

Authentication and Authorization Service for Developers: Implementation of Authentication-as-a-Service on the .NET platform

Bc. Martin Gabriel



ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Martin Gabriel
Osobní číslo:	A23596
Studijní program:	N0613A140022 Informační technologie
Specializace:	Softwarové inženýrství
Forma studia:	Kombinovaná
Téma práce:	Autentizační a autorizační služba pro developery: Realizace authentication-as-a-service na platformě .NET
Téma práce anglicky:	Authentication and Authorization Service for Developers: Implementation of Authentication-as-a-Service on the .NET platform

Zásady pro vypracování

- Provedte rešerši existujících autentizačních a autorizačních služeb typu authentication-as-a-service.
- Vypracujte stručný rozbor technologií pro návrh a implementaci řešení.
- Provedte analýzu požadavků na navrhovanou službu.
- Navrhněte a implementujte webovou aplikaci a webové API na základě výsledků analýzy.
- Věnujte pozornosti zabezpečení aplikace.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. FREEMAN, Adam. *Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*. 8th ed. 2020. Berkeley, CA: APress, 2020 ;. ISBN 978-1-4842-5440-0.
2. WILSON, Yvonne. *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0*. Online. London, UK: Abhishek Hingnikar, 2019. ISBN 978-1-4842-5095-2. Dostupné z: <https://doi.org/10.1007/978-1-4842-5095-2>. [cit. 2024-10-19].
3. DEJONGHE, Derek a NUGARA, Arlan. *Application delivery and load balancing in Microsoft Azure: practical solutions with NGINX and Microsoft Azure*. Beijing: O'Reilly, 2021. ISBN 978-1-098-11586-9.
4. FREEMAN, Adam. *Pro ASP.NET Core identity: under the hood with authentication and authorization in ASP.NET Core 5 and 6 applications*. New York, NY: Apress, 2021 ;. ISBN 978-1-4842-6857-5. Dostupné z: <https://doi.org/978-1-4842-6857-5>.
5. HOFFMAN, Kevin. *Building Microservices with ASP.NET Core: Develop, Test, and Deploy Cross-Platform Services in the Cloud*. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1-491-96173-5.
6. PRICE, Mark J. *C# 10 and .NET 6 – modern cross-platform development: build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code*. Sixth edition. Birmingham: Packt, 2021. ISBN 978-1-80107-736-1.
7. ANDERSON, Ross. *Security engineering: a guide to building dependable distributed systems*. Third edition. Indianapolis: Wiley, [2020]. ISBN 978-1-119-64278-7.

Vedoucí diplomové práce: **doc. Ing. Petr Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce popisuje řešení, design a implementaci služby typu Authentication-as-a-Service postavenou na platformě .NET. Popisuje autentizační a autorizační standardy jako je OAuth a OpenID Connect a porovnává přední poskytovatele obdobných služeb s cílem stanovit architektonické požadavky pro navrhovanou službu. Na základě této analýzy navrhuje systém, který kombinuje OAuth flow, webové JSON tokeny s ohledem na uživatelské role a ASP.NET Core back-end s uživatelským rozhraním pomocí Razor Pages. Výsledná služba s otevřeným zdrojovým kódem nabízí přihlášení klientských aplikací, místní a federované přihlašování, jemné řízení přístupu a ochranu proti běžným kybernetickým útokům.

Klíčová slova: autentizace, autorizace, OAuth, OpenID Connect, JSON Web Token, .NET, mikroslužby

ABSTRACT

This thesis presents the research, design and implementation of an Authentication-as-a-Service solution built on the .NET platform. It reviews authentication and authorization standards such as OAuth and OpenID Connect and evaluates leading providers to derive architectural requirements for designed service. Guided by this analysis, it constructs a containerized system that combines OAuth flows, role-based JSON Web Tokens and an ASP.NET Core back-end with a Razor Pages user interface. The resulting open-source service offers client application onboarding, local and federated sign-in, fine-grained access control and protection against common cyber attacks.

Keywords: authentication, authorization, OAuth, OpenID Connect, JSON Web Token, .NET, microservices

I would like to thank my supervisor doc. Ing. Petr Šilhavý Ph.D., for his guidance, valuable advices and consulting during the writing of this thesis. I would also like to thank my fiancée Veronika and my entire family for their support. Finally, I would like to wish health and happiness to the seven White Swiss Shepherd puppies born during the completion of this thesis.

I declare that the submitted printed version of the thesis and the electronic version uploaded to IS/STAG are identical.

Complying with rules for using modern technology in creative work, assistive software with elements of artificial intelligence was occasionally used during the preparation of this thesis. Specifically, the ChatGPT generative model, available at <https://chatgpt.com/>, Write-full model as a part of Overleaf editor and GitHub Copilot integrated into the Microsoft Visual Studio Community 2022 development environment were utilized to support text revision, code suggestions and structural refinement.

All generated content was carefully reviewed, edited and validated by the author, who takes full responsibility for the final form, originality and accuracy of this thesis.

TABLE OF CONTENTS

INTRODUCTION	13
I THEORETICAL PART	14
1 AUTHENTICATION AND AUTHORIZATION SERVICES	15
1.1 DIFFERENCE BETWEEN AUTHENTICATION AND AUTHORIZATION.....	15
1.1.1 Authentication	15
1.1.2 Authorization	16
1.2 TECHNOLOGIES BEHIND AUTHENTICATION AND AUTHORIZATION SER- VICES	16
1.2.1 OAuth 2.0	17
1.2.2 OAuth 2.1	17
1.2.3 OpenID Connect	17
1.2.4 JSON Web Token	18
1.2.5 SAML 2.0.....	19
1.3 EXISTING SERVICES ON THE MARKET.....	19
1.3.1 Auth0	20
1.3.2 Azure Active Directory B2C	20
1.3.3 Firebase Authentication.....	20
1.3.4 Okta	21
1.3.5 Keycloak	21
1.3.6 FusionAuth	21
1.4 COMPARATIVE ANALYSIS AND CONSIDERATIONS.....	22
2 TECHNOLOGY ANALYSIS FOR SOLUTION DESIGN	24
2.1 FRAMEWORK.....	24
2.2 AUTHORIZATION PROTOCOL	25
2.2.1 Authorization flows	25
2.3 AUTHENTICATION PROTOCOL.....	28
2.4 DATABASE	29
2.5 OBJECT-RELATIONAL MAPPING	29

2.6	FRONTEND	29
2.7	DEPLOYMENT	30
2.8	LOAD-BALANCING AND API GATEWAY	30
II	PRACTICAL PART.....	31
3	SOLUTION GOALS AND USER STORIES	32
3.1	USER STORIES	32
4	REQUIREMENTS ANALYSIS	36
4.1	FUNCTIONAL REQUIREMENTS	36
4.2	MAPPING OF USER STORIES TO THESIS CHAPTERS	40
4.3	NON-FUNCTIONAL REQUIREMENTS	41
4.4	QUALITY ATTRIBUTES AND DESIGN DECISIONS	43
4.5	IMPLICATIONS FOR ARCHITECTURE	44
5	SOLUTION DESIGN AND IMPLEMENTATION	45
5.1	AUTHENTICATION AND AUTHORIZATION FLOW	46
5.2	AUTHENTICATION FLOW	47
5.2.1	Authorization logic	48
5.3	SERVICE ENDPOINTS	50
5.3.1	Authorization endpoint /connect/authorize	51
5.3.2	Consent handlers /connect/authorize (Accept / Deny)	51
5.3.3	Token endpoint /connect/token	51
5.3.4	Logout endpoint /connect/logout	51
5.3.5	Client-server communication	52
5.3.6	Token types and lifecycles	53
5.3.7	Token creation implementation	54
5.3.8	Refresh Token Support.....	56
5.3.9	OIDC external integration.....	57
5.3.10	Service frontend	58
5.3.11	Service deployment	61
6	SECURITY AND RESILIENCE.....	63

