MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Identity and access management system

Master's Thesis

BC. MILOŠ HEGR

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Identity and access management system

Master's Thesis

BC. MILOŠ HEGR

Advisor: Ing. Lukáš Grolig

Department of Computer Systems and Communications

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools: Grammarly for grammar checking and ChatGPT to improve my writing style. I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it

Bc. Miloš Hegr

Advisor: Ing. Lukáš Grolig

Acknowledgements

I would like to thank Lukáš Grolig for his inspiring ideas and continuous motivation. Additionally, I am deeply grateful to my family and friends, especially to Klárka Boštíková, for unwavering support, encouragement, and patience throughout the creation of this thesis.

Abstract

Modern software applications increasingly rely on secure and flexible identity and access management (IAM) systems to protect sensitive data and ensure proper authorization. Despite the availability of robust open-source IAM solutions such as Keycloak, Zitadel, and Ory, there remains a significant gap in the .NET ecosystem, particularly following the deprecation of IdentityServer4.

This thesis addresses the need for a comprehensive, extensible, and developer-friendly IAM system tailored for .NET developers, by proposing a new IAM solution called CleanIAM. The CleanIAM project is an open-source IAM system built on the OpenIddict framework, providing full compliance with OAuth 2.0 and OpenID Connect protocols. It implements essential IAM features, including user and client management, single sign-on (SSO), multi-factor authentication (MFA), and role-based authorization. The system is designed with modern architectural principles, such as vertical slice and clean architecture patterns, alongside event-driven design, to ensure modularity, scalability, and ease of customization using native C#.

Keywords

IAM, Single sign-on, OAuth, OpenID Connect, Authorization, User management, Event-driven architecture

Contents

1 Introduction **1**

2 State Of Art **2**
2.1 Existing IAM Solutions Analysis 2
2.2 Keycloak 3
2.3 Zitadel 4
2.4 Ory 5
2.5 Identity Server 4 7
2.6 Analysis summary 9

3 Identity and access management **12**
3.1 Single Sign On 12
3.2 Multi-Factor Authentication 14
3.3 User Management 15
3.4 User Revocation 16
3.5 Client Management 16

4 OAuth 2.0 **18**
4.1 Roles 18
4.2 Client types 19
4.3 Authorization grants 20
4.4 Authorization Code Flow In Depth 22
4.5 Access Token Types 27
4.6 Proof Key for Code Exchange technique (PKCE) 28
4.7 Refresh Token 30

5 Design **31**
5.1 Requirements 31
5.2 Architecture Design 34
5.3 Event Storming 38
5.4 Application UI Design 39

6 Implementation **41**
6.1 Wolverine 41
6.2 Databases 42
6.3 Password Storage 44

6.4 Project structure 46
6.5 Error handling 49
6.6 User Interface 50
6.7 Deployment 51
6.8 Testing 51
6.9 Documentation 52

7 Future Plans **54**

8 Conclusion **56**

A SSO auth flow example diagram **58**

B Use case diagram for CleanIAM project **59**

C Event Storming for CleanIAM project **60**

Bibliography **62**

List of Figures

3.1 SSO flow diagram 13

4.1 Client redirection URI example 24
4.2 Redirect URI example 25
4.3 Access token request example 26
4.4 Access token response example 27

5.1 Clean architecture diagram [27] 37
5.2 UI wireframes for the sign-in flow 40

6.1 The CleanIAM root directory structure 46
6.2 The CleanIAM slice directory structure 47
6.3 Command and command handler example structure . . . 48
6.4 Body of error response generated by custom Result type. 50

1 Introduction

Modern applications increasingly rely on secure and flexible identity and access management (IAM) systems. However, within the .NET ecosystem, the most popular solution, Identity Server 4, has moved to commercial licensing, making the open-source version deprecated. This left developers with limited open-source options. While robust solutions like Keycloak, Zitadel, or Ory are available in the wider open-source ecosystem, they are either not customized for easy .NET use, lack easy extensibility using C#, or impose constraints that do not allow complete UI or logic customization.

This thesis addresses the lack of a comprehensive, modern, and developer-friendly IAM solution in the .NET ecosystem. While multiple IAM frameworks are available, this work demonstrates that none offer the same level of integration, simplicity, and extensibility specifically for .NET developers as the solution proposed in this thesis. The solution proposed in this thesis is an open-source IAM system built using the OpenIddict stack, designed to provide OAuth2 and OpenID Connect compliance, user and client management, single sign-on, and multi-factor authentication, all while being easily modifiable and customizable in C# and deployable as a standalone service.

While it is not the primary goal of this thesis to contribute to the theoretical discussion of identity and access management, this work provides a thorough analysis of existing open-source IAM solutions and presents an in-depth overview of the OAuth 2.0 protocol and related standards. This theoretical background serves to clarify the fundamental concepts and security mechanisms, reinforcing modern authentication and authorization systems. Building on this foundation, the thesis identifies key functional and architectural criteria that guide the design of the proposed IAM solution tailored for the .NET ecosystem. The proposed IAM system design is used as a foundation to implement a modern, extensible, and developer-friendly platform that meets the specific needs of .NET developers.

2 State Of Art

Identity and access management is an important part of almost all applications, but each application has a specific workflow and needs, and the requirements for IAM systems differ.

For those reasons, it is good to have open-source, easily customizable, and modifiable solutions. In this chapter, an analysis of some of the most used open-source IAM systems will be conducted, and their key features, strengths, and limitations will be examined.

Missing features will be identified by evaluating the analyzed IAM systems against industry standards, best practices, and specific use case requirements.

2.1 Existing IAM Solutions Analysis

To better understand what the current solutions of IAM implementation look like, we can analyze existing solutions. The solutions analyzed in this section were chosen based on the following criteria:

Open-sourced The solution must be open-source, meaning its source code is publicly available, allowing users to inspect, modify, and contribute to its development.

Widely adopted The solution must be widely adopted, meaning it is used by many organizations or developers, indicating its reliability, stability, and trustworthiness.

Active community support The solution must have active community support and ongoing maintenance, meaning a dedicated group of contributors regularly updates the project, fixes security vulnerabilities, improves features, and addresses issues.

Based on those criteria, Keycloak, Zitadel, and Ory were selected for further analysis. IdentityServer4 will be compared to these selected solutions at the end of this chapter to evaluate how it measures up in terms of functionality, extensibility, and suitability for modern identity and access management scenarios.

The analysis of the solutions will focus on several key aspects of IAM solutions, including customizability, extensibility, GitHub star count, and programming language.

Security features such as compliance with OAuth 2.0, OpenID Connect standards, and MFA support will also be considered. Furthermore, the quality of documentation will be assessed to determine how well the solution is supported for development and integration.

The ease of deployment and configuration will be a crucial factor, as well as integration capabilities with cloud providers, databases, and external identity providers. Lastly, the analysis will also consider the licensing model, identifying whether the solution is governed by an MIT, Apache, or GPL license and the implications these licensing models have for the usage.

2.2 Keycloak

Keycloak is a widely used open-source identity and access management solution developed by Red Hat and is currently part of the Cloud Native Computing Foundation (CNCF). It provides robust authentication, authorization, and identity federation features, supporting standards like OAuth 2.0, OpenID Connect, and SAML 2.0.

Keycloak is designed to be highly customizable via a plugin system. It also supports custom themes for the login user interface. However, its high customizability can require significant development work since the plugin system and creating a custom theme flow must be learned [1].

Based on a GitHub star rating, this is the most popular among the three selected solutions. Keycloak GitHub repository currently has over 25 thousand stars. [2]

This solution is written in Java, and Java expertise is required to extend the Keycloak solution. It runs on top of Quarkus, a Kubernetes-native Java framework, which enhances its scalability and performance in cloud environments.

Keycloak supports OAuth 2.0, OpenID Connect, and SAML 2.0, with built-in MFA options like TOTP, WebAuthn, and SMS and email OTP. It enables custom authentication flows, user federation with

LDAP or Active Directory, and Kerberos-based Windows SSO. With strong protocol compliance and early adoption of new standards.

It offers extensive official documentation, including an administration guide, server developer guide, and user guide on its website. The documentation covers setup, configuration, and extension points. However, some parts of the documentation can be outdated or unclear for readers, based on the author's experience.

It can be run as a standalone server or via the official Docker image. Configuration can be done through a web admin console, via configuration files, or environment variables for automation. It also supports Kubernetes deployment using a prepared Keycloak Operator.

It can integrate with external identity providers such as Google, GitHub, Facebook, or other identity providers implementing OIDC or SAML. Keycloak also supports LDAP and Active Directory as federation sources, as mentioned above.

Keycloak is fully open source and licensed under the Apache License 2.0. This is a permissive license, meaning Keycloak can be used, modified, and distributed freely, even in commercial products.

The Keycloak project is actively developed. In April 2023, it was donated to the Cloud Native Computing Foundation as an incubating project, bringing in more contributors and integrating with the cloud-native community. [3, 2, 4]

2.3 Zitadel

Zitadel is a modern, cloud-native IAM solution designed for scalability and ease of use. Zitadel allows custom logic through Zitadel Actions, which are scriptable event handlers that run code in JavaScript in response to defined triggers, such as the occurrence of specific events. It also supports branding, allowing customization of login screens. However, only certain properties can be configured, making it less customizable than the Kycloack solution.

Zitadel solution currently has almost 10 thousand stars on GitHub, making it less popular than Keycloak but still showing significant usage. [5]

This IAM system is written in Go, which compiles to a single binary for deployment. For the admin console, UI creators decided to use

TypeScript. For integration purposes, Zitadel offers SDKs and API clients in multiple languages since customizing Zitadel via code can require deep Go and Zitadel expertise.

Zitadel supports all the major protocols. It is fully compliant with OAuth 2.0 and OpenID Connect, implements core and optional specifications, and supports SAML 2.0 for federation. It offers multi-factor authentication options out of the box, OTP via email or SMS, TOTP for OTP codes, and even built-in OTP delivery via authenticator apps.

Zitadel documentation is well-structured and developer-friendly. It provides quick-start guides, an official documentation portal with how-to guides, and a well-structured API reference. In comparison with Keycloak, the documentation seems to be well-maintained and up-to-date.

Deploying Zitadel is quick and easy. Also, the authors advertise on their GitHub repository that you can get it running in under 3 minutes for a local setup [5]. Zitadel can be run as a single binary, locally in Docker, or in a Kubernetes cluster supporting helm chart configuration. Zitadel also offers a managed cloud service called Zitadel Cloud.

Zitadel is built as an API-first system, meaning almost every function is exposed via gRPC or REST APIs, allowing complex integration. It provides SDKs in multiple languages, like Go, Java, C#, and others, to facilitate integrating applications or backend services with Zitadel.

Zitadel is an open-source project licensed under the Apache License 2.0. This means the community edition of Zitadel is free to use and modify with the same freedoms as Keycloak. Zitadel also has an enterprise license for a premium self-hosted offering with some additional features, such as custom thread analysis or custom metrics.

The rapid growth of the GitHub star count suggests that the Zitadel community is quickly growing. The project is actively developed and maintained, reacting to new security threats. [6, 5, 7]

2.4 Ory

Ory is a modern IAM solution with a focus on modularity, scalability, and flexibility. Ory architecture is different from previous monolithic solutions. It is a collection of applications, each handling separate aspects of identity and access management. Ory provides the follow-

ing four main applications: Hydra for OAuth2/OIDC, Kratos for user management, Keto for authorization, and Oathkeeper for access proxy. Hydra doesn't have authentication or UI at all. It is supposed to be integrated into a custom login application using a well-defined REST interface. This allows the creation of a completely custom user interface for authentication.

Ory solution is developed in multiple GitHub repositories. The most favorite is the Hydra repository, which handles OAuth and OpenID standards and has 16 thousand stars, suggesting that Ory is also a widely used IAM solution. [8]

Like the Zitadel, the core Ory components are written in Go. Since Ory provides extensive REST and gRPC interfaces, Go knowledge is not required. However, since Ory is open-source, developers can extend or edit its source code, and for that, extensive Go knowledge is required.

Ory's platform is built with compliance with security standards as a top priority. Ory Hydra has been officially validated for compliance, gaining certification from the OpenID Foundation as an OAuth 2.0 Authorization Server and an OpenID Connect Provider. Ory Kratos provides features like multi-factor authentication and passwordless login, TOTP, Passkeys, recovery codes, and social logins via OAuth2/SAML.

