow, culminating in a successful user login and redirection.

1. Introduction

In modern application development, building a secure and scalable authentication and authorization system is a complex and critical task. Handling user registration, login, password management, and access control from scratch is not only time-consuming but also prone to security vulnerabilities. Identity and Access Management (IAM) solutions address this challenge by providing a centralized, robust framework for managing user identities and controlling their access to resources.

This project focuses on **Keycloak**, an open-source IAM product sponsored by Red Hat. It provides out-of-the-box support for standard protocols like **OAuth 2.0**, **OpenID Connect (OIDC)**, and **SAML 2.0**, making it an ideal choice for securing web applications, mobile apps, and APIs.

The core goal was to leverage Docker to run Keycloak locally. This container-based approach decouples the IAM server from the host operating system, allowing for a clean, reproducible setup that can be easily shared and managed, which is invaluable for development and testing environments.

2. Technology Stack and Architecture

2.1 Keycloak

Keycloak served as the central IAM server. Its key responsibilities in this project included:

- **User Federation:** Storing and managing user credentials
- **Authentication Server:** Presenting a login interface and verifying user credentials
- **Authorization Server:** Issuing access tokens (like JSON Web Tokens - JWTs) after successful authentication, which client applications can then use to grant access

2.2 Docker

Docker was the containerization platform used to run the Keycloak instance. The benefits of using Docker in this context were:

- **Isolation:** The Keycloak server and its dependencies run in an isolated environment (a container), preventing conflicts with other software on the local machine
- **Portability:** The same Docker image can run on any machine with Docker installed, guaranteeing a consistent environment
- **Simplified Setup:** A single `docker run` command is sufficient to download, configure, and start the entire server, abstracting away complex installation procedures

2.3 Project Architecture

The authentication flow established in this project follows these steps:

1. **User Initiates Login:** The user attempts to access a protected resource. For this project, this was simulated by directly accessing a specially crafted Keycloak authentication URL
2. **Redirect to Keycloak:** The application (represented by the `my-web-app` client) redirects the user to the Keycloak login page
3. **User Authentication:** The user enters their credentials (`testuser`/`password123`) on the Keycloak page. Keycloak validates these credentials
4. **Redirection with Code:** Upon successful authentication, Keycloak redirects the user back to the application's pre-configured `redirect_uri` (`https://www.google.com`), including a temporary authorization code in the URL
5. **(Conceptual Next Step):** A real application would then exchange this code with Keycloak to receive an ID token and an access token, completing the login process

3. Detailed Implementation Steps

3.1 Step 1: Environment Preparation - Docker Installation

The foundational step was to set up the containerization environment. Docker Desktop for Windows was installed, which provides the Docker Engine, CLI, and a management UI. After installation, the following command was run to confirm that the Docker client was installed and could communicate with the Docker daemon:

```
docker --version
```

A successful output displaying the Docker version confirmed the environment was ready.

3.2 Step 2: Instantiating the Keycloak Server Container

With Docker running, the official Keycloak image was downloaded and executed with the following command:

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin quay.io/k
```

Command Breakdown:

- `docker run`: The fundamental command to create and start a new container from an image
- `-p 8080:8080`: This flag maps the host machine's port 8080 to the container's internal port 8080. This makes the Keycloak server accessible via `localhost:8080`
- `-e KEYCLOAK_ADMIN=admin` and `-e KEYCLOAK_ADMIN_PASSWORD=admin`: These flags set environment variables inside the container for the initial administrator account
- `quay.io/keycloak/keycloak:latest`: Specifies the Docker image to use from the container registry
- `start-dev`: Starts Keycloak in development mode, which simplifies setup and is ideal for testing

3.3 Step 3: Initial Access and Realm Configuration

Once the container was running, the Keycloak Admin Console was accessed by navigating to `http://localhost:8080/` and logging in using the administrator credentials (`admin`/`admin`).

The first administrative task was to create a **Realm**. A realm in Keycloak is a logical space that manages a set of users, credentials, roles, and clients. It provides isolation, allowing a single Keycloak instance to manage multiple separate applications or organizations (multi-tenancy).

A new realm named `my-test-app` was created to ensure that the users and application configurations for this project were completely separate from the default `master` realm used for Keycloak administration.

3.4 Step 4: Client Application Registration

The next step involved registering the conceptual application with Keycloak. In IAM terminology, an application that uses Keycloak for authentication is called a **Client**.

1. Inside the `my-test-app` realm, a new client was created in the "Clients" section
2. **Client ID:** The ID was set to `my-web-app` - a unique identifier used in authentication requests
3. **Valid Redirect URIs:** This crucial security setting is a whitelist of URLs to which Keycloak is allowed to redirect a user after successful login. This was set to `https://www.google.com` for simulation purposes. In a real-world scenario, this would be a specific callback URL in the application

3.5 Step 5: User Creation

To test the login flow, a non-administrator user was needed:

1. Within the `my-test-app` realm, the "Users" section was accessed and a new user was added
2. A user with the username `testuser` was created
3. After creating the user, the "Credentials" tab was used to set an initial password, `password123`, with the "Temporary" switch turned off so the user wouldn't be forced to change it on first login

3.6 Step 6: Testing the End-to-End Authentication Flow

The final step involved simulating an authentication request from the client application (`my-web-app`) for the user (`testuser`). A specific OpenID Connect (OIDC) authentication URL was constructed and opened in the browser:

```
http://localhost:8080/realms/my-test-app/protocol/openid-connect/auth?client_id=my-web-app
```

URL Parameter Breakdown:

- `.../realms/my-test-app/.../auth`: Specifies the authentication endpoint for the specific realm
- `client_id=my-web-app`: Identifies which application is making the request
- `response_type=code`: Specifies the OIDC "Authorization Code Flow," the standard and most secure flow for web applications
- `scope=openid`: A required scope for OIDC, indicating that the application wants to verify the user's identity
- `redirect_uri=https://www.google.com`: Tells Keycloak where to send the user back after login

Upon accessing this URL, the Keycloak login page was correctly presented. After entering the credentials for `testuser`, Keycloak successfully redirected the browser to `https://www.google.com`, confirming that the entire flow was configured correctly.

4. Challenges and Troubleshooting

During the project implementation, a common issue was encountered:

"Invalid parameter: redirect_uri" Error: Initially, a typo was made in the `redirect_uri` parameter in the URL. Keycloak correctly displayed an error page, refusing to proceed. This highlighted the importance of strict URI matching for security. The issue was resolved by ensuring the URL parameter perfectly matched the value in the client configuration.

5. Key Learning Outcomes

This project provided comprehensive hands-on experience with core IAM concepts and modern development practices:

- **Centralized Authentication:** Understanding the role of a centralized authentication server in modern architectures
- **Containerization Benefits:** Experiencing the power and convenience of Docker for setting up complex services
- **OIDC Protocol:** Learning the fundamental mechanics of the OpenID Connect protocol
- **Security Best Practices:** Implementing proper redirect URI validation and secure credential handling

6. Conclusion and Future Work

This project was successfully completed, demonstrating the deployment of a fully functional IAM server locally using industry-standard tools like Keycloak and Docker. The implementation provided practical experience with essential IAM concepts, including realms, clients, users, and the OIDC authentication flow.

Future Enhancement Opportunities

The current setup provides an excellent foundation for further exploration:

1. **Real Application Integration:** Develop a simple web application (using Node.js/Express or Python/Flask) and integrate it with Keycloak using adapter libraries to secure endpoints
2. **Role-Based Access Control (RBAC):** Define roles within Keycloak (e.g., `user`, `admin`), assign them to users, and enforce access rules in client applications based on those roles
3. **Multi-Factor Authentication (MFA):** Enhance security by configuring MFA (using Google Authenticator) for users within the realm
4. **Docker Compose Implementation:** Use `docker-compose.yml` to define the Keycloak service and potentially a database for persistent storage, making the entire stack manageable with a single command
5. **Production Deployment:** Explore deployment strategies for production environments, including SSL/TLS configuration, database persistence, and scalability considerations

This project successfully demonstrates the practical implementation of modern identity management solutions and provides a solid foundation for building secure, scalable authentication systems.