6.1	THREAT MODELING	63
6.2	ENDPOINT SECURITY AND RESILIENCE	64
6.2.1	Declarative protection	64
6.2.2	Anti-forgery and input validation	65
6.2.3	Rate limiting and abuse prevention	65
6.2.4	Explicit error handling	66
6.2.5	Secure credential generation	66
6.3	CLIENT VALIDATION	67
6.3.1	Uniqueness and existence checks	67
6.3.2	Redirect URI validation	67
6.3.3	Integration with OpenIdDict	68
6.4	CLIENT REGISTRATION	68
6.4.1	Registration workflow	68
6.4.2	Server-side validation and persistence	69
6.4.3	Client creation and configuration	69
6.4.4	User feedback and completion	70
6.5	PKCE VALIDATION	70
6.5.1	Enforcing PKCE at client registration	71
6.5.2	PKCE validation at runtime	71
6.5.3	Password management	71
6.6	RATE LIMITER	73
6.6.1	Implementation and configuration	73
6.6.2	Security and resilience	74
6.6.3	Token expiration	75
6.7	TOKEN REVOCATION	75
	CONCLUSION	77
	REFERENCES	79
	LIST OF FIGURES	84
	LIST OF TABLES	85
	LIST OF LISTINGS	86

LIST OF ABBREVIATIONS	87
LIST OF APPENDICES	89

INTRODUCTION

Modern web and mobile applications increasingly rely on distributed microservices and third-party integrations, creating a pressing need for a centralized, secure, and standards-based approach to authentication and authorization. There are many solutions on the market that address issue of authentication and authorization, each with its own strength and weaknesses.

This thesis presents the research, design and implementation of authentication and authorization solution built on .NET platform. The solution uses OAuth 2.0 protocol and OpenID Connect (OIDC) standards, which provides Single Sign-On (SSO) across multiple clients, supports local and federated login, and enforces fine-grained access control via role and scope based JSON Web Token (JWT) access tokens. By combining short-lived access tokens with refresh tokens, employing Proof Key for Code Exchange (PKCE), Argon2id-hashed secrets, and rigorous input validation and rate limiting, the solution demonstrates how to achieve a balance between strong security guarantees, simple integration and smooth user experience.

Chapter 1 presents a survey of existing commercial authentication-and-authorization solutions, evaluating their feature sets, user technologies and standards. Chapter 2 then analyzes and selects the core technologies, including protocols, frameworks and database, that form the foundation of custom service design. Chapter 3 describes the service goals and user stories. Building on that, Chapter 4 derives functional and non-functional requirements, security constraints and performance targets that drive the architecture. Chapter 5 describes in detail the resulting service design and its implementation, covering component diagrams, data flows and key code concepts. Finally, Chapter 6 examines the solution's security and resilience, demonstrating how input validation, rate limiting, token protection and other controls are designed against common cyber-attacks.

The mentioned chapters will detail solution market research, industry used technologies, requirement analysis, architecture, component workflows, and security considerations that underpin this robust and maintainable service.

I THEORETICAL PART

1 AUTHENTICATION AND AUTHORIZATION SERVICES

In modern web applications it is important to have a solutions that ensure secure a reliable client authentication and authorization of user rights. Authentication and authorization services address critical challenges by enabling SSO across platforms, supporting standard protocols such as OAuth 2.0, OIDC, and Security Assertion Markup Language known as SAML 2.0, and offering centralized management of user identities. Many of these solutions also provide seamless integration with enterprise systems such as Lightweight Directory Access Protocol (LDAP) and Active Directory (AD), while their clustering capabilities ensure scalability and high availability [1].

1.1 Difference between Authentication and Authorization

The terms authentication and authrorization refer to two distinct but closely related concepts in the context of web application and web services security. Both are critical components in securing modern distributed systems and ensuring that users or client access only the resources they are permitted to. While authentication is concerned with identity verification, authorization covers what an identified user is allowed to do with the system and resources. According to the publication *Solving Identity Management in Modern Applications* [1], authentication and authorization are described as separate but sequential steps within the lifecycle of identity management. Authentication establishes user identity, whereas authorization enforces access policies associated with that identity [1].

1.1.1 Authentication

Authentication is the process of verifying the identity of a user, essentially answering the question: *Who are you?* This process typically involves the use of credentials such as a username and password. More advanced methods include multi-factor authentication, biometric verification or federated identity mechanisms such as OAuth 2.0 or OIDC [2].

The authentication process can be illustrated as following diagram 1.1

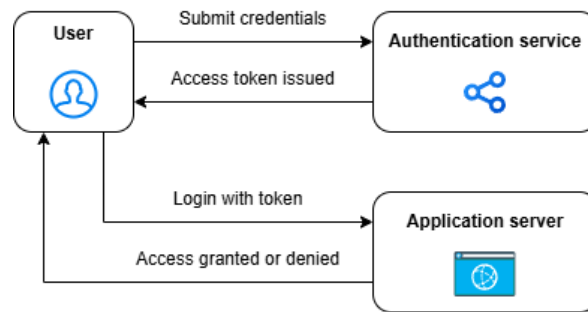


Figure 1.1 Authentication (diagram: own, source: [3])

Once the identity of the user is confirmed, an access token may be generated and returned to the client application, representing the authenticated session.

1.1.2 Authorization

Authorization is another important part to successful authentication. It is the process of determining whether an authenticated user or client has permission to perform specific actions or access certain resources [4]. Authorization mechanisms often rely on Role-Based Access Control, Attribute-Based Access Control or similar models [5].

The authorization process can be illustrated as following diagram 1.2

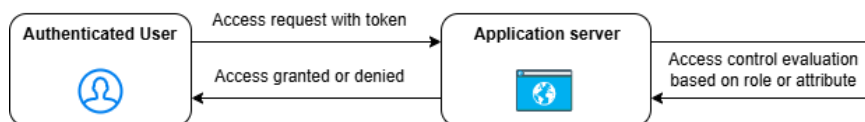


Figure 1.2 Authorization (diagram: own, source: [3])

1.2 Technologies behind authentication and authorization services

Modern authentication and authorization services rely on a couple of industry standard protocols and token formats to ensure secure and scalable identity management. Many of these services implement OAuth 2.0, a robust framework for delegated authorization that enables applications to securely obtain access tokens without exposing user credentials. Building upon this, OpenID Connect adds an authentication layer that provides verified user identities through ID tokens and dedicated user information endpoints. Additionally, JSON Web Tokens offer a compact and self-contained method for representing claims, supporting state-

less authentication across distributed systems. In enterprise contexts, SAML 2.0 is often employed for cross-domain single sign-on, leveraging XML-based assertions to securely exchange authentication and authorization data [1].

1.2.1 OAuth 2.0

OAuth 2.0 is a robust security framework designed for authorizing third-party applications to access protected resources without requiring users to share their credentials. It enables an application to obtain explicit consent from a user, allowing secure access to resources hosted on a different server. This is achieved through an authorization server, which issues access tokens that grant the application the necessary permissions to perform API requests. OAuth 2.0 defines several grant types, such as the authorization code, implicit grant, and client credentials, that cater to various use cases and security requirements. The framework's design not only enhances security by limiting the exposure of sensitive information but also provides granular control over data access, thereby addressing effectively needs of a modern applications [6, 7].

1.2.2 OAuth 2.1

OAuth 2.1 builds on OAuth 2.0 by formalizing security best practices and removing outdated flows. It deprecates the implicit and resource owner password credentials grants, requires PKCE for all authorization code exchanges, enforces the use of the state parameter to prevent Cross-site Request Forgery (CSRF), and standardizes refresh token rotation and sender-constrained tokens. By eliminating legacy options and codifying previously optional recommendations, OAuth 2.1 simplifies the protocol and raises its baseline security posture [8].