Ory provides extensive documentation on their official site. The documentation explains every part of Ory's ecosystem in detail. This is also the reason why it might be challenging and overwhelming for developers to start with Ory. For this reason, Ory offers "quick start" guides to help developers quickly understand the basic concepts of this ecosystem.

Ory's components are designed to be cloud-native and easy to deploy. Each application is a single Go binary that can be run via the command line or as a Docker container. Since Ory is not just a single application, as was the case for previous solutions, it requires more configuration and setup to reach a usable state. This is balanced by an extensive description of deployment possibilities in Ory's documentation.

Ory is integration-friendly, largely due to its headless and API-first design. It can integrate with any existing user database or identity provider because Ory Hydra is not itself an identity store, but it is connected to an identity system via a login and consent application.

Ory's open-source components are licensed under Apache License 2.0, allowing the use, modification, and distribution of its source code freely, even in commercial products, the same as it was for all solutions discussed before. Ory also offers hosted products and some enterprise features.

Ory's projects are currently under active development and have strong support in the open-source community. [9, 8, 10]

2.5 Identity Server 4

IdentityServer4 is not a standalone IAM solution. Rather, it is a framework that can be integrated into .NET applications to implement authentication and authorization.

Since IdentityServer4 is just a framework, developers can use it to build their own IAM solution. This gives developers full control over their IAM solution and allows them to make a fully custom design and implement any IAM feature needed. However, this means that IdentityServer4 is not plug-and-play, and developers are responsible for writing and maintaining any customizations within their application's codebase.

IdentityServer4 was focused on the community in the .NET ecosystem. It was popular among ASP.NET Core developers, with many downloads via NuGet and a decent GitHub presence. The original project repository had several thousand stars on GitHub. However, since the original repository was archived, then removed from the public, and then brought back as an archived repository, the current archived repository has only around 30 GitHub stars, and the .NET community is trying to find an up-to-date alternative. [11]

IdentityServer4 is written in C#. It's essentially a set of .NET Core libraries that developers include in an ASP.NET Core application. Thus, its primary language is C#, and it runs on the .NET runtime. Because of its origin, IdentityServer4 is suitable exclusively for .NET applications.

IdentityServer4's security features are focused on OAuth2 and OpenID Connect protocol compliance. It provides the fundamental building blocks for authentication and authorization in an ASP.NET Core context, issuing access and ID tokens, validating clients, and other

authentication and authorization features. IdentityServer4 itself does not handle user management. Features like password policies, MFA, or account lockout are expected to be handled by the ASP.NET Identity system or another user store that developers decided to integrate it with. IdentityServer4 ensures tokens are issued securely and follows the OIDC specification. This was officially certified by the OpenID Foundation for compliance. However, since this project is no longer maintained, it has multiple known security vulnerabilities and bugs that are never going to be fixed.

IdentityServer4 documentation is well-structured and easy to read. It also provides a series of quickstart tutorials with code examples, making it easier for developers to start with IdentityServer4. Since there is a big community around IdentityServer4, there are many community articles written, helping developers understand the base functionalities and even solving specific needs of some projects. However, some parts of the official documentation can be outdated since this project is no longer under maintenance.

Since IdentityServer4 is just a framework, it cannot be run on its own. Rather, it is run as part of the custom application that the developers implemented. Because of this, deploying IdetityServer4 is fully dependent on the final application deployment. Configuration of identityServer4 is based on .NET conventions, and with the use of quickstart guides from its documentation, it can be easy and straightforward for developers with basic ASP.NET knowledge.

Because of the IdentityServer4 nature, it is designed exclusively for integration into custom applications. Developers can use prepared functions, middleware, and other constructs to create user flows required by their use cases. It can be integrated with any database or also with none at all, since clients and resources can be defined directly in code or in configuration files.

IdentityServer4 is licensed under Apache License 2.0, joining the previously discussed solutions. However, its successor Duende IdentityServer[1] moved to the commercial license, and now it is only free for small-scale or non-production uses. Projects using the deprecated version of IdentityServer4 have to comply with Apache License 2.0. [12]

1. `https://duendesoftware.com/products/identityserver`

With the end of .NET Core 3.1 in 2022, the IdentityServer4 repository was archived, and it is no longer under development, and no maintenance has been performed since. It is not recommended to build a new .NET project around IdentityServer4, and for this reason, there is now an empty space with the IAM solutions in the .NET world. [13, 11, 12]

2.6 Analysis summary

When comparing modern open-source identity and access management solutions like Ory, Keycloak, Zitadel, and IdentityServer4, it's clear that they are focused on offering flexibility and extensive integration options to accommodate the different needs of different developer projects. Selected solutions share important features for authentication and authorization, like implementing OAuth and OpenID Connect specifications or basic user management features.

The analyzed solutions differ considerably in several aspects. From an architectural perspective, each takes a distinct approach to system design. Keycloak and Zitadel are distributed as standalone binaries, which simplifies deployment and setup. In contrast, Ory adopts a modular, headless architecture, prioritizing flexibility and customizability. IdentityServer4, on the other hand, is a framework rather than a plug-and-play service, offering deep integration possibilities with the enclosing application. Since the new solution should be as easy as possible to deploy, it should take the same path as Keycloak and Zitadel and support easy deployment via Docker images.

Overall, the current IAM ecosystem features a diverse selection of open-source solutions, each designed to address distinct needs and priorities, except for the .NET world, where there is currently no go-to solution.

In the .NET world, IdentityServer4 was a widely used solution that developers were used to. Because of IdentityServer4 .NET, developers were used to integrating IAM solutions directly into their projects. However, with the end of support for IdentityServer4, there is currently no such widely used solution.

There is one solution that seems to have gained traction after the end of IdentityServer4, called OpenIddict[2]. OpenIddict is the OpenID Connect stack that helps .NET developers implement their own authentication and authorization systems that comply with OAuth and OpenID standards. However, this OpenIddict stack is not a complete IAM solution but rather just an OAuth 2.0 and OpenID Connect library. The OAuth 2.0 and OpenID Connect part of the OpenIddict seems to be well tested and actively developed. This makes OpenIddict a viable solution for authentication. [14]

Based on previous IAM solution analyses, the new proposed IAM solution should have all the basic IAM features that all the analyzed solutions have in common, such as compliance with OAuth and OpenIddict standards, or basic user management. Other specific features, such as OAuth client management or multifactor authentication, should also be implemented. In addition, the new solution should be open-source, and customization should be prioritized to enable developers to change the solution to their needs.

One of the biggest issues with the reviewed solutions was that they used custom solutions for extension. For this reason, extending the new IAM solution should be possible with a native language, in this case, C#. This makes it easier for developers to implement new features since they don't have to learn new syntax and semantics of a proprietary language. The frontend part should also be written in a commonly known language for easy design customizability, to avoid problems like Zitadel has, that it is nearly impossible to fully customize login screens, as mentioned in section 2.3.

All mentioned solutions have a public GitHub repository, allowing developers to read through code, catch bugs and security issues, and propose fixes via pull requests. This can help keep the project up to date, secure, and attract more developers to use this specific solution. For this exact reason, the new proposed solution should be available as a public GitHub repository.

The deeper requirements analysis will be conducted in chapter 5, where all design decisions will be explained.

The remaining part of this thesis will describe important authentication and authorization functionalities in depth. Finally, a new IAM

2. `https://openiddict.com`

solution for the .NET world will be proposed based on the OpenIddict stack. Since it was already mentioned that OpenIddict is actively maintained and has already implemented and tested many security standards, the new solution will use the OpenIddict stack for the OAuth and OpenID Connect parts of the project. Another reason for choosing the OpenIddict stack will be further discussed in chapter 6. This new solution will aim to replace deprecated IdentityServer4 and thus fill the empty spot in the IAM solutions written in .NET.

3 Identity and access management

Identity and Access Management is a fundamental component of modern cybersecurity, ensuring that only authorized users and systems can access specific resources. IAM is a set of policies, technologies, and processes designed to manage digital identities and regulate access to sensitive data and applications.

A robust IAM framework provides authentication, authorization, and accountability mechanisms that help organizations enforce security policies and minimize the risk of unauthorized access. This includes identity provisioning and deprovisioning, role-based access control, multi-factor authentication, single sign-on, and privileged access management.

Modern IAM solutions leverage technologies such as biometrics, artificial intelligence (AI), and behavioral analytics to enhance security and detect anomalies in real-time. Additionally, IAM is crucial for regulatory compliance, helping organizations adhere to industry standards such as GDPR, HIPAA, and ISO 27001.

3.1 Single Sign On

As internet users, we visit many online websites and systems every day. Most of them require us to authenticate to them using a username and password. For our safety, we should use different passwords on every one of these websites, meaning we need to remember many passwords, resulting in using simple and easy-to-guess passwords or even reusing them. This makes it easy for cybercriminals to steal user accounts.

Single sign-on (SSO) is an identity solution that solves exactly this problem. It enables multiple applications to share the same authentication session, eliminating the need for users to enter their various sign-in credentials repeatedly. [15]

In the case of SSO from the point of security, there are fewer passwords the attacker can steal, and more importantly, multi-factor authentication, explained in section 3.2, can be used to protect this single point of authentication even more.

Third-party identity providers can be used to authenticate users for even better user usability and security. Because of this approach, when the user is authenticated to the application implementing SSO, besides the authentication using username and password, it can be authorized using third-party systems such as Google Identity, Microsoft Entra ID, Okta, and many others. [16]

Single sign-on is usually implemented as a separate system handling authentication for all of the other applications. Suppose the application needs to authenticate the user. The user is redirected to the SSO system, where it is authenticated. Based on the configured policies, multiple authentication methods, including the already mentioned third-party identity providers or multifactor authentication, can be used. After the authentication, the user is redirected back to the application with the authentication secret, for this flow can be used OAuth 2.0 which is more deeply explained in chapter 4.

When another application wants to authenticate the user, after redirecting to the SSO system, it has already authenticated the user, so no credentials or any other actions are needed. The user is redirected back to the second application as a successfully authenticated user.

This single sign-on principle is graphically explained in the following figure 3.1, and a detailed graphical explanation of SSO can be found in appendix A.

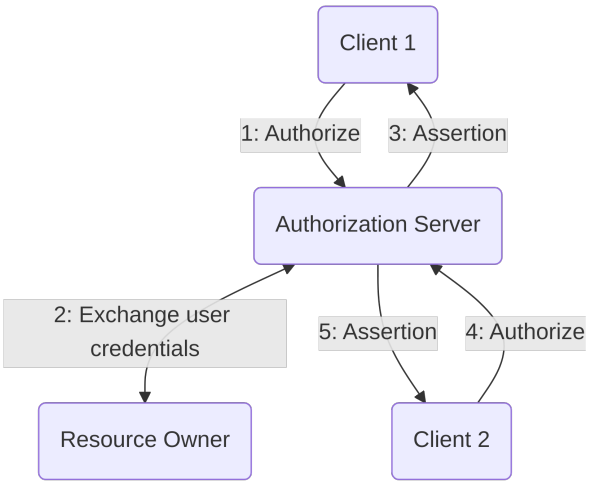


Figure 3.1: SSO flow diagram

3.2 Multi-Factor Authentication

In the current world, users are used to authenticating with their credentials, most commonly passwords. Since SSO unites signing to a single application, the user's account should be protected with more security layers than just a password.

Multi-factor authentication (MFA) adds multiple additional steps to the authentication flow to prevent unauthorized users from gaining access to a user account even when the password has been stolen.

The factors that can be added to the authentication flow can be split into three main types based on the interaction with the user:

Knowledge

This first type groups together factors that are based on information only the user knows. A typical example of this factor is passwords or answers to personal security questions. This factor type is the most common since the password factor is used in many applications. This factor type is secure only as long as no one else knows the secret information.

Possession

Possession factors are based on the fact that the user owns something unique. This category can be further divided into two sections, physical assets and digital assets.

Physical assets are typically electrical devices that the user owns, like hardware keys, display cards, or mobile phones. These physical devices provide secret information to users without the need to remember anything. In the case of mobile phones, one-time passwords (OTP) can be sent via SMS, for example.

Digital assets are typically some applications that provide secrets to the user. Typical digital assets are, for example, sending one-time passwords to the user's email or using authenticator applications that generate simple time-limited codes the user can use to pass this authentication step.

Inference

Inference factors are based on what the user is. Inference factors are mostly based on the user's biometrics. Typical examples of

these factors are fingerprint scans, facial recognition, retina scans, or behavioral analysis of, for example, keystroke dynamics.

These factors can be combined to make the authentication flow secure enough for the client's use case. With each additional factor, the authentication flow is more secure, but it makes it less friendly to the users. Because of that, it is important to find the balance between the two properties.

Some factors can be required only under specific conditions, to make it easier for users. Machine learning and artificial intelligence can help with monitoring the user authorization activity and deciding when some factor will be required. Other factors, such as location, time, and others, can also make the authentication process more secure and intuitive. [17, 18]

3.3 User Management

Identity and access management systems, besides user authentication and authorization, provide an equally important feature called user management. User management allows selected users, such as system administrators.

The important feature of user management is the ability to create new user accounts. This can be done in multiple ways, for example, by the user themselves filling out the registration form or by system administrators creating accounts for other users.

When creating a new user account, some possession authorization factor, described in the previous section 3.2, should be used. This factor should verify that the user really exists and prevent the robot from creating too many spam accounts.

Another important feature is the capability to edit existing accounts. This can be done in two ways. Users are usually allowed to update their personal information, such as passwords or other details. This mostly depends on IAM system policies. In a second way, selected users, such as system administrators, can edit the personal info of other people and, most importantly, assign roles and permissions to other users.

Revoking user accounts is also an important part of user management. This topic will be described in the following section 3.4.

3.4 User Revocation

There are several cases/scenarios where the user account should be revoked, such as when the user no longer wants to use the system or is forbidden to use it. Based on the scale for user account revocation, there are three types. In some cases, user revocation is called identity deprovisioning.

Soft revocation, also called suspension, is temporarily disabling user access to the system without removing their identity.

Hard revocation, also called deletion, completely removes all user data, such as user credentials, permissions, and identity. This process is irreversible and

The last type is the user session, terminating all active sessions to ensure that user access is removed. This type is typically part of the two previously mentioned types. This process mostly involves deleting active user tokens. Based on the token types, this can be immediate session termination in case of reference tokens described in the sections 4.5, but in case of self-encoded tokens described in section 4.5 the termination cannot be immediate because of the principle of this token type.

User revocation doesn't have to be initiated only by the user themselves or the administrators, but can also be triggered by automated revocation mechanisms. An example of the mechanisms can be event-driven revocation, where the revoking of users is initiated after a certain event has occurred, such as revoking users based on HR changes like offboarding automation. Another example of automated user revocation mechanisms can be time-based expiry, where the user is only temporary and is deleted after a certain period. Users can also be revoked automatically based on audit-trigged revocation, deleting dormant or risky users. [19, 20]

3.5 Client Management

A client in IAM systems refers to an application, service, or systems that interact with IAM systems. What IAM systems clients are and what their role is is deeply explained in section 4.2.

From the point of view of the IAM client management, the important feature is creating new clients. Usually, creating new clients is allowed only for administrators. They need to provide information such as client name, client ID, redirect URLs, and others. These attributes are better described in sections 4.1 and 4.2.

When creating a new client, the correct authorization grant type must be chosen. Authorization grants are explained in section 4.3. Different authentication methods are used for this grant type.

Based on systems politics, clients' data can also be updated if the situation requires that. This may include modifications to redirect URLs, authentication methods, or access permissions to ensure compliance with security requirements and evolving business needs. Proper logging and auditing mechanisms should be in place to track changes in client configurations.

Client deletion is another important aspect of IAM client management. Deleting a client typically involves revoking its credentials, removing its access permissions, and ensuring that dependent applications or services are not disrupted. Some systems implement a soft-delete mechanism, where deleted clients can be restored within a certain period before permanent deletion. Proper auditing should be in place to log deletion events, helping maintain security and compliance with access management policies.

In most IAM systems, creating, updating, or removing a client requires administrative privileges to prevent accidental or unauthorized actions. [21]

This chapter covered the key features of Identity and Access Management, including SSO, MFA, user management, and management of OpenID Connect applications. The next chapter will focus on the base standard for authorization called OAuth 2.0.

4 OAuth 2.0

One of the most essential parts of the IAM systems is authorization. The base standard for the authorization is OAuth 2.0 [22]. This chapter will map key aspects of this protocol, which will be further used in the application design.

OAuth 2.0 is an authorization framework defining protocols and best practices for a third-party application to obtain limited HTTP service access.

This authorization framework was released to the public in 2012. It is described in RFC 6749 [23]. Since the release of the original OAuth 2.0 specification, many new RFCs, adding new features or editing original specifications, have been released.

OAuth 2.0 and its updated specifications describe many details about the authorization flows, components, syntax, and semantics. In this chapter, I will go through the most important concepts in those specifications.

4.1 Roles

The OAuth 2.0 specification describes four main roles that interact during the course of the complete OAuth 2.0 flow. In this section, I will describe all of them.

Resource Owner

This role describes an entity capable of granting access to a protected resource. This is typically the end-user.

Resource Server

A resource server hosts the protected resources and is capable of accepting protected resource requests using access tokens.

Client

The client is an application that makes the request for protected resources on behalf of the `Resource Owner`.

Authorization Server

The server issues access tokens to the `Client` after successfully authenticating the `Resource owner` and obtaining proper authorization.

The specification defines that the Resource Server and Authorization Server may be the same server or a separate entity.

4.2 Client types

As described in section 4.1, the client is an application that accesses data from the resource server on behalf of the resource owner. Depending on the type of application, we can define two types of clients.

The client types are based on the client's ability to authenticate securely to the authorization server and on the exposure levels of client credentials.

Each client is identified by its client ID assigned by the authorization server. In the case of a confidential client, it can also have its own client secret with whom the client is able to authenticate against the authorization server.

When registering a client with authentication using any of the redirection flows described in section 4.3, allowed redirect URIs must be provided for security reasons. [24]

Confidential client

This type of client can have secret information embedded in its source code without the possibility that an end-user can extract it. This is because the end-user never comes into contact with the source code.

An example of this application can be a back-end server providing a public API hosted on our own machine. Due to this fact, this type of client can easily prove to the authorization server its real identity, thus making the authorization process more secure.

Public client

On the other hand, the public client type defines an application whose source code can be accessed and read by the end-user. This makes it

impossible to hide any secret in the source code, and thus, the client cannot easily prove its identity to the authorization server.

An example of a public client is a webpage written in JavaScript. When the user loads the page in his browser, all the source files are transferred to him, and we cannot stop him from exploring the source code and extracting the possible secrets.

Another example of a public client is a mobile app, which brings problems similar to a web page application. But it also adds some of its own problems that developers must solve. These problems will be explained in the following sections.

4.3 Authorization grants

An authorization grant is a properly defined flow of obtaining access tokens by an OAuth client. The OAuth 2.0 specification describes exactly four such flows. Each flow is suited for different use cases based on the client type described in section 4.2.

Authorization Code

This flow describes how a client can obtain an access token without ever getting in contact with the resource owner's credentials.

Instead of authorizing the resource owner directly from within the client. The resource owner is redirected to the authorization server. Here, the resource owner is authenticated, and after successful authentication, the resource owner is redirected back to the original client with the authorization code, with which the client can request the final access token from the authorization server.

Because the resource owner authenticates only with the authorized server, the client never comes into contact with the resource owner's credentials.

This flow is ideal for use with public-type clients, explained in section 4.2. For security reasons, combining this flow with the PKCE technique described in section 4.6 is recommended.

Implicit

The implicit grant type is the simplified version of the previously described authorization code flow. The main difference between these two flows is how the authorization server provides the final access token to the client. In implicit flow, the access token is provided as part of the client URI, to which the resource owner is redirected after successfully authenticating to the authorization server.

This flow has never been secure from the beginning. The reason why this flow exists is the historical fact that, in the past, web browsers didn't allow cross-origin requests. Thus, the last step of acquiring the access token in the authorization code flow was not possible.

Even though this grant type is defined in the original OAuth 2.0 specification, nowadays, it is considered deprecated, and it is recommended to use the authorization code grant type instead.

Resource Owner Password Credentials

This grant type requires the resource owner to provide credentials directly to the client. After obtaining credentials, the client makes an HTTP request to the authorization server. The authorization server validates the received credentials and responds to the client with either an access token when the credentials validation is successful or an error otherwise.

Because this flow makes the client access the resource owner's credentials directly, it should be used only when the client is a highly trusted application and flows based on redirection are not possible.

Even though this grant type allows the client to access the resource owner credentials directly, they are only used for a single request to obtain the access token. After that, all the following requests use the obtained access token so the client doesn't have to store the credentials.

Client Credentials

The client credential flow is based on exchanging the client credentials, such as client ID and client secret, for an access token. This requires the client to be able to handle its own secrets.

The client typically uses this grant type when it acts on its own behalf, and it doesn't require authorization from a specific resource owner.

This flow is ideal to use with confidential client type, described in section 4.2 since this type of client is able to store its secrets securely.

4.4 Authorization Code Flow In Depth

In this section, I will go through each step of the authorization code grant type to better explain how the client obtains an access token without ever seeing resource owner credentials.

Only the authorization code flow for web applications will be described since this is one of the main use cases of this flow. It is also used for mobile applications, but differences in authorization code flow for those two application types are unimportant for this thesis.

Throughout this flow, there are four actors, each of them described in section 4.1.

This flow can be divided into two phases. The first phase uses front-channel communication, meaning that all the data transmitted are in the query part of requests and visible to anyone watching this conversation. The second phase communicates through a back-channel, which means that data are transmitted securely in the body part of the HTTP request.

Step One

In the first step of the authorization code flow, the resource owner initiates the flow, for example, by clicking the login button. The official flow does not require this step because the client can initiate it without user interaction.

Step Two

In the second step, the client redirects the resource owner to the authorization server. The authorization server has defined a specific endpoint to which all users are redirected. This endpoint usually has a `/authorize` route. Redirection URI created by the client must contain

the required parameters in the query component. These parameters must be in `application/x-www-form-urlencoded` format. OAuth defines the following parameters:

response_type [Required]

This parameter specifies the type of token returned in the authorization request. In this case, the value of this parameter must be "`code`".

client_id [Required]

Client ID identifies the client making the current request. Each client has assigned its own unique ID by the authentication server, as explained in section 4.2.

redirect_uri [Optional]

The redirect URI parameter specifies where the authorization server should redirect the resource owner after successful authentication. The value of this parameter must match the redirect URIs defined at the time of client registration at the authentication server, as mentioned in section 4.2.

scope [Optional] This parameter defines the scope of access the client wants to acquire. The value of this parameter is defined as a space-separated list of strings, each defining a specific scope.

state [Recommended]

The state parameter contains an opaque value that the server includes when redirecting back to the client. The value of this parameter is recommended to be random. When the resource owner is redirected back to the client, the client should check if the state value provided by the authorization server in response is the same value sent in the initial redirect URI. This check can prevent Cross-site request forgery (CSRF) attacks.

The example of the final redirection URI created by the client can be seen in Figure 4.1.

```
https://authserver.com/auhtorize?
    response_type=code&
    client_id=exampleClientId&
    state=asdf&
    redirect_uri=https%3A%2F%2Fclient.com%2Fcallback
```

Figure 4.1: Client redirection URI example

Step Three

In the third step, the resource owner interacts directly with the authorization server. First, the resource owner must authenticate with the authorization server. This can be done by providing the resource owner's credentials or by providing valid tokens obtained in the previous authentication.

After the resource owner is successfully authenticated and all additional steps, like two-factor authentication or giving permission to the client, are finished, the resource owner is redirected back to the client.

Same as in the previous step, this redirection URI has a specific format and required parameters. The location where the resource owner should be redirected is defined by the `redirect_uri` parameter from the previous step. The authorization server must check if this URI matches any of the redirect URIs specified at the client registration as explained in section 4.2. If this check is successful, parameters in `application/x-www-form-urlencoded` encoding are appended to this URI, and the resource owner is redirected back to the client. Required URI parameters for this step are the following:

code The authorization code is issued by the authorization server, which can be later exchanged for the access token. This token can be used only once and should have a short lifetime.

state If the state parameter was provided in the previous step. The authorization server must set this parameter to the exact same value.

The example of the final redirection URI created by the authorization server is shown in Figure 4.2.

```
https://client.com/callback?
    code=WXKe1lr96hElU2stu&
    state=asdf
```

Figure 4.2: Redirect URI example

After this step is finished, if all verifications have been passed, the client has successfully obtained the authorization code.