1.2.3 OpenID Connect

OpenID Connect is an authentication protocol built on top of OAuth 2.0 that introduces an identity layer to the authorization process. It enables applications to delegate user authentication to an OpenID Provider, which then issues an ID Token containing verified information

about the user. By utilizing endpoints such as the UserInfo endpoint, OIDC allows secure retrieval of user details, including name, email, and profile image, and supports multiple authentication flows like Authorization Code, Implicit, and Hybrid Flow. This flexibility, coupled with support for both standard and custom claims, makes OIDC an effective solution for implementing standard local and federated sing-in and enhancing overall identity management in modern applications [6, 9].

1.2.4 JSON Web Token

JSON Web Token is a standardized token format used for securely exchanging information between parties. The token is composed of three parts [10]:

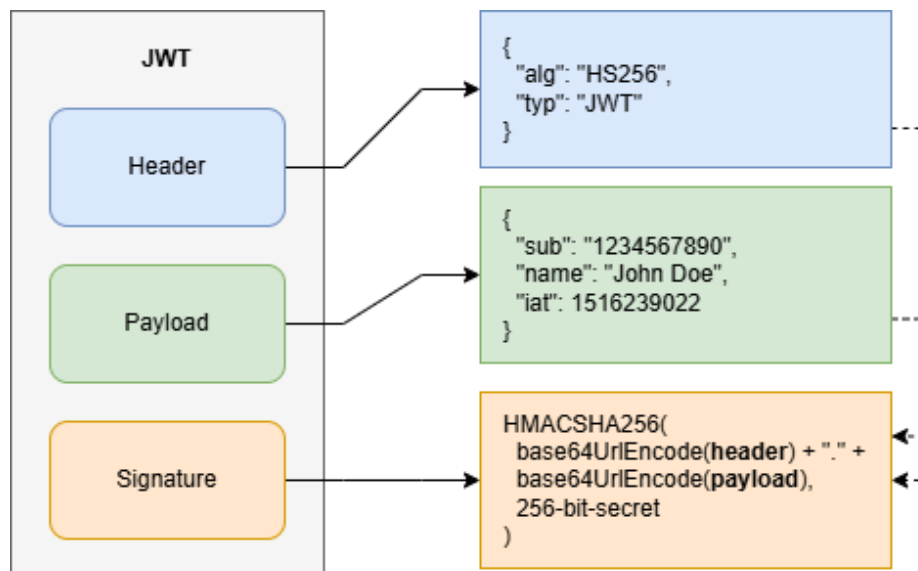


Figure 1.3 JWT structure (diagram: own, source: [10])

- **Header** that specifies the token type and encryption algorithm.
- **Payload** containing claims about the user.
- **Signature** that verifies the token's integrity.

JWT is widely used in authentication systems and the transmission of access permissions, often in conjunction with OAuth 2.0 and OpenID Connect. A key advantage of JWT is that these tokens are self-contained and can be validated independently without needing to

contact a central server. These features makes JWT ideal for stateless authentication in modern web and mobile applications [1, 11].

1.2.5 SAML 2.0

Security Assertion Markup Language is an XML-based authentication and authorization protocol that facilitates the secure exchange of identity information between an Identity Provider and a Service Provider. It enables cross-domain Single Sign-On, allowing users to sign in once and gain access to multiple services without repeated authentication. SAML operates through the use of SAML assertions which are security tokens that convey details about a user's authentication status and associated permissions. Widely adopted in enterprise environments, SAML is instrumental in integrating SaaS applications and managing federated identities, even as modern applications increasingly favor the flexibility and enhanced API support provided by protocols like OpenID Connect and OAuth 2.0 [1, 12].

1.3 Existing services on the market

Securing applications is a critical aspect of modern software development. Developers are increasingly relying on authorization and authentication platforms to manage user identities, streamline security processes, and reduce development overhead. Outsourcing these functions to specialized providers not only minimizes implementation time but also ensures compliance to evolving security standards [6].

In the current market there are multiple commercial offerings that provide authentication and authorization as a service. In order to evaluate and compare these solutions, this subchapter focuses on these aspects:

- **Industry standard protocols:** Support of protocols OAuth 2.0 and OpenID Connect.
- **.NET platform support:** Level of .NET support in particular the availability of an official SDK and sample projects.
- **Deployment:** A choice between *SaaS* solution, *self-hosting* solution, or *hybrid* solu-

tion models in the context of data control and availability.

- **Costs and licensing:** Open-source options offer greater flexibility, allowing organizations to customize the identity management stack and avoid vendor lock-in. Commercial solutions offers paid models, for which they provide developer support, or service management and monitoring.

1.3.1 Auth0

Auth0 is a leading provider in the authentication-as-a-service market, offering a highly customizable identity management solution. The service supports multiple identity protocols including OAuth 2.0, OpenID Connect, and SAML. Auth0 integrates smoothly with .NET applications via Software Development Kit (SDK). Its scalable design makes it suitable for organizations ranging from small startups to large enterprises. Licensing follows a subscription-based model: a free developer tier with limited monthly active users, followed by paid plans that scale according to monthly active users and feature sets as described in online documentation [13].

1.3.2 Azure Active Directory B2C

Azure Active Directory B2C is a SaaS cloud-based identity management platform designed specifically for customer-facing applications. It empowers organizations to offer seamless sign-up, sign-in, and profile management by integrating various identity providers. The platform supports protocols OAuth 2.0, OpenID Connect, and SAML and it offers official tooling for .NET, including SDK and templates. Solution is using PAYG licensing as described in online documentation [14].

1.3.3 Firebase Authentication

Firebase Authentication is a Google hosted SaaS solution designed to simplify user authentication for web and mobile applications. It supports a variety of sign-in methods including email/password, phone number, and federated identity providers such as Google, Facebook, and Twitter. Firebase Authentication natively supports protocols OAuth 2.0 and OpenID

Connect and SAML. Developers can integrate through the official SDK for .NET. Licensing follows a PAYG model with free quotas as described in online documentation [15].

1.3.4 Okta

Okta is a cloud-based identity management solution delivered as a SaaS platform. It supports multifactor authentication, single sign-on, and user lifecycle management. Integration with the .NET ecosystem is enabled through first-party SDKs for ASP.NET Core, Blazor and extensive REST APIs. Licensing is subscription-based (per active user / per feature tier) with enterprise add-ons for advanced policies and compliance as described in online documentation [16].

1.3.5 Keycloak

Keycloak is an open-source identity and access management solution designed to secure modern applications and services. It offers features such as single sign-on, multi-factor authentication, and social login, all of which simplify the process of managing user identities across diverse systems. Keycloak supports standard protocols including OAuth 2.0, OpenID Connect, and SAML 2.0. The administration console enables centralized management of users, roles, and permissions, while advanced capabilities like user federation, identity brokering, and clustering facilitate scalability and high availability in enterprise environments. Keycloak is released under the Apache 2.0 licence and is intended for self-hosted or hybrid deployment (container image, Kubernetes operator, etc.) as described in online documentation [17].

1.3.6 FusionAuth

FusionAuth is a identity management solution that provides secure authentication and authorization for modern applications. It offers a wide set of features, including user management, passwordless authentication, and detailed security analytics, to address a wide range of identity requirements. Solution supports protocols OAuth 2.0, OpenID Connect, and SAML. FusionAuth provides Community Edition released under the Apache 2.0 license, while ad-

vanced features are distributed under commercial license. Solution can be deployed as a self-hosted or as a fully managed cloud SaaS as described in online documentation [18].