Step Four

The fourth step initiates the second phase of the authorization code flow. The client tries to exchange the obtained authorization code for the final access token. This is done by making a POST-type HTTP request to the token endpoint on the authorization server. This endpoint usually has a route ending `/token`. Data are transferred in the body part of this HTTP request with `application/x-www-form-urlencoded` encoding. Since this request is sent via HTTPS, the data is securely encrypted. This request has the following parameters:

grant_type [Required]

Parameter defining the used flow. In this case, it must have a value equal to `authorization_code`.

code [Required]

Value of the authorization code received in the previous step.

redirect_uri [Required]

This parameter is required only if the `redirect_uri` value was provided in the redirection URI from step two in section 4.4.

client_id [Required]

Value identifying the client making the request.

This simplified version of an access token request using HTTP could look as shown in Figure 4.3.

```
POST /token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=WXKe1lr96hElU2stu&
redirect_uri=https%3A%2F%2Fclient.com%2Fcallback&
client_id=exampleClientId
```

Figure 4.3: Access token request example

Step Five

In the last step, the authorization server verifies the received code token and other values included in the client's request. After successful validation, the new access token is created and returned to the client. Part of the response can be a refresh token, explained in section 4.7. The response should be in `application/json` formatting and contain the following values:

access_token The final access token the client uses for all following requests.

token_type Describes the type of the access token. Most commonly set to `Bearer` value.

expires_in If the access token has an expiration time. This value indicates the number of seconds after which the access token will expire.

refresh_token If the refresh token flow is enabled, the authorization server will issue the refresh token together with the access token. Refresh token flow is explained in section 4.7.

additional_values The authorization server can also include other values if required.

An example of the access token response can be seen in Figure 4.4.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"dr7Nk55Gn82MD8BBwTRCJy",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"HmzRn3H3C8fy8yoDP7cwDa",
}
```

Figure 4.4: Access token response example

After the client obtains the access token, it can be used to authenticate the resource owner in future requests to the resource server.

4.5 Access Token Types

Based on the token usage, access tokens can be divided into two families. Reference tokens and self-encoded tokens. Each family has its advantages and disadvantages. In this section, I will describe each family and compare them.

Reference tokens

The most important feature of reference tokens is that the string value of this token doesn't mean anything. These tokens are just random opaque strings. This token becomes meaningful only when combined with the authorization server. Reference tokens can be seen as a pointer to the meaningful values somewhere in the authorization server.

Generally, these types of tokens tend to be shorter because no information is encoded in them. This also means that it is impossible to tell if the token is valid without consulting the authorization server.

The biggest advantage of this token family is that the entire logic is located at the authorization server, and no key has to be distributed to the clients to verify those tokens.

On the other hand, every time the client needs to validate a token, which is typically at least once per request, clients have to ask the authorization server for validation. This adds more network tips and adds to the request response time.

Another advantage is the possibility of immediately revoking tokens since only the authorization server holds control over them.

Self-encoded tokens

A family of self-encoded tokens is a type of token that is represented by self-contained data that actually means something by itself, the opposite of reference tokens.

Since the token is self-contained and has data encoded in it, the clients don't need to contact the authority server to verify the token's validity and origin. On the other hand, this also means that these tokens tend to be much longer than reference tokens.

The most used self-encoded token type is JSON Web Token (JWT). This type of token has a strict structure, which is recognized by many applications.

A small disadvantage of self-encoded tokens is that it is impossible to revoke already issued tokens. Because already-issued tokens normally turn invalid only after they expire. This can also be a problem when these types of tokens are stolen. Because of that, these tokens' lifetime is generally shorter.

4.6 Proof Key for Code Exchange technique (PKCE)

The Proof Key for Code Exchange technique is an extension for the authorization code flow described in section 4.3 to prevent attacks such as CSRF[1] or authorization code injection attacks.

PKCE was initially created to protect authorization code flows in mobile applications, but it proved helpful in other applications. Using

1. https://owasp.org/www-community/attacks/csrf

PKCE with every public type client using the authorization code flow is recommended today.

The main goal of PKCE is to ensure that the client making the authorization code request is exactly the same as the client making the access token request. This is done by adding special parameters to both requests.

Before the authorization code flow is initiated, the client generates a secret called a code verifier and stores it in persistent storage. The code verifier must be a high-entropy cryptographic random string with a minimum length of 43 characters and a maximum length of 128 characters.

Then, a code challenge string is derived from the previously generated code verifier. The PKCE specification defines that the algorithm used to transform the code verifier to the code challenge should be SHA256. SHA256 is a hash algorithm, meaning it is hard to find out what input was used to produce the desired output. This algorithm can be too computationally demanding. The PKCE specification allows skipping the SHA256 algorithm for devices with insufficient computation power.

In step two of the authorization code grant, described in section 4.4, PKCE adds two parameters:

code_challenge The string derived from the code verifier

code_challenge_method The method used to derive the code challenge from the code verifier.

When the authorization server creates the code token, it must associate those two additional parameters with the token by including them directly in the token itself or storing the data on the server and linking them to the code token. If these additional parameters are stored in the token, they must be stored in encrypted form, so no other entities can extract those values from the token.

In the fourth step of the authorization code flow, the client sends the original code verifier with the code token and other parameters. The authorization server must check if applying the code challenge method linked to this code token produces output equal to the code challenge. If those two values match, the authorization server can be

more confident that the client requesting the access token is the same client that received this code token.

Even though the PKCE adds only three values, it significantly helps catch authorization code injection and CSRF attacks.

4.7 Refresh Token

Since access tokens usually have defined expiration dates, after some time, there is a need to acquire new ones. For security reasons, using shorter expiration times for access tokens is recommended, and it would be impractical to let users authorize so often. Because of these problems, the OAuth 2.0 specification defines a special type of token called a refresh token.

The refresh token is an opaque token allowing the client to request a new access token when the old one has expired. The client can do this without needing the resource owner to interact. Because of this accruing, the new access token can be seamless from the user's point of view.

The client acquires the access token with the access token using one of the flows defined in section 4.3. Specifically, in the case of the authorization code access grant, the refresh token is returned by the authorization server in the access token response as explained in step five described in section 4.4.

The new access token is acquired with the HTTP request to the token endpoint described in step five in section 4.4. This request must contain `grant_type` parameter with value `refresh_token`. If creating a new access token is successful, the authorization server will include the new access token and possibly a new authorization token in the response.

5 Design

When developing a new application, it is important to focus not only on functionality but also on its structure and overall design. This chapter discusses the design process of the new custom IAM solution. It outlines the essential requirements, both functional and non-functional, and explains the architectural decisions that influence the implementation.

The following parts of this thesis are associated with the new custom IAM solution, which is part of this thesis. For this reason, the text will refer directly to the IAM solution by its name to make clear to the reader that it refers to a specific solution, and thus it will be referred to as the CleanIAM project. All the information and files corresponding to the CleanIAM project can be found on the CleanIAM GitHub profile[1].

5.1 Requirements

This section will go through all the requirements for the CleanIAM project. Each requirement will be described, and acceptance criteria will be provided.

Chapter 2 provided some insight into the current solution of IAM systems. Based on this, a list of requirements will be presented, as well as a use case diagram in the appendix B.

Functional Requirements

FR-1: OAuth 2.0 and OIDC Core Complience

Description: The system shall implement the essential OAuth 2.0 and OpenID Connect protocols to provide standard authentication and authorization capabilities. This includes supporting the Authorization Code flow with PKCE extension and the Client Credentials flow for service-to-service authentication.

1. `https://github.com/CleanIAM`

Acceptance Criteria: The system supports Authorization Code flow with PKCE. The system supports Client Credentials flow. The system issues access tokens and ID tokens. The system validates tokens and scopes. The system allows token introspection for confidential clients.

FR-2: Basic User Management

Description: The system shall provide basic user account management functionality. Users must be able to register new accounts, sign in using their credentials, and reset their passwords when needed. Users must also be able to edit their basic profile information.

Acceptance Criteria: Users are able to register and sign in. The system implements password reset functionality. Users are able to change their usernames and other basic profile information.

FR-3: Multi-Factor Authentication

Description: The system shall support at least one additional authentication factor beyond passwords to enhance security.

Acceptance Criteria: The system provides an MFA authentication option. The system provides functionality to enable or disable MFA per user. The system must implement an MFA enrollment process.

FR-4: Client Application Management

Description: The system shall support the registration and management of client applications that will interact with the IAM service. Both confidential clients and public clients, described in section 4.2, must be supported.

Acceptance Criteria: The system supports the registration of both confidential and public clients. The system provides functionality to manage allowed redirect URIs. The system allows for enabling and disabling clients. It shall support setting allowed scopes per client.

FR-5: External Identity Provider

Description: The system shall support at least one external identity provider for login, such as Google or Microsoft identity services.

Acceptance Criteria: The system integrates with at least one external identity provider. The system supports linking external accounts to local user accounts.

FR-6: Role-Based Authorization

Description: The system shall implement basic role management for authorization purposes. Administrators should be able to create and manage roles and assign roles to users.

Acceptance Criteria: The system allows for the creation and management of roles. The system supports assigning roles to users. The system returns the role information in tokens.

FR-7: Basic Audit Logging

Description: The system shall log critical security events for auditing purposes. This includes logging all authentication attempts, both successful and failed, administrative actions, and token operations.

Acceptance Criteria: The system logs all authentication attempts. The system logs all administrative actions. The system logs all token operations.

Non-Functional Requirements

NFR-1: Security Essentials

Description: The system shall implement fundamental security practices to protect sensitive information. All communications should use HTTPS encryption, and passwords should be stored using a secure hashing algorithm.

Acceptance Criteria: The system uses HTTPS for all communications. The system implements secure password storage using a secure hashing algorithm. The system includes protection against CSRF and XSS attacks. The system performs input validation on all endpoints.

NFR-2: .NET Integration

Description: The system shall integrate seamlessly with the .NET ecosystem. It must be compatible with the latest .NET LTS version.

Acceptance Criteria: The system is compatible with the latest .NET LTS version. The system supports dependency injection patterns.

NFR-4: Documentation

Description: The system shall provide sufficient documentation for developers to effectively extend, customize, and use the solution.

Acceptance Criteria: The system is sufficiently documented. The system includes an installation and setup guide. The system provides basic API documentation. The system includes sample applications demonstrating common scenarios.

NFR-5: Extensibility and Customization

Description: The system shall be designed to be easily extensible and customizable. The architecture should allow developers to extend functionality without modifying the core system.

Acceptance Criteria: The system provides configuration options for customizing system behavior without code changes. The system exposes well-defined extension points for common customization scenarios.

5.2 Architecture Design

Based on the research conducted in chapter 2, the following features were selected as key features that the CleanIAM project will focus

on. The most important feature is easy customizability, allowing developers to quickly change the solution to their needs. Extensibility is another key feature of the new solution, allowing developers to easily add new features missing in the core implementation. And the last key feature the new implementation will focus on is user-friendly configuration. This makes it easy for users to orient themselves in the CleanIAM project and perform the required operations.

The architecture of the CleanIAM project will be designed using multiple architectural patterns, which will be detailed in the following sections. The final architecture will take the form of a modular monolith, combining the simplicity and ease of deployment characteristic of monolithic architectures with the benefits of separation of concerns and loose coupling from microservices. This approach ensures seamless development while maintaining the flexibility to extract individual modules into standalone microservices when needed, thanks to the loose coupling between components. [25]

Vertical Slices

Typical layer architecture in a .NET application can get confusing and overwhelming when the project starts growing. For that reason, the CleanIAM will not go this way. Instead, this solution will use an architecture pattern called vertical slices.

The vertical slice architecture approach structures an application by domains rather than technical layers. Making it easy to orient in the project, even with the incoming new features. This method allows each feature to be developed and tested in isolation, leading to better maintainability and scalability.

Each slice encapsulates all aspects of a specific domain, including the user interface, business logic, core entity definitions, and data access. This approach contrasts with traditional architectures that typically segregate an application into horizontal layers. This allows the design of each slice separately. Each slice can also be designed differently from others if the use case requires it. [26]

The slices into which the CleanIAM project is divided will be discussed in section 5.3, where the project's exact structure will be proposed.

Clean Architecture

In the CleanIAM project, each slice will be subsequently designed with clean architecture, allowing easy orientation in the final version of the project.

Clean Architecture is an architectural pattern that helps organize the code to emphasize the separation of concerns, making it easier to maintain and extend the application over time.

The core concept behind clean architecture is to separate the application into different layers, each with a clear responsibility. The primary goal is to ensure that the business logic is decoupled from the infrastructure and frameworks, thus providing flexibility to adapt to different technologies or requirements without affecting the core functionality.

In clean architecture, the Entities layer forms the core, containing business logic through domain models or entities that define the data and rules of the application and are free from external dependencies like databases or UI. The Use Cases layer, also called the application layer, follows, outlining the specific business rules of the application and managing interactions between entities, acting as a bridge between core logic and external elements. The Interface Adapters layer manages the conversion of data between the internal domain model and external systems. The outermost layer, the Frameworks and Drivers layer, also called the Infrastructure layer, includes the technical components such as databases, web frameworks, third-party libraries, and other services, which can be swapped out without impacting the core business logic. [27]

The following Figure 5.1 shows interactions between each layer and demonstrates how each layer is separated.

Event Driven Architecture

To further enhance scalability, responsiveness, and maintainability, the CleanIAM project will also adopt the Event-Driven Architecture (EDA) design pattern. This architectural pattern facilitates asynchronous communication between components, allowing the system to react quickly to events as they occur.

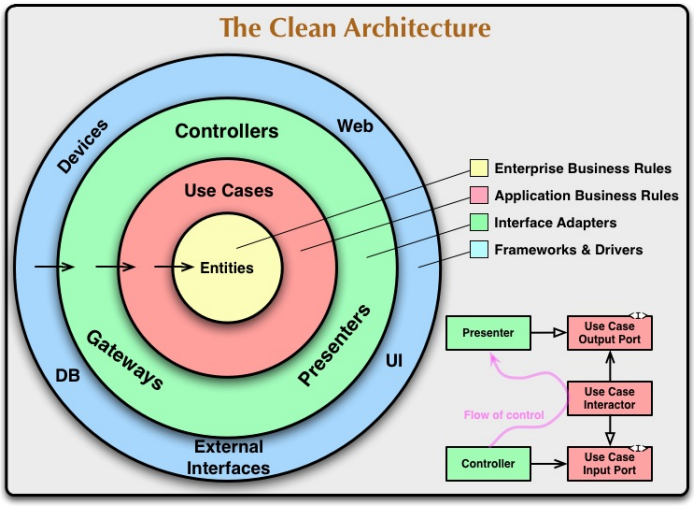


Figure 5.1: Clean architecture diagram [27]

In Event-Driven Architecture, components of a system communicate by producing and consuming events. An event represents a change in the state of the application, such as a result of some user action or a system update. Instead of tight coupling of the components, EDA allows for loose coupling by enabling components to interact through events, making the system more flexible and scalable. [28]

This design pattern nicely aligns with the vertical slices architecture, since the communication between slices can be done with events and thus removing tight coupling between slices. Each slice can operate independently, triggering and responding to events, which allows them to remain decoupled while still interacting with other parts of the application.

This decoupling ensures that each slice can evolve and be extended without impacting the rest of the system, contributing to the CleanIAM project's key features of customizability and extensibility. Furthermore, by using events as the communication mechanism, new features can be added as separate slices that listen for specific events, making it easier to scale the application as new requirements appear.

5.3 Event Storming

Event Storming is a collaborative and interactive modeling technique used to explore and design systems by focusing on the events that drive business processes. It helps stakeholders, developers, and domain experts come together to visualize and understand the flow of information and the underlying processes within a system. By modeling events, commands, and aggregates, Event Storming provides a shared understanding of how different parts of the system interact, which is crucial for designing complex applications like the CleanIAM project. [29]

The appendix C represents a product of event storming for the CleanIAM project. The figure shows all of the events generated in the system, and what aggregates each event is associated with. For the event, the colors are important since each color represents a different entity. The color orange represents domain events, the color blue represents commands, and the color yellow represents aggregates associated with events.

The figure C also shows how the CleanAIM project will be divided into multiple slices. The events storming output contains pivotal events indicating the borders of bounded contexts. Bounded contexts represent sections of the project operating on a particular domain model. In the case of the CleanIAm project, each bounded context will represent a single slice.

The event storing divided the CleanIAM project was divided into five slices:

Identity Slice handling OAuth and OIDC flows

Applications Slice managing OIDC applications

Users Slice handling user management

Scopes Slice handling OIDC scope management

Tenants Slice managing tenants

The event storming effectively decoupled the CleanIAM project into multiple bounded contexts, ensuring each slice operates independently with its own set of business logic and domain model. This

structure facilitates a more modular and maintainable system, where changes to one slice do not directly impact others. Each slice, while maintaining clear boundaries, communicates with other slices through well-defined events, ensuring loose coupling and enabling scalability.

5.4 Application UI Design

Since one of the goals of this thesis was to make it easy to modify solution UI design isn't as important as it could be for other projects. If the main goals of this thesis are accomplished, the developer can easily customize the design to their needs.

However, even if the UI design isn't the primary focus, the application still needs to be designed to allow pleasant interaction with users. A well-structured and intuitive user interface enhances the user experience, making it easier for users to navigate and perform tasks.

The application UI can be divided into two sections. The first section represents the UI for Identity slice implementation, providing the SSO for other applications. This section is designed to be as simple as possible, to ensure that users can authenticate seamlessly without unnecessary complexity. The simplicity of the Identity UI is represented by wireframes in Figure 5.2. The second section of the UI handles management, users, application scopes, and tenants, as well as per-user configuration. This section contains mostly tables with given entities and allows users to invoke actions on them.



Figure 5.2: UI wireframes for the sign-in flow

6 Implementation

The most important part of this thesis is the actual implementation of the CleanIAM project. This section follows up on the previous chapter 5, where application design was discussed. This section will describe the integration of the design into the actual code and also show the most important or interesting parts of the CleanIAM implementation.

As discussed in the section 2.6, the CleanIAM project is implemented in the C# language using the .NET framework. The CleanIAM project leverages a modular architecture that emphasizes maintainability, scalability, and clean separation of concerns. The implementation follows modern software engineering practices such as Domain-Driven Design (DDD), Command Query Responsibility Segregation (CQRS), and event-driven communication, some of which were already described in previous sections, and others will be discussed later.

This chapter outlines the technical decisions made during the development of the CleanIAM project, focusing on the tools, libraries, and patterns used. It aims to give insight into how the architecture described in section 5.2 is translated into actual code and systems. Key areas such as messaging with WolverineFx, database integration, security mechanisms like password hashing, deployment strategies, and testing are discussed in detail. By examining these implementation details, the chapter demonstrates how theoretical design principles are applied to build a robust and flexible identity and access management system. The deeper explanation of each part of the CleanIAM project implementation is described in the project's documentation, which is part of the main project repository[1].

6.1 Wolverine

As was discussed in the chapter 5, the ClainIAM project is designed as an event-driven architecture, focusing on creating and handling events. This idea has to be propagated to the code to correspond to the project design. To leverage all of these features, the CleanIAM solution

1. `https://cleaniam.github.io/CleanIAM/`

uses the messaging framework called WolverineFx[2]. The WolverineFx framework provides a message bus abstraction for easy and clean event operations.

The WolverineFx also allows for easier implementation of the command query responsibility segregation (CQRS) pattern, making the CleanIAM project even more event-driven and also cleaner and easier to maintain. WolverineFx provides automatic message routing and scheduling, as well as support for retry policies, dead-letter queues, and error handling strategies. This ensures that transient failures do not lead to data loss and that messages can be retried or handled appropriately when issues arise. Additionally, Wolverine offers native integration with ASP.NET Core, enabling straightforward configuration of middleware and message handlers.

Overall, WolverineFx plays a central role in enabling the CleanIAM project to follow modern architectural patterns, making the system more modular, resilient, and easier to evolve over time.

6.2 Databases

The persistent data storage is an important part of most applications, and it is not different for the CleanIAM project. Present data storage allows application to preserve their data across multiple sessions or even between application restarts.

On the lowest level, the CleanIAM project uses a PostgreSQL database, which provides solid and stable data persistence. Since the PostgreSQL database is a widely used solution, it has wide user support, is well documented, and is easy to use and scale [30]. This gives the project a solid base for future extensibility and customizability.

On the higher level project uses two database frameworks. The first framework is `Entity Framework`. It is a widely used object-relational mapper (ORM) for .NET that enables developers to work with a database on the level of .NET objects, eliminating the need to write raw SQL. The project relies on Entity Framework because it serves as the underlying data access layer for the OpenIddict library, which is widely used in the CleanIAM project.

2. `https://wolverinefx.net/`

The other database framework is MartenDb[3]. Marten enables PostgreSQL as a document database in .NET, with a focus on allowing developers to work efficiently compared to the typical ORM approach. In the CleanIAM project, MartenDb is used to store all data that is not managed by the OpenIddict library. Since MartenDb enables PostgreSQL as a document database, it brings many advantages, such as flexible data schemas with automatic migrations or natural data modeling for complex hierarchical data structures, and is well suited for Event sourcing and CQRS pattern, which is more described in section 6.1. Deeper usage of MartenDb and conventions used in the CleanIAM project are explained in the main documentation website.

Since the OpenIddict library uses Entity Framework and has no support for events, the implementation decision was made that slices `Identity`, `Scopes`, and `Applications`, described in section 5.3, will access the same database schema. This decision will make it much easier to deploy the application and work with OpenIddict, since having separate database tables for each of the two slices would require some external database synchronization, and that would go against the goals of this project, to make the CleanIAM usage as simple as possible. However, to make this decision work without conflicts, some rules were created.

The OpenIddict creates four database tables: OpenIddictApplications, OpenIddictAuthorizations, OpenIddictScopes, and OpenIddictTokens. Each table is assigned to a specific slice, which acts as its primary owner and is responsible for inserting and updating data. Other slices that are not assigned a particular table have read-only access to it. This structure ensures that, even when multiple slices interact with the same tables, a single source of truth is maintained. The OpenIddictApplications table is assigned to the `Applications` slice, the OpenIddictScope table is assigned to the `Scopes` slice, and the remaining OpenIddictAuthorizations and OpenIddictTokens tables are assigned to the `Identity` slice.

3. `https://martendb.io/`

6.3 Password Storage

Passwords represent one of the most sensitive data within IAM systems. Due to their critical nature, implementing industry best practices for password storage is essential. In the case of attackers successfully breaching databases, improperly stored passwords become vulnerable assets that are easily exploitable by attackers who can leverage them in subsequent attacks. This dangerous scenario places application users at significant risk, and thus, storing passwords securely should be a high priority.

Passwords should never be stored in plaintext. Another option is to encrypt passwords before storing them in the database. However, data encryption is a two-way function, and encrypted data can be decrypted, and thus the original data can be retrieved. For that reason, data encryption is not good for storing passwords. [31]

Password hashing uses a special hashing function to transform a password into a different string called a password hash. The password hashing function must have the following properties so it can be used for appropriate password hashing.

- The hashing function must be deterministic, meaning that for the same input, there will be the same output.
- The hashing function must have a fixed output length independent of the input.
- The hashing function should be a one-way operation, meaning that attackers cannot reconstruct the original password from the password hash.
- The hashing function should be collision-resistant, meaning that two inputs should never have the same output.
- The hashing function should have an avalanche effect, meaning a small change in the input results in a significant change in the output.

Based on these properties, it is hard for the attackers to reconstruct the original password from the password hash.

For an additional layer of security, it is recommended to use the salting principle when hashing passwords. A salt is a random value that is generated and added to the password before it is hashed. This ensures that even if two users have the same password, their hashes will be different due to the unique salt. Salting effectively prevents attackers from using precomputed hash tables called rainbow tables to crack passwords. Salts should be unique for each password and should be stored along the password hash in a database, as they are needed for verification during login attempts. By incorporating salting, password hashes become much more resistant to common attack methods, greatly enhancing overall security. [32]

Another layer of security, in addition to the salting, can be peppering. Same as in the case of salting, peppering adds an additional string to the password, but in this case, the pepper is not stored in the database alongside the password hash. Instead, it is stored in secret vaults or hardware security modules (HSMs)[4]. This prevents an attacker from being able to crack any of the hashes if they only have access to the database.

The CleanIAM project uses password hashing for password storage with an additional security layer using salting. The exact implementation of the hashing function is a hashing function called `argin2id`. This hashing function is the most recommended password hashing function implementation at the time of implementation [32]. The configuration of this function is based on the best security standards, and the exact values are 19 MiB of memory, with an iteration count of 2 and 1 degree of parallelism. Salts are generated using `System.Security.Cryptography` library and are stored alongside the password hash in the database, along with the information about the algorithm used to hash the password, helping with future algorithm migration. The hash comparison on user sign-in is implemented using the 'CryptographicOperations.FixedTimeEquals'. The fixed time comparison helps to mitigate the timing attacks.

4. `https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html`

6.4 Project structure

One of the main goals of this project is to make a clean, maintainable, and well-structured project. To achieve this, the codebase has been organized according to architectural patterns discussed in section 5.2, such as clean architecture and vertical slices. These were the main templates when structuring the repository and the code.

Repository Directory Structure

At the root of the directory structure, there is a main dotnet solution file `CleanIAM.sln`, a main readme file `ReadMe.md` with important project information, and the `docs` directory containing the documentation files. At the main directory at the root of the CleanIAM project is also a `src` directory. The src directory contains all of the source files of each slice. The diagram in Figure 6.1 graphically shows the root project hierarchy.