1.4 Comparative Analysis and Considerations

When selecting an authentication and authorization service, developers should consider several key factors:

- **Integration with .NET:** Providers such as Auth0, Azure AD B2C, Okta and FusionAuth offer native libraries and detailed documentation tailored for .NET developers, reducing development time and enhancing security.
- **Scalability and deployment:** Cloud-based solutions like Firebase, Auth0 or Okta are engineered to scale dynamically, accommodating applications with rapidly growing or fluctuating user bases.
- **Licensing flexibility:** Open-source options such as IdentityServer, Keycloak, and FusionAuth offer greater flexibility, allowing organizations to customize the identity management stack and avoid vendor lock-in.
- **Cost considerations:** Commercial solutions such as Auth0 and Okta typically operate on a subscription-based model, whereas open-source options can present a more cost-effective solution when paired with in-house expertise.

These factors for the services mentioned are summarized in the following table 1.1:

Table 1.1 Comparison of SaaS solutions (source: own)

Service	OAuth/OIDC	.NET support	Deployment	Licensing
Auth0	Yes	Yes	SaaS	Free / Subscription
Azure AD B2C	Yes	Yes	SaaS	PAYG
Firebase Auth	Yes	Yes*	SaaS	PAYG
Okta	Yes	Yes	SaaS	Free / Subscription
Keycloak	Yes	Community	Self-host	Apache-2.0
FusionAuth	Yes	Yes	SaaS / Self	Free / Subscription

*Admin SDK only.

Each of the mentioned services presents unique trade-offs. For example Firebase Authentication is known for its simplicity and ease of setup, but it might not offer the granular control provided by solutions like Keycloak. Keycloak is a good example of a widely used service due to its open-source approach, but it can be difficult to deploy, manage and maintain. In contrast, cloud providers such as Azure AD B2C or Okta deliver good scalability and integration with broader cloud ecosystems, but they cause high operating costs.

A developer looking for a secure open-source solution with support for OAuth 2.0 and OIDC protocols that is lightweight, easy to use, deploy and monitor, doesn't have many options.

2 TECHNOLOGY ANALYSIS FOR SOLUTION DESIGN

This chapter establishes the technological foundation on which the proposed Authentication-as-a-Service platform is built and explains the reasons behind each choice. It is important to identify a modern, long-term-supported development framework that offers support for most commonly used authentication and authorization frameworks and helps reduce integration effort and development of the proposed solution. Next it is important to justify the selection of a high-throughput database engine and compatible Object-relational mapping (ORM) for satisfy durability requirements while simplifying data access patterns. Proposed solution must present data through responsive front-end which also needs to be selected. From an operational perspective, it is needed to discuss service deployment strategy for cloud and self-hosted environments. Scalable service also needs a load-balancing solution that distributes authentication traffic across multiple service instances, which will be also discussed [19, 20].

2.1 Framework

ASP.NET Core is a modern, high-performance framework that forms the backbone of scalable and secure enterprise applications. Its cross-platform capabilities and modular design enable developers to build robust solutions that can be efficiently maintained and easily extended. In the context of authentication and authorization services, ASP.NET Core provides comprehensive support for middleware, dependency injection, and integrated security features, which simplify the implementation of complex identity management systems. Its seamless integration with the broader .NET ecosystem further solidifies its role as the ideal foundation for developing secure, cloud-based authentication-as-a-service solutions [19, 21].

.NET 9 is the latest release of Microsoft's unified, cross-platform development stack. Its selection is motivated by two principal factors. First, the thesis requirements explicitly stipulate the use of the .NET ecosystem; adopting the latest version maximizes the technological lifespan of the resulting service and ensures immediate compatibility with current tooling and libraries. Second, .NET 9 introduces measurable improvements in just-in-time compilation, native ahead-of-time (AOT) publishing, and the ASP.NET Core request pipeline, collectively

reducing latency for authentication-intensive workloads and facilitating horizontal scalability in containerized deployments [22]. Although .NET 9 is classified as a Current rather than Long-Term Support release, its feature set, class library and support for OAuth 2.0 and OpenID Connect middleware make it suitable for the experimental and evaluative character of this study [21].

2.2 Authorization protocol

OAuth 2.0 is a modern protocol that facilitates secure and efficient authorization across distributed systems. It enables applications to delegate access to user resources through the use of access tokens, eliminating the need to directly share user credentials. This protocol offers a variety of grant types, such as authorization code, implicit, and client credentials flows, which allow it to address a wide range of use cases while maintaining robust security measures. Within the context of authentication-as-a-service, OAuth 2.0 is instrumental in establishing secure communication channels between clients and servers, ensuring that access to protected resources is both controlled and scalable [1, 4].

2.2.1 Authorization flows

The OAuth 2.0 protocol, as defined in RFC 6749 [4], clearly delineates the roles and responsibilities involved in the authorization process to ensure secure access to protected resources. OAuth 2.0 protocol defines these roles:

- **Resource owner** is typically the end-user who controls access to their protected data. End-user grants permission to access their data.
- **Client** is the application that seeks access to these protected resources on behalf of the resource owner. The client initiates the process by requesting authorization from the resource owner, which is managed by the *Authorization server*. In authorization process client can be web application, mobile application or different kind of client application and it is not representing user at this point.

- **Authorization server** authenticates the resource owner and, upon successful verification, issues an access token to the client.
- **Resource server** uses this token to grant the client access to the requested resources. Resource server can be protected API, or protected parts of website.

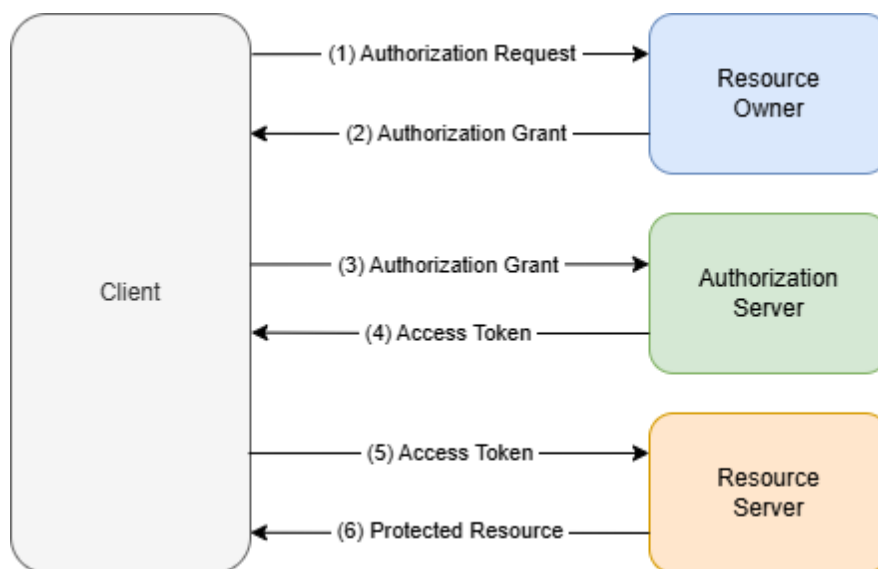


Figure 2.1 OAuth abstract protocol flow (diagram: own, source: [4])

OAuth protocol describes the interaction between roles in diagram 2.1. These interaction can be separated into following steps [4]:

- (1) The client initiates the process by requesting authorization from the resource owner, either directly or preferably through the authorization server acting as an intermediary.
- (2) The client receives an authorization grant, which is a credential representing the resource owner's consent, using one of the predefined or extended grant types.
- (3) The client then presents this authorization grant to the authorization server, along with its own credentials, in a request for an access token.
- (4) The authorization server authenticates the client and validates the authorization grant; if valid, it issues an access token to the client.
- (5) The client uses the access token to request the protected resource from the resource server.