```
root of the project
├── docs
├── src
├── CleanIAM.sln
└── ReadMe.md
```

Figure 6.1: The CleanIAM root directory structure

The directory structure of each slice is shown in Figure 6.2. The structure is based on the clean architecture with four layers.

The `Api` directory contains everything that the end user can come in contact with, such as endpoints defined in controllers, API models used by controllers, or, in the case of `Identity` slice, the views the user's web browser displays.

The `Application` directory contains the implementation of the business logic implemented with the CQRS pattern, thus the directories `Commands` and `Queries`. The `EventHandlers` directory contains functions reacting to selected application events. The common functionality is sometimes grouped into services to contain the imple-

```
Slice                    -- Root folder for the slice
├── Api                  -- API layer
│   ├── Controllers      -- API controllers
│   │   └── Models       -- API  models
│   └── Views            -- Views and view models
├── Application          -- Application layer
│   ├── Commands         -- Commands and their handlers
│   ├── EventHandlers    -- Handlers for domain events
│   ├── Interfaces       -- Service interfaces
│   └── Queries          -- Queries and their handlers
├── Core                 -- Core domain layer
│   │                       + Domain aggregates
│   └── Events           -- Domain events
└── Infrastructure       -- Infrastructure layer
    ├── AnticorruptionLayer -- Mapping for external events
    └── Services         -- Implementations of interfaces
```

Figure 6.2: The CleanIAM slice directory structure

mentation and make the code cleaner. The interfaces specifying this functionality are defined in `Interface` directory.

In the `Core` directory are located definitions of core domain aggregates, and also the `Events` directory that contains the definitions of the slice-specific events.

The `Infrastructure` directory contains `AnticorruptionLayer` and `Services` directories. The `AnticorruptionLayer` contains definitions of mappers that map local events to global events. This provides another layer of isolation between slices. And in the `Services` directory are implementations of the interfaces defined in the `Interfaces` directory.

Code Structure

It is also important to define the regularly used code structures to ensure consistency, readability, and ease of collaboration across the project. Establishing clear patterns for common structures allows contributors to quickly understand and navigate the codebase.

The controllers used to create the REST endpoints are structured as is typical for traditional MVC applications. The only differences are the usage of primary constructors for dependency injection and the body of a typical endpoint function, where there is typically only a command or query message invocation.

Another important constructs frequently used in the codebase are commands and queries, since the CleanIAM project applies the CQRS pattern. As mentioned in section 6.1, the CleanIAM project uses the WolverineFx framework for messaging. The WolverineFx message handler discovery depends on naming conventions, so the structure of the command and query handlers is influenced by this fact.

```
1  public class ExampleCommandHandler
2  {
3      public static async Task<Result> LoadAsync(ExampleCommand command,
4          IExampleService service /* Injecting service from DI */)
5      {
6          return Result.Ok();
7      }
8
9      public static async Task<Result<ExampleEvent>> HandleAsync(ExampleCommand command,
10         Result loadResult, IExampleService service /* Injecting service from DI */)
11     {
12         if (loadResult.IsError())
13             return loadResult;
14
15         // Perform the command logic here
16
17
18         // Publish the ExampleEvent event and return
19         var exampleEvent = command.Adapt<ExampleEvent>();
20         await bus.PublishAsync(exampleEvent);
21         return Result.Ok(exampleEvent);
22     }
23 }
```

Figure 6.3: Command and command handler example structure

The Figure 6.3 contains an example structure of a command and its handler. This structure would be located in a file with the same name as the command, in this case `ExampleCommand.cs`, in the `Commands` directory. The command name is saying to do something with the suffix `command`. The command is a type of record and can have any properties in it.

The command handler is a class with the name with the suffix `Handler`. It contains two methods, `LoadAsync` and `HandleAsync`. These

methods must be static. The methods are called in order by WolverineFx runtime, and the return value of `LoadAsync` function can be accessed as a second parameter in the `HandleAsync`, as shown in the example.

The first argument of both `LoadAsync` and `HandleAsync` must always be the command. This is how WolverineFx connects commands and their handlers. All of the other arguments are retrieved from the dependency injection container.

At the beginning of the `HandleAsync` function, there must be a check for the return value from the `LoadAsync` function, since WolverineFx doesn't do this by itself. After that, the main business logic follows. And finally, there is the event publishing and returning the Result value, at the end of the `HandleAsync` function.

The query and query handler are the same as the commands and their handler, only with small changes. The query handler doesn't have a `LoadAsync` function, and the body of the `HandleAsync` function contains only a database query.

6.5 Error handling

The CleanIAM project defines its own Result value to handle error situations without the need for exceptions, which makes the execution path cleaner and straightforward.

The Result value is inspired by the Result implementation in the Rust programming language. If the result represents success, it can contain some generic return value, and if the result value represents an error, it contains optional error messages of type string and optional error code, simplifying the integration with HTTP endpoints.

The Result class also implements the `IActionResult`, allowing it to be used as a return value directly in the endpoint. If the success value of the non-generic Return type is returned from the HTTP endpoint, the HTTP status code is `204 NoContent`, and if the Result type is generic, the HTTP status code is `200 Ok`, and the generic value is serialized to JSON format using the `System.Text.Json` nugget package.

If the Result type represents an error, the HTTP status code of the response is the error code defined when creating the Result value

with the fallback value to `500 InternalError`. and the body of the response is displayed in figure 6.4.