- (6) Finally, the resource server validates the access token and, if it is valid, serves the requested resource.

To ensure robust security and seamless user experience, the solution must implement a token management flow that leverages short-lived access tokens for resource access and longer-lived refresh tokens for session continuity. By validating and rotating access tokens frequently, the system minimizes the attack window in case of token compromise, while refresh tokens allow clients to obtain new access tokens without forcing the user to re-authenticate on every expiration. This dual-token approach strikes a balance between minimizing risk and preserving usability across distributed clients and APIs [4].

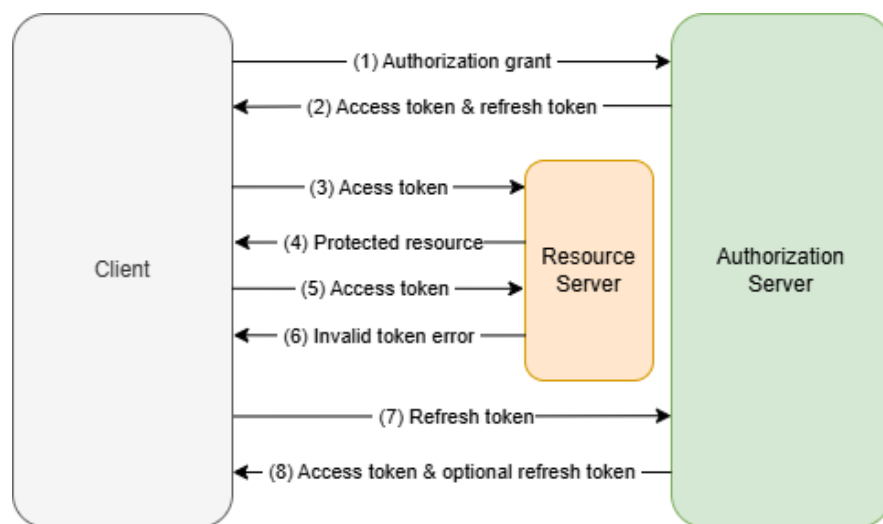


Figure 2.2 OAuth access token with refresh token (diagram: own, source: [4])

In case of access token and refresh token authorization flow, OAuth protocol describes the interaction between roles in diagram 2.2. These interaction can be separated into following steps [4]:

- (1) The client initiates the exchange by authenticating to the authorization server and submitting its authorization grant to request an access token.
- (2) The authorization server verifies the client's identity, checks the validity of the presented grant, and if all checks pass, issues both an access token and a refresh token.
- (3) Armed with the access token, the client then issues a request to the resource server to access a protected endpoint.

- (4) The resource server examines the incoming token; if it is still valid, the request is fulfilled and the resource is returned.
- (5) Steps 3 and 4 continue in a loop until the access token expires; at that point the client proceeds to the refresh-token exchange.
- (6) Should the access token be rejected as invalid before expiration, the resource server responds with an appropriate error (e.g. `invalid_token`).
- (7) To recover from expiration or invalidation, the client re-authenticates to the authorization server and submits its refresh token in order to obtain a new access token.
- (8) The authorization server validates the refresh token against its policies, and upon successful verification issues a fresh access token (and optionally rotates the refresh token).

2.3 Authentication protocol

OpenID Connect is an identity layer that extends the OAuth 2.0 authorization framework with a standardized mechanism for authenticating end-users and conveying identity claims. While OAuth 2.0 governs delegated access to protected resources, OIDC augments the protocol with the ID Token and user-info endpoints, thereby coupling access control with reliable user identification in a single, interoperable flow. OIDC was selected because it constitutes the de facto industry standard for modern web and cloud applications: it is supported by all major identity providers, benefits from mature .NET middleware, and offers well-defined discovery and metadata documents that simplify federation scenarios [9, 23].

OpenIddict is an open-source framework that delivers a complete OAuth 2.0/OpenID Connect stack for .NET applications, offering three composable components: a server engine, a universal client library, and a validation layer. The server package allows developers to stand up a fully-featured authorization server that issues JSON Web Tokens or opaque tokens, while the client package can be embedded in web, desktop, or mobile applications to negotiate standard OIDC flows without hard-coding provider endpoints thanks to automatic discovery. By encapsulating OAuth 2.0 mechanics beneath the higher-level OIDC abstractions, OpenIddict reduces boilerplate, ensures protocol compliance, and provides a uniform

programming model across ASP.NET Core. Consequently, the same library can be reused on both sides (client and server) of the trust boundary, exposing secure authorization endpoints and, in downstream services, consuming them, thereby simplifying maintenance and version alignment throughout the solution [24].

2.4 Database

PostgreSQL is a open-source relational database which was selected primarily as the author's preferred platform; the service could be deployed on alternatives such as MySQL or Microsoft SQL Server without fundamental changes to the data-access layer. In comparison with Microsoft SQL Server, PostgreSQL require no license fees, which reduces total cost of ownership in both on-premises and cloud deployments. Consequently, while the choice of PostgreSQL reflects personal preference, the design remains database-neutral and portable [25].

2.5 Object-relational mapping

Entity Framework Core is a open-source ORM that that aims to abstract the relational schema and eliminate handwritten SQL. It maps tables to strongly typed classes, provides automated change tracking, and offers a migration tool that propagates model changes to all environments in a repeatable manner. Queries are expressed with Language Integrated Query (LINQ), whose declarative syntax improves readability and composability. Although LINQ translation introduces minor runtime overhead compared with hand-tuned SQL, the gains in maintainability, refactor friendliness, and developer productivity outweigh the marginal performance cost for an authentication workload dominated by short, index-backed transactions [20]. Entity Framework is currently the most used framework by .NET developers [26].

2.6 Frontend

Razor Pages is the page-centric web framework introduced in modern .NET that overlays the Model View Controller (MVC) paradigm with a simplified file-per-page convention. Each page couples its markup and minimal PageModel class while still leveraging the established

MVC separation of concerns. This hybrid approach was selected for both the administrative user interface of the authentication service and for the demonstrative client applications, because it enables rapid scaffolding of forms, validation logic, and anti-forgery protections without introducing a separate single-page-application stack. At the same time, the underlying MVC infrastructure remains available, ensuring that more sophisticated interactions or future module additions can be added without architectural refactoring [20].

2.7 Deployment

Docker is a powerful containerization tool that plays a important role in the deployment of any web services in cloud environments. By encapsulating applications and their dependencies into portable containers, Docker ensures consistent and reliable deployments across different environments, from development to production. This approach not only streamlines the deployment process but also enhances scalability and resource efficiency, which are essential for cloud-based applications. Docker was selected because it offers support in Visual Studio, tight integration with GutHub Actions and GitLab runners and a rich catalogue of official images that simplify continuous integration pipelines [19].

2.8 Load-Balancing and API gateway

Yet Another Reverse Proxy (YARP) is an open-source reverse-proxy library for ASP.NET Core that adds programmable routing, health checks, and load-balancing directly inside a .NET application. Using YARP as the API gateway removes the need for a separate ingress service, reuses the same dependency-injection and middleware pipeline as the authentication server, and allows configuration changes without redeploying. This tight integration simplifies horizontal scaling, observability, and future extensions such as rate limiting or tenant isolation [27].

II PRACTICAL PART

3 SOLUTION GOALS AND USER STORIES

The authentication and authorization service is conceived as a shared component consumed by multiple web applications developed inside the organization or shared product domain. Expressing objectives from the perspective of the integrating software engineer makes it possible to translate high-level requirements into concrete, verifiable functionality [28]. To better understand what user goals the service is trying to solve, it is necessary to define server user stories. Each story therefore delineates a distinct capability that the AaaS solution must provide so that downstream applications can obtain secure sign-in, single sign-on, token life-cycle management, client onboarding, and centralized role administration without re-implementing security controls in every service tier.

3.1 User stories

1. Authenticate and authorize end-users of connected web applications

- **User story 1:** *As a developer, I want the service to take care of authenticating and authorizing every request to registered client pages and APIs so I don't have to implement or store custom login logic.*
- **Service feature:** OIDC Authorization code flow + PKCE issuing short-lived JWT access tokens and long-lived refresh tokens.
- **Service feature:**

2. Provide single sign-on (SSO) across relying parties

- **User story 2:** *As a developer, I want users to sign in once and then access any of my connected web applications without re-entering credentials so that user experience is seamless.*
- **Service feature:** Shared session cookie issued by the identity service, front-channel logout endpoint, and automatic token renewal through the `prompt=none` OIDC parameter.

3. Register client applications

- **User story 3:** *As a developer, I want a self-service endpoint where I can register new client apps and receive a `client_id` and hashed `client_secret` so that onboarding a micro-service or web applications is fast and consistent.*
- **Service feature:** Admin-only UI for client onboarding, enforced secret hashing and metadata validation (redirect URIs, grant types).

4. Expose internal OIDC endpoints

- **User story 4:** *As a developer, I want the service to publish a standard OpenID Connect discovery document and token endpoints so that my .NET applications can plug in with minimal configuration.*
- **Service feature:** Full OIDC discovery endpoint issuing signed ID tokens with configurable claims (`/.well-known/openid-configuration`).

5. Offer external OIDC federation (Google)

- **User story 5:** *As a developer, I want to enable Google sign-in for my web applications via one unified endpoint so I don't have to maintain separate OAuth flows for each OIDC provider.*
- **Service feature:** Identity brokering module supporting Google OIDC and automatic account linkage in login page.

6. Obtain explicit user consent for requested scopes

- **User story 6:** *As a developer, I want the service to display a consent screen during authentication so that end-users can explicitly approve the scopes (`openid`, `profile`, `offline_access`, etc.) requested by my application.*
- **Service feature:** Consent UI rendered by the identity server, persistent consent records, ability to revoke consent via user self-service portal.

7. Manage registered clients

- **User story 7:** *As a developer, I want to list existing client registrations through an API or admin UI with the possibility of canceling a client's registration.*
- **Service feature:** Razor-Pages admin console secured by the admin role, audit log of changes, bulk export (JSON/CSV).

8. Manage active tokens

- **User story 8:** *As a developer, I want an introspection and revocation API so I can query token status during troubleshooting and immediately invalidate tokens in case of suspected compromise.*
- **Service feature:** Token introspection endpoint, persistent refresh-token store, per-subject revocation list.

9. Manage user roles

- **User story 9:** *As a developer, I want to assign roles to users via the service so that authorization decisions are centralized and reflected in the token claims my apps consume.*
- **Service feature:** List of registered users with role-editing capability and role-based claim emission.

User stories describe the requirements from the perspective of a developer who is going to use the proposed service. Some scenarios (especially user stories 1, 2, 5) also describe the behavior of an end user who uses the developer's connected services.

Table 3.1 Overview of the defined user stories

User Story	Title
US1	Authenticate and authorize end-users of connected web applications
US2	Provide single sign-on (SSO) across relying parties
US3	Register client applications
US4	Expose internal OIDC endpoints
US5	Offer external OIDC federation (Google)
US6	Obtain explicit user consent for requested scopes
US7	Manage registered clients
US8	Manage active tokens
US9	Manage user roles

The user stories presented in 3.1 define capabilities the platform must provide, such as secure authentication and authorization, SSO, internal and external OIDC support, and necessary client, token, and role management. These concise requirements will set the direction for the design and validation of the architecture and implementation. Requirements and acceptance

criteria for fulfilling defined user stories are summarized in Chapter 4. The coverage of these features is demonstrated in Chapters 5–6.

4 REQUIREMENTS ANALYSIS

Requirements analysis establishes both *functional* and *non-functional* requirements for the proposed authentication and authorization service and serves as the foundation for the subsequent architectural design.

- **Functional requirements** translate the previously defined user stories into concrete capabilities that the system must provide, such as credential validation, federated sign-in, single sign-on, token issuance, and client onboarding, and attach measurable acceptance criteria and verification methods to each capability. In essence, they describe *what* the service must do to satisfy its users [29].
- **Non-functional requirements** specify the quality attributes that determine *how well* the service must perform these functions. They capture cross-cutting concerns including security (e.g. resistance to common attack vectors, mandatory transport encryption, memory-hard password hashing), performance (e.g. predictable response times under representative load), scalability, availability, maintainability, and deployability. For each attribute, qualitative acceptance thresholds and suitable validation strategies are defined so that the resulting solution is not only functionally correct but also robust, efficient, and sustainable throughout its life-cycle [29].

Together, these complementary requirement classes map the key usage scenarios, set measurable success criteria, and define the defensive controls that guide the architectural, implementation, and verification activities described in the following chapters.

4.1 Functional Requirements

1. US1 – Authenticate and authorize end-users

- **Acceptance Criteria 1:** A successful OIDC code+PKCE exchange returns an RS256-signed JWT access token with a time-to-live ≤ 15 min, ID token with a time-to-live ≤ 60 min and an opaque refresh token valid for 30 days.

- **Acceptance Criteria 2:** After five repeated login attempts within the same session, the user will be blocked under the rate-limiter.

A 15-minute lifetime for the RS256-signed access token keeps the replay window short while avoiding excessive silent renewals; shorter values would increase endpoint load, whereas longer values prolong exposure after leakage. The ID token is granted a more generous 60-minute lifetime because it is used only by the client for session management and is never presented to downstream APIs; this limits risk even with the extended lifetime and reduces needless re-authentication prompts [30]. A 30-day refresh-token lifetime offers a good balance between user convenience and fast incident response. High-security systems can shorten this period, while consumer applications sometimes choose a longer one [30].

Blocking a session after five consecutive failures follows OWASP guidance [29], deterring brute-force attempts without unduly punishing users for occasional typos. Adaptive throttling can replace the fixed threshold if usability testing reveals frequent lock-outs [31].

2. US2 – Provide single sign-on (SSO)

- **Acceptance Criteria 1:** After authenticating in *Application A*, the user navigates to *Application B* within the same browser session and is recognized without re-entering credentials.
- **Acceptance Criteria 2:** A global logout invalidates every remaining session and refresh token within 5 s.

Browser-based SSO is realized through a shared cookie issued by the AaaS solution. Recognition within the same user-agent provides the expected “sign-in once, roam everywhere” experience while preventing cross-device session replay [30].

A ≤ 5 s global-logout propagation target keeps the residual risk window small enough for security critical environments yet achievable with standard back-channel logout or token revocation push mechanisms. Tighter bounds ≤ 1 s require WebSocket or message bus fan-out at the cost of higher complexity, while wider bounds (approx. 1

min) are common in consumer SaaS where user perception of “logged out everywhere” is less time critical.

Alternatives include front channel logout (simpler but susceptible to tab/browser loss) or short-lived access tokens without back-channel revocation, shifting the burden to token expiry.

3. US3 – Register client applications

- **Acceptance Criteria 1:** An user with admin role may POST to `/clients` and receives a generated unique `client_id` and an Argon2id hash of the freshly generated `client_secret`.
- **Acceptance Criteria 2:** As part of client registration, it is necessary to specify allowed `redirect_uris`.

Client identifiers are generated as unique random strings, simplifying onboarding and avoiding guessable values. Client identifiers could alternatively be editable or user-enterable, but must remain unique [4]. The accompanying client secret is displayed only once and stored as an Argon2id hash to resist brute-force recovery [32]. Redirect URIs must be explicitly listed to prevent open-redirect attacks; wildcards are not accepted. Multi-tenant needs are addressed through separate client registrations [4].