```
{
    "Message": "The message describing the error"
    "Code": 400
}
```

Figure 6.4: Body of error response generated by custom Result type.

This custom Result type removes the hidden execution flows that the exceptions bring, making the code cleaner and thus contributing to the important goal of making the CleanIAM project simple and easy to maintain and extend.

6.6 User Interface

As it was already described in section 5.4, the user interface for the CleanIAM project can be divided into two sections. The first section represents the user interface for the `Identity` slice managing OAuth and OpenID flows. This part of the UI is implemented using .NET views served directly from CleanIAm controllers. This adds another layer of security since the views are generated server-side, ensuring that sensitive data such as client secrets or tokens is never exposed on the client-side. By handling the authentication flows within the CleanIAM controllers, this approach minimizes the risk of security vulnerabilities like cross-site scripting (XSS) or token leakage.

The second part of the UI represents represents frontend application called the Management portal [5]. The management portal provides a user interface for managing OIDC applications, scopes, users, and tenants, and also provides a profile page for each user for configuring their profile, such as changing name, or configuring MFA.

The Management portal is implemented using the React library and is written in TypeScript. There is also a custom script prepared for generating ready-to-use functions wrapping TanStack Query and also providing types for all API calls. These functions and types are

5. `https://github.com/CleanIAM/CleanIAM-ManagementPortal`

generated from the OpenAPI schema generated by the main CleanIAM application. This generation allows easy integration with any API, and because of the generated types, makes the application more secure and easier to use.

6.7 Deployment

To be able to use this project properly, it needs to be deployed to a server. Sometimes the deployment can be hard and time-consuming. For this reason, the CleanIAM project tries to make the deployment as simple as possible.

The primary deployment process is based on containerization, which allows the CleanIAM project to run consistently across various environments. There is a prepared Dockerfile in the root of the CleanIAM project, packing the whole project into a single container image. These container images can be run simply on any server or even a Kubernetes cluster. The application even provides static check endpoints such as `\healthz` or `\readyz`. The Docker container can be built for both the core ClientIAM applications and the Management Portal user interface.

Another option for deployment, which has been tested, is the Microsoft Azure App Service. This is also a simple and quick way to deploy the CleanIAM project. This deployment option removes many deployment concerns from the developer, such as routing, SSL certificate, or DNS management, since all of these are automatically managed by Microsoft Azure.

In summary, the CleanIAM project supports multiple deployment options and is designed for simplicity and flexibility to accommodate a wide range of production environments.

6.8 Testing

Testing is a critical part of the development life cycle, ensuring that the produced code is functioning correctly and that the required features are met. The CleanIAM project is tested through a combination of automated testing and manual testing using demo applications.

The OAuth flow correctness is manually verified by multiple demo applications that use all of the OAuth endpoints and simulate different authorization scenarios. These demo applications cover common use cases such as authorization code flow, token refresh, and error handling. Each demo is designed to ensure that endpoints like `/authorize`, `/token`, and `/userinfo` behave as expected under valid and invalid conditions.

Automated tests are conducted using the K6 testing tool[6], which is used to verify the functionality of management endpoints, including operations like creating, updating, or deleting OIDC applications, as well as user management. The version of K6 used in this project runs testing scripts written in JavaScript. These tests serve as integration tests, as they validate the correct behavior of the API endpoints within the CleanIAM project.

6.9 Documentation

An essential part of open source projects is well-written documentation. Documentation helps new developers understand the projects and orient themselves in the codebase. It also assists users in learning how to install, configure, and effectively use the project.

The CleanIAM project includes documentation on multiple levels, comments in the code, a ReadMe file in the root of the project, and a main documentation website describing each part of the project. The OpenAPI schema[7] is also generated to document the project endpoints formally. This should help developers understand every part of the project and quickly accommodate their needs.

The project code base is extensively commented with C# documentation comments [8]. This helps developers to easily use prepared classes and their methods since most of the IDEs can parse these documentation comments and show the developer usage and constraints of constructs being used.

---

6. `https://k6.io/`
7. `https://spec.openapis.org/oas/latest.html`
8. `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments`

The documentation comments in the codebase also help generate a better OpenAPI schema since the comments propagate to the OpenAPI schema and describe endpoints and models with human language, helping the users understand the project from a connectivity point of view. The OpenAPI scheme is automatically generated using the `Swashbuckle.AspNetCore` package.

The ReadMe file in the root of the project is the first part of the documentation that the developers will see, since it is rendered on the main page of the project's GitHub page. For this reason, the ReadMe file quickly introduces the project and provides basic information about it. On top of that, the ReadMe file also includes a link to the main documentation website, instructions on how to use and run the project, and even instructions on how to deploy it to production.

The main documentation website describes each part of the project in a clean format. This documentation page uses the VitePress[9] to make the page well-designed and easily readable documentation. The source documentation text for VitePress is written in multiple files in markdown format, which makes it easy to write the documentation quickly without compromising on quality. The documentation markdown files are located in `docs` directory in the root of the project.

The generated documentation website is deployed using GitHub Pages to make it part of the main GitHub repository. The documentation page is updated on each push to the master branch using GitHub Actions, ensuring the documentation page is always up to date.

For even easier usage, the CleanIAM project GitHub profile contains multiple repositories with example client applications showing the connection to the CleanIAM project in multiple programming languages.

9. `https://vitepress.dev/`

7 Future Plans

In the project development, there is a continuous need for updates and fixes to ensure that the system remains reliable, functional, secure, and adaptable to changing requirements. As the CleanIAM project moves forward, several key areas will require attention and improvement.

Enhanced Security Features Security is a critical area for the CleanIAM project. New security standards must be continuously monitored and integrated into the system. This includes staying up to date with industry best practices, implementing new protocols and encryption methods as needed, and ensuring the system remains secure against emerging threats.

SAML Integration Another important step is the integration of Security Assertion Markup Language (SAML) for Single Sign-On functionality. SAML is widely used in enterprise environments, and implementing this protocol will increase the flexibility and reach of the CleanIAM solution, allowing it to be easily adopted by organizations using SAML for authentication.

Microservices Transition Over time, the modular monolith architecture may evolve into microservices, depending on the project's growth and the need for scalability. This would involve extracting certain slices or functionalities into independent services, allowing for easier scaling and better isolation of concerns.

API Enhancements To ensure that CleanIAM remains adaptable to a variety of applications, future work should extend the existing API to expose additional endpoints and functionality. This will enable better integration with third-party systems and allow for more customizable use cases.

Documentation As the project evolves, it will be crucial to maintain up-to-date documentation. This will include API documentation, guides for integrating CleanIAM into existing applications, and troubleshooting support for developers.

Community Involvement Engaging with the user community and gathering feedback is an important part of the iterative development process. Regularly collecting input from developers and end-users will guide future improvements and help prioritize features that align with user needs.

Each of these areas will contribute to the long-term success of the CleanIAM project, ensuring that it remains a flexible, secure, and scalable solution for managing authentication and identity across applications.

8 Conclusion

This thesis aimed to design and implement a custom Identity and Access Management system specifically adapted to the needs of modern environments. It explored the state-of-the-art IAM solutions, analyzed their strengths and limitations, and proposed a new solution to address the gap in the .NET ecosystem left by the deprecation of IdentityServer4.

Through extensive research, a review of key IAM frameworks like Keycloak, Zitadel, and Ory, and an examination of their integration possibilities and shortcomings, this thesis established the key requirements for a robust IAM solution. The most notable contribution of this work is the development of the CleanIAM project, an open-source IAM solution built using the latest technologies, designed for flexibility, scalability, and ease of customization.