4. US4 – Expose internal OIDC endpoints

- **Acceptance Criteria 1:** The discovery document lists all mandatory metadata fields and publishes a JWKS endpoint containing at least one active RSA-2048 key available from `/.well-known/openid-configuration`.

Providing a standards compliant discovery document and JWKS endpoint enables client libraries to obtain configuration and keys automatically, minimizing manual setup and ensuring key-rollover can be handled transparently [33].

5. US5 – Offer external OIDC federation (Google)

- **Acceptance Criteria 1:** Selecting “Sign in with Google” redirect user to web page `accounts.google.com` upon consent the returned ID token is verified and either linked to, or just-in-time creates, a local user.

Integrating Google as an external identity provider simplifies social login for end-users with their existing external accounts. Automatic account linking or creation streamlines onboarding while keeping a single identity store. The same broker mechanism can be extended with minimal effort to additional providers (e.g., Microsoft, GitHub, or Meta) should project requirements evolve.

6. US6 – Obtain explicit user consent for requested scopes

- **Acceptance Criteria 1:** When a client requests any scope the user has not previously approved, the authorization flow must display a consent screen enumerating each scope before issuing tokens.
- **Acceptance Criteria 2:** Selecting *Deny* must terminate the flow and redirect the client with the `error=access_denied` parameter in < 500 ms.
- **Acceptance Criteria 3:** A user's *Accept* decision must be persisted so that future identical requests from the same client skip the consent screen.

Explicit consent ensures the user clearly understands which personal attributes (e.g. profile, email) the relying party will receive before any token is issued, satisfying the transparency principle of modern privacy regulations [1, 10]. Persisting the user's *Accept* decision, revocable via a dedicated “Manage Consents” interface, strikes a pragmatic balance between security and usability [4]. Confining the redirection latency following an `error=access_denied` response to at most 500 ms curtails the interval in which phishing-oriented UI spoofing could occur. Deployments that demand stronger guarantees may elect either to re-prompt for consent on every authentication attempt or to enforce stricter latency threshold.

7. US7 – Manage registered clients

- **Acceptance Criteria 1:** The admin UI lists clients by name or `client_id`.
- **Acceptance Criteria 2:** Deleting a client prevents further token issuance and revokes all refresh tokens in ≤ 60 s.

Registered clients listing ensures that the administration console remains responsive even as the number of registered clients grows, while still permitting deterministic

lookup by either human friendly names or the immutable `client_id` [4]. The 60-second revocation window offers near-immediate risk mitigation without imposing excessive load on the token store. Deployments prioritizing tighter breach containment could shorten the window or employ synchronous revocation at the cost of higher latency and resource utilization.

8. US8 – Manage active tokens

- **Acceptance Criteria 1:** The admin UI lists active tokens and token owners.

Displaying active tokens alongside their associated principals equips administrators to correlate suspicious activity with specific users or clients and to perform targeted revocations. Installations with strict privacy constraints may expose only anonymized identifiers or restrict visibility to audit-log queries rather than real-time listings.

9. US9 – Manage user roles

- **Acceptance Criteria 1:** An admin can assign or remove predefined roles (such as user, admin). The next ID token issued to that user reflects the change.

Immediate propagation of role changes guarantees consistent authorization decisions across relying parties, eliminating the delay windows that could otherwise be exploited. Larger deployments with high issuance rates may instead refresh roles only at access-token renewal to reduce identity-server load, accepting a brief interval of eventual consistency.

4.2 Mapping of User Stories to Thesis Chapters

Table 4.1 correlates each developer-centric user story with the chapter or section in which it is primarily addressed.

Table 4.1 User story coverage across thesis chapters

User Story	Chapter / Section
US1	5.1 <i>Authentication Flow</i> 5.6 <i>Token Creation Implementation</i> 5.14 <i>Password Management</i>
US2	5.1 <i>Authentication Flow</i> 5.3 <i>Service Endpoints (Logout)</i> 5.4 <i>Client–Server Communication</i>
US3	5.12 <i>Client Registration</i>
US4	5.3 <i>Service Endpoints</i>
US5	5.8 <i>OIDC External Integration</i>
US6	5.2 <i>Authorization Logic</i> 5.4 <i>Consent Management in Client–Server Communication</i>
US7	5.9 <i>Service Frontend: Registered Clients List</i>
US8	5.7 <i>Refresh Token Support</i> 5.9 <i>Service Frontend: Active Tokens</i>
US9	5.9 <i>Service Frontend: Users and Roles</i>

4.3 Non-Functional Requirements

- **Security:** All endpoints enforce TLS 1.3; user credentials and client secrets are stored with Argon2id (64 MiB, 4 iterations). PKCE (S256) and a unique `jti` claim protect against code interception and replay.

Argon2id is configured with 64 MiB of memory and four iterations, matching the interactive profile in RFC 9106 [32]: enough to deter GPU based dictionary attacks without noticeable login delay. Lowering these values would cheapen offline cracking, whereas sharply higher settings could exclude memory constrained clients. Periodic tuning is therefore recommended.

- **Performance:** Median latency for token issuance ≤ 200 ms at 95th-percentile load (1 000 requests/s). Argon2id hash must complete in < 300 ms on the reference hardware.

A median token-issuance latency of 200 ms under the 95th-percentile load of 1 000 requests per second is set to keep the full authentication round-trip below the perceptual "instant" threshold recommended for interactive web applications [34], while still leaving a space for occasional network variance. Increasing this target (e.g., to 400

ms) would simplify cryptographic signing and database contention, but would degrade user perceived responsiveness and risk time-outs in mobile networks. Tightening it (e.g., 100 ms) would demand more aggressive caching and horizontal scaling, raising infrastructure cost.

- **Resilience:** Key endpoints (/login, /token) are rate-limited (5 requests / minute / IP address for login endpoints, 10 requests / minute / IP address for token endpoints) and protected by exponential back-off to mitigate credential stuffing and DoS attacks.

The threshold of 5 login attempts and 10 token requests per minute per IP is calibrated to stay below typical human interaction speed yet high enough to avoid false positives from page refreshes or programmatic renewals. The specific settings depend on the service requirements, for example, the GitHub REST API considers more than 60 requests per hour suspicious for unauthenticated users [35]. Reducing the limits (e.g., 2 requests/min/IP) would further protect credential stuffing but risks locking out users behind shared NAT gateways or aggressive mobile network retries. On the other hand, increasing them would lengthen the window in which automated attack tools can probe passwords or replay refresh tokens before detection. In environments with a high density of users behind a single address (such as enterprise proxies) IP-based quotas can be complemented with device fingerprint or user account keys, whereas deployments with low-risk profiles may adopt sliding-window algorithms tuned to higher volumes to minimize support incidents.

- **Deployability:** A multi-stage Dockerfile produces a minimal OCI image (< 220 MB).

Building a multi-stage Docker image under 120 MB limits download time, cold-start latency, and the attack surface. Allowing larger images (e.g., 400 MB SDK bases) simplifies builds but increases bandwidth, storage, and patching overhead, while going smaller with scratch or distro-less layers would harden security yet complicate debugging and observability. The chosen size thus balances operational efficiency with maintainable build complexity.

4.4 Quality Attributes and Design Decisions

To address the non-functional requirements established in the previous subsection, the following quality attributes provide a structured summary of each acceptance criterion and explain the corresponding architectural choices that have been made to satisfy them, thereby creating an explicit link between abstract system qualities, such as security, performance, resilience, and deployability, and the concrete design mechanisms implemented in the solution.

Security Acceptance Criteria:

- All endpoints must enforce HTTPS and reject plaintext HTTP requests.
- Passwords and client secrets must be stored using a memory-hard hashing algorithm that resists brute-force [32].
- Authorization codes and tokens must be protected against interception (PKCE) and replay (unique `jti` claim).
- Common attacks (brute-force, CSRF, token hijacking) must be mitigated via input validation, state parameters, and rate limiting.

Security Design Decisions:

- Enforce Transport Layer Security (TLS) on all endpoints.
- Use Argon2id with a 16-byte salt, 64 MiB memory, 4 iterations, parallelism 8 for both user passwords and client secrets [32].
- Implement PKCE (S256 only) and state validation to block code-injection and CSRF [36].
- Apply strict input length checks and JSON-encoding of all user-supplied values.

Performance Acceptance Criteria:

- Average response time for authentication and token requests must remain below 200 ms under typical load.
- Hashing operations must complete within acceptable bounds (e.g. < 300 ms per Argon2id computation).

Performance Design Decisions:

- Tune Argon2id parameters (64 MiB memory, 4 iterations) to balance resistance to GPU-based attacks and computation time.
- Cache validator instances to avoid repeated JIT or reflection overhead.

4.5 Implications for Architecture

The requirements outlined above drive a modular and flexible architecture. The need for components that can handle token creation, verification, and secure storage of identity data. The inclusion of SSO and detailed claims are important part for solution design. The non-functional requirements for lightweight performance and robust security underscore the importance of efficient coding practices and security measures, such as rate limiting. Quality attributes for this solution, including robust security (TLS enforcement, Argon2id hashing, PKCE and state validation), high performance (caching and optimized database indexing), and resilience against abuse (rate limiting and strict input validation), are integrated into every architectural layer. This set of requirements defines the architectural strategy to develop a secure, scalable, and maintainable authentication-as-a-service solution.

5 SOLUTION DESIGN AND IMPLEMENTATION

The architecture is designed to ensure secure, easy to use, and efficient identity management through clearly defined interactions between the system components. A high-level diagram (see Figure 5.1) illustrates the interactions between the user, the client application, and the Authorization-as-a-Service server.

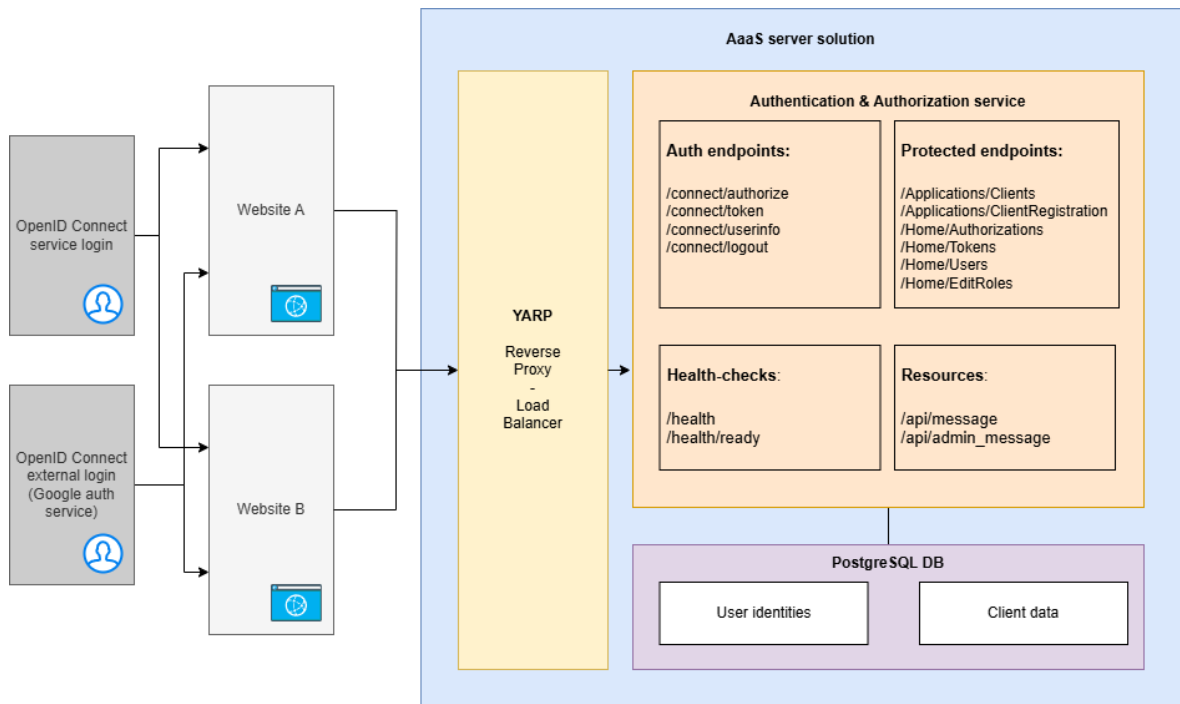


Figure 5.1 Solution architecture (source: own)

The AaaS solution is composed of four primary components:

- **User** who is accessing one of websites using OIDC authentication.
- **Client** in form of a website (Website A, Website B) that communicates with the authentication and authorization service.
- **PostgreSQL database** that stores identity information and configuration data securely.
- **Authentication and Authorization service** that handles user verification and token issuance.

At the core of the solution, the user interacts with the client application to access secured resources. When a user initiates an authentication request, the client application forwards

this request to the AaaS server. Once the user's credentials are verified, the AaaS server issues an access token, which the client application subsequently uses to access protected resources, id token with user identity and refresh token in form of a JWT bearer token. This architecture not only secures the user's credentials but also streamlines the overall process, enabling a seamless integration and robust identity management solution.

5.1 Authentication and authorization flow

The authentication and authorization flow consists of two distinct phases that ensure secure access to protected resources. Initially, the authentication phase verifies the user's identity by having the client submit appropriate credentials, which the server validates against the stored records in the database. Upon successful verification, the user is redirected to the authorization endpoints, where the system issues an access token after confirming the granted permissions. This access token then enables the client to request and access protected resources, ensuring that only authenticated and authorized entities can interact with protected resources [2].

Authentication typically occurs using one of two common schemes, each with its own mechanism for managing user sessions and credentials [2].

- **Cookie:** In this approach, following successful user authentication, a session cookie is created and stored on the client side. The cookie is then automatically included with each subsequent request, allowing the server to maintain stateful session management [37].
- **JWT Bearer Token:** In this scheme, the server issues a JWT upon successful authentication. The token, which encapsulates the necessary claims, is then provided by the client in the HyperText Transfer Protocol (HTTP) Authorization header for each request, enabling stateless and scalable verification [11].

5.2 Authentication flow

Authentication represents the primary security control of the proposed service, ensuring that tokens are issued exclusively to legitimate users before any access to protected resources is granted. The implementation combines ASP.NET Core Identity with the OpenIddict framework, while unifying local user management with the OpenID Connect protocol and follows Authorization Code flow with PKCE.

When a client browser is redirected to `/connect/authorize` endpoint, the controller extracts the `OpenIddictServerRequest` from the HTTP context. This object captures the client identifier, requested scopes, prompt values, and any duration constraint supplied by the relying party. The ASP.NET Core authentication middleware is invoked via method `AuthenticateAsync` to determine whether a valid session cookie is already present.

- **Unauthenticated or re-authentication required:** If no session exists, or if the request mandates re-authentication through `prompt=login` or an expired `max_age`, the controller issues a challenge that redirects the user to the server-hosted login GUI. All original query parameters are preserved so that, upon successful login, the user is returned seamlessly to the authorization flow. If the client requires `prompt=none`, the server responds with `error=login_required`, as interactive authentication is disallowed in that case as described in code example 5.1.
- **Authenticated:** After successful credential verification, the corresponding *ApplicationUser* entity is retrieved via `userManager