The proposed solution offers several improvements over existing IAM solutions. Focusing on OAuth 2.0 and OpenID Connect compliance ensures compatibility with industry standards. It supports core authentication and authorization flows such as the Authorization Code Flow with PKCE, multi-factor authentication, and client management. In addition, the CleanIAM project offers advanced user management, role-based authorization, and integration with external identity providers, which improves usability and security.

From an architectural standpoint, the CleanIAM project leverages Vertical Slice Architecture, which organizes the application by its features rather than technical layers, making the solution more maintainable and scalable. Using event-driven and Clean Architecture patterns further strengthens its modularity and extensibility. This approach ensures that the system can evolve without causing disruptions to other components, an essential trait for future-proofing the solution in dynamic environments.

Having met all requirements outlined in Chapter 5, and after thorough validation through automated K6 tests as well as manual testing, the solution demonstrates reliability and robustness.

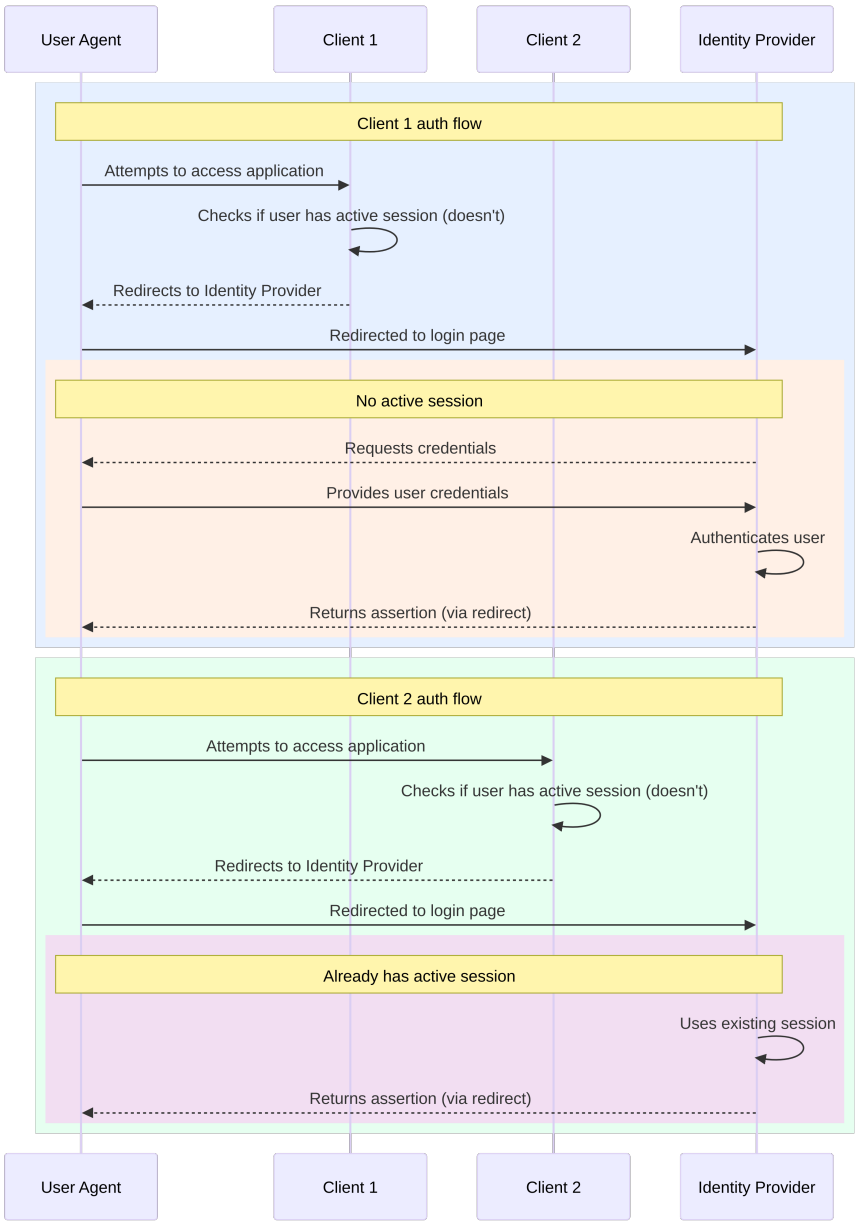
This work brings a new approach to IAM in the .NET ecosystem, particularly in addressing the gap left by IdentityServer4. By leveraging OpenIddict for OAuth and OpenID Connect functionalities, the

CleanIAM project offers an alternative for developers in the .NET world who need a customizable IAM system with built-in security protocols and user management support.

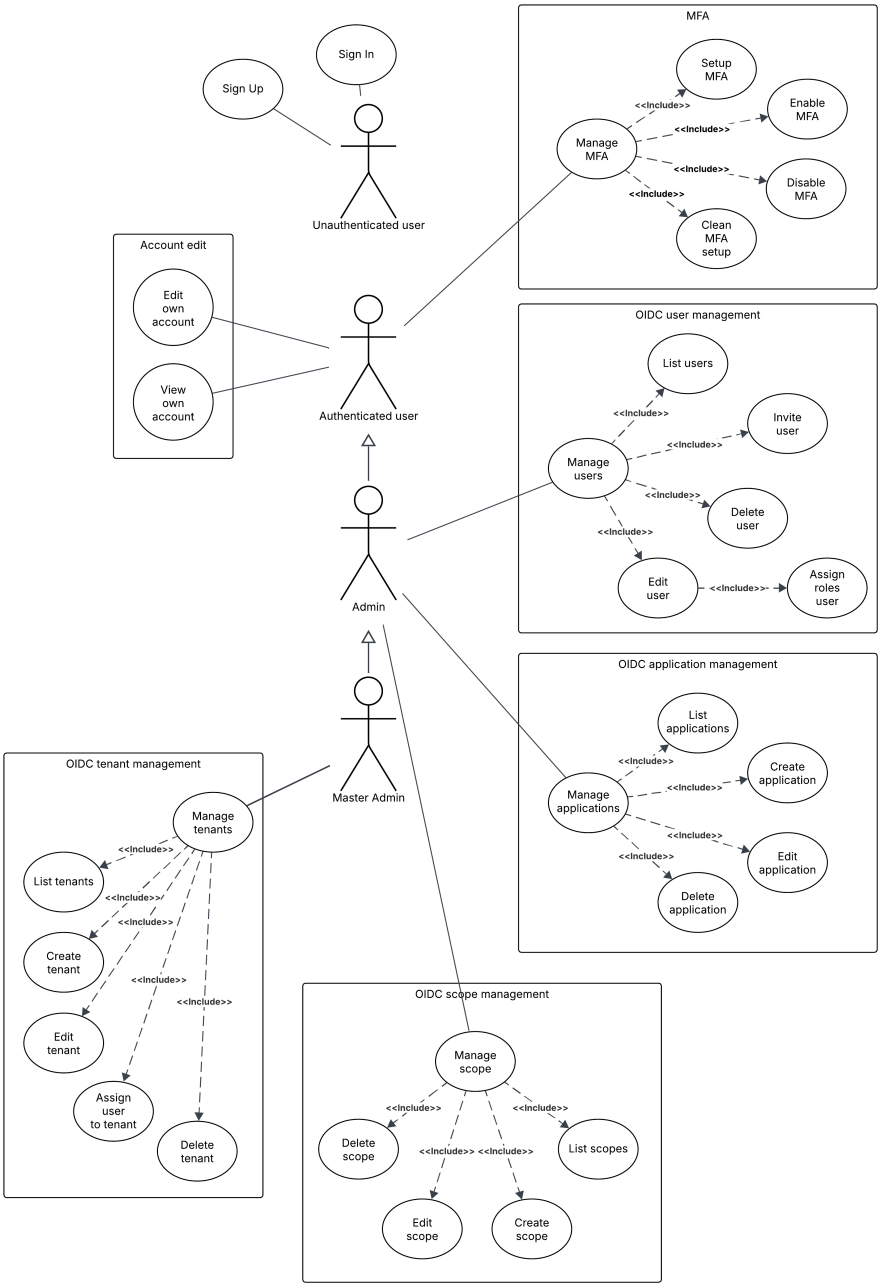
The CleanIAM solution brings a strong base for developing secure and adaptable identity management systems within the .NET ecosystem. Its modular design allows organizations to tailor the system to their requirements without extensive rewriting or compromising security standards. However, the base CleanIAM solution can be further enhanced and extended to address more specialized use cases and emerging security challenges. Future improvements may include the integration of the SAML protocol or extending the API. The SAML integration would ensure compatibility with legacy systems and enterprise environments that depend on this widely adopted standard for federated identity management. And the extension of the API would provide more granular control and customization options for client applications.

In summary, this thesis provides a comprehensive solution for IAM needs, particularly addressing gaps in the .NET world, and contributes to the field with an open-source, highly customizable IAM solution. The CleanIAM project is a solid foundation for further development and could be expanded to meet the growing demands for secure and flexible identity management solutions. The ability to integrate new features, whether in response to emerging technologies or changing security standards, positions this work as a valuable asset for developers and organizations aiming to improve their IAM systems.

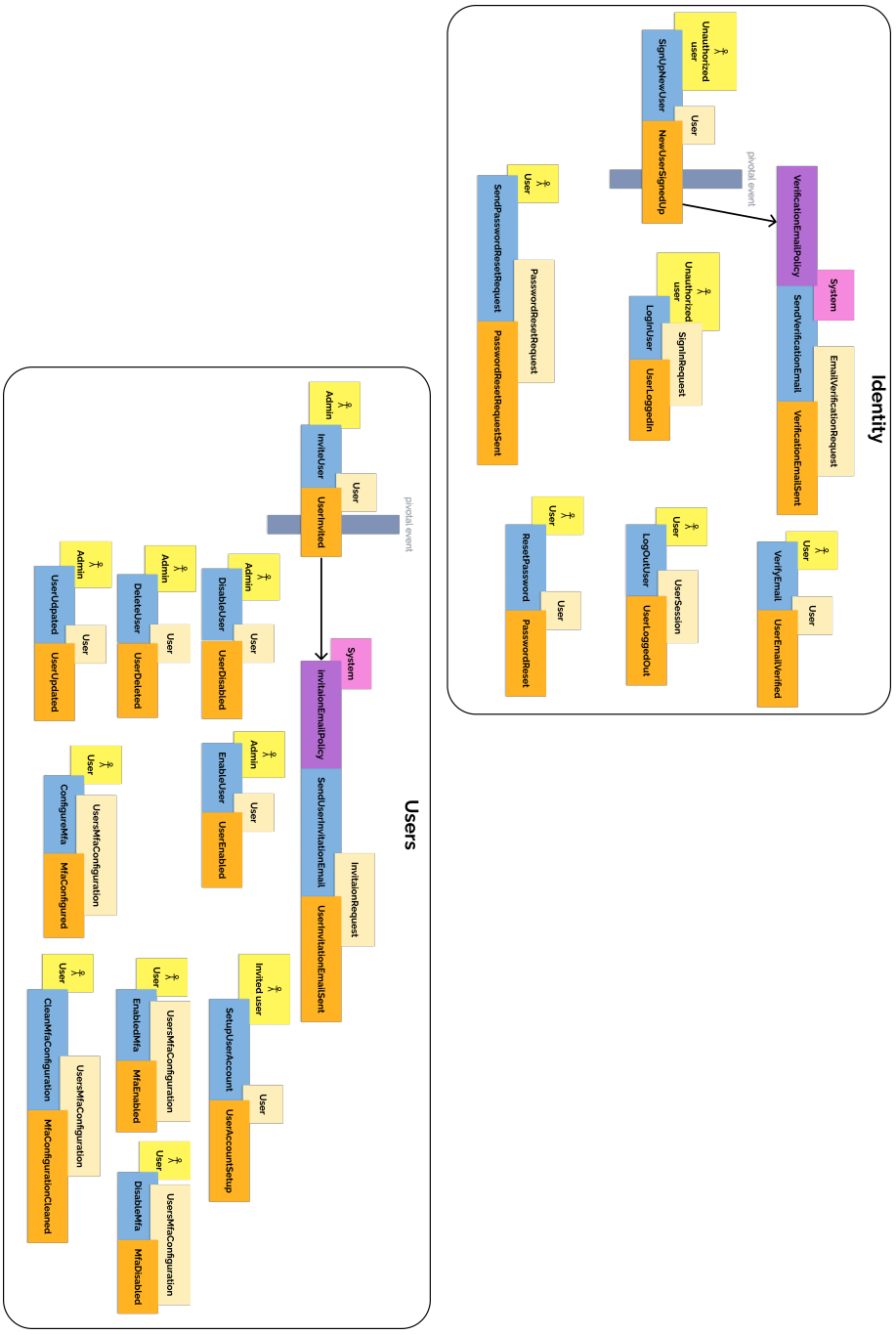
A SSO auth flow example diagram

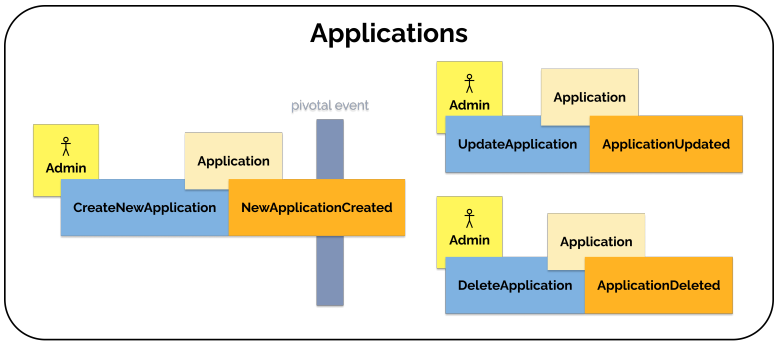


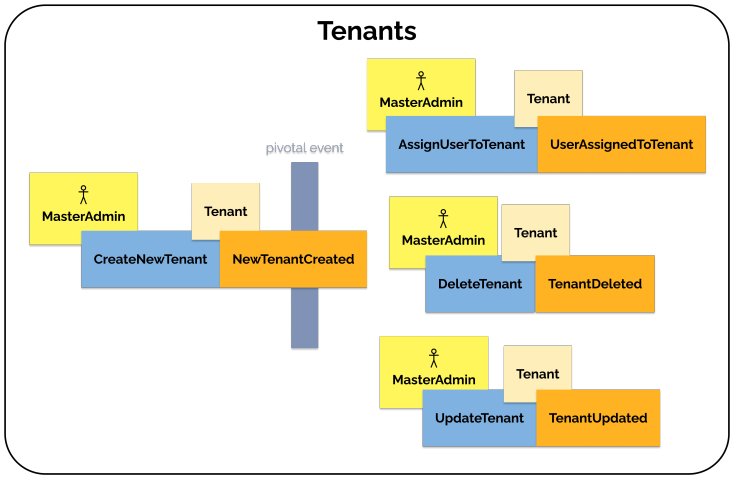
B Use case diagram for CleanIAM project

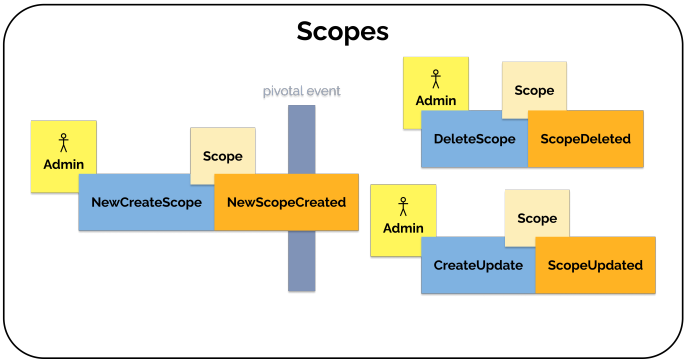


C Event Storming for CleanIAM project




Applications
pivotal event
Admin
Application
CreateNewApplication
NewApplicationCreated
Admin
Application
UpdateApplication
ApplicationUpdated
Admin
Application
DeleteApplication
ApplicationDeleted


Tenants
pivotal event
MasterAdmin
Tenant
CreateNewTenant
NewTenantCreated
MasterAdmin
Tenant
AssignUserToTenant
UserAssignedToTenant
MasterAdmin
Tenant
DeleteTenant
TenantDeleted
MasterAdmin
Tenant
UpdateTenant
TenantUpdated


Scopes
pivotal event
Admin
Scope
NewCreateScope
NewScopeCreated
Admin
Scope
DeleteScope
ScopeDeleted
Admin
Scope
CreateUpdate
ScopeUpdated

Bibliography

1. SINGH, Chetanpal; THAKKAR, Rahul; WARRAICH, Jatinder. IAM Identity Access Management—Importance in Maintaining Security Systems within Organizations. *European Journal of Engineering and Technology Research*. 2023, vol. 8, no. 4, pp. 30–38. Available from DOI: 10.24018/ejeng.2023.8.4.3074.
2. CLOUD NATIVE COMPUTING FOUNDATION. *Keycloak GitHub Repository* [https://github.com/keycloak/keycloak]. GitHub, 2013 [visited on 2025-03-22].
3. CHATTERJEE, Ayan; PRINZ, Andreas. *Applying Spring Security Framework with KeyCloak-Based OAuth2 to Protect Microservice Architecture APIs: A Case Study*. Vol. 22. Switzerland: MDPI AG, 2022. ISSN 1424-8220.
4. CLOUD NATIVE COMPUTING FOUNDATION. *Keycloak Documenation* [online]. 2025. [visited on 2025-03-22]. Available from: https://www.keycloak.org/documentation.
5. CAOS AG. *Zitadel GitHub Repository* [https://github.com/zitadel/zitadel]. GitHub, 2020 [visited on 2025-03-23].
6. SANDHU, Ravi S; SAMARATI, Pierangela. Access control: principle and practice. *IEEE communications magazine*. 1994, vol. 32, no. 9.
7. CAOS AG. *Zitadel Documenation* [online]. 2025. [visited on 2025-03-23]. Available from: https://zitadel.com/docs.
8. ORY CORP. *Ory GitHub* [https://github.com/ory]. GitHub, [n.d.] [visited on 2025-03-25].
9. OSMANOGLU, Ertem. *Front cover image for Identity and Access Management : Business Performance Through Connected Intelligence Identity and Access Management : Business Performance Through Connected Intelligence*. Syngress, 2014. Lecture Notes in Computer Science. ISBN 9780124104334.
10. ORY CORP. *Ory Documenation* [online]. 2025. [visited on 2025-03-23]. Available from: https://www.ory.sh/docs.

11. DUENDE SOFTWARE. *IdentityServer4* [`https://github.com/DuendeArchive/IdentityServer4`]. GitHub, 2025 [visited on 2025-03-26].

12. DUENDE SOFTWARE. *IdentityServer documentation* [online]. 2021. [visited on 2025-03-26]. Available from: `https://docs.duendesoftware.com/identityserver/v7`.

13. WHITESELL, Sean; RICHARDSON, Rob; GROVES, Matthew D. ASP.NET Core Overview. In: *Pro Microservices in .NET 6: With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes*. Berkeley, CA: Apress, 2022. ISBN 978-1-4842-7833-8. Available from DOI: `10.1007/978-1-4842-7833-8_2`.

14. OPENIDDICT. *openiddict-core* [`https://github.com/openiddict/openiddict-core`]. GitHub, 2025 [visited on 2025-03-28].

15. LI, Kepeng; HUNT, Phil; KHASNABISH, Bhumip; NADALIN, Anthony; ZELTSAN, Zachary. *System for Cross-domain Identity Management: Definitions, Overview, Concepts, and Requirements* [RFC 7642]. RFC Editor, 2015. Request for Comments, no. 7642. Available from DOI: `10.17487/RFC7642`.

16. DE CLERCQ, Jan. Single Sign-On Architectures. In: DAVIDA, George; FRANKEL, Yair; REES, Owen (eds.). *Infrastructure Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 40–58. ISBN 978-3-540-45831-9.

17. WANG, Ding; ZHANG, Xizhe; ZHANG, Zijian; WANG, Ping. Understanding security failures of multi-factor authentication schemes for multi-server environments. *Computers security*. 2020, vol. 88, p. 101619. ISSN 0167-4048.

18. CVRČEK, Tadeáš; DZURENDA, Petr. Access Control System Using Multi-factor Authentication. In: *Access Control System Using Multi-factor Authentication*. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2022, pp. 190–193. ISBN 978-80-214-6030-0.

19. NAGHMOUCHI, Montassar; AYED, Hella Kaffel Ben; LAURENT, Maryline. An Automatized Identity and Access Management System for IoT Combining Self-Sovereign Identity and Smart Contracts. In: *FPS 2021: The 14th International Symposium on Foundations and Practice of Security*. Switzerland: Springer International Publishing AG, 2022, vol. 13291, pp. 208–217. ISBN 9783031081460. ISSN 0302-9743.

20. SINGH, Chetanpal; THAKKAR, Rahul; WARRAICH, Jatinder. IAM Identity Access Management—Importance in Maintaining Security Systems within Organizations. *European Journal of Engineering and Technology Research*. 2023, vol. 8, no. 4, pp. 30–38. Available from DOI: 10.24018/ejeng.2023.8.4.3074.

21. JØSANG, Audun. IAM—Identity and Access Management. In: *Cybersecurity: Technology and Governance*. Cham: Springer Nature Switzerland, 2025, pp. 191–214. ISBN 978-3-031-68483-8. Available from DOI: 10.1007/978-3-031-68483-8_9.

22. LEIBA, B. OAuth Web Authorization Protocol. *IEEE internet computing*. 2012, vol. 16, no. 1. ISSN 1089-7801.

23. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [RFC 6749]. RFC Editor, 2012. Request for Comments, no. 6749. Available from DOI: 10.17487/RFC6749.

24. BAKER, Matthew. OAuth2. In: *Secure Web Application Development : A Hands-On Guide with Python and Django*. Berkeley, CA: Apress, 2022, pp. 351–397. ISBN 978-1-4842-8596-1. Available from DOI: 10.1007/978-1-4842-8596-1_11.

25. NEWMAN, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. First edition. Beijing: O'Reilly, 2019. ISBN 9781492047841. Available also from: http://lib.ugent.be/catalog/rug01:002822828.

26. RATNER, Ian Michael; HARVEY, Jack. Vertical Slicing: Smaller is Better. In: *2011 Agile Conference*. 2011, pp. 240–245. Available from DOI: 10.1109/AGILE.2011.46.

27. MARTIN, Robert C. *Clean architecture : a craftsman's guide to software structure and design*. London, England: Prentice Hall, 2018. Robert C. Martin series. ISBN 978-0-13-449416-6.

28. CLARK, Tony; BARN, Balbir S. Event driven architecture modelling and simulation. In: *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. 2011, pp. 43–54. Available from DOI: 10.1109/SOSE.2011.6139091.

29. BRANDOLINI, Alberto. Introducing event storming. *blog, Ziobrando's Lair*. 2013, vol. 18.

30. OBE, Regina O; HSU, Leo S. *PostgreSQL: up and running: a practical guide to the advanced open source database*. " O'Reilly Media, Inc.", 2017. ISBN 9781491963364.

31. LOPEZ, Javier; WU, Yongdong. *Information Security Practice and Experience: 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015, Proceedings*. Vol. 9065. 2015th ed. Cham: Springer Nature, 2015. Lecture Notes in Computer Science. ISBN 3319175335.

32. OWASP. *Password Storage Cheat Sheet* [online]. 2025. [visited on 2025-04-03]. OWASP Cheat Sheet Series. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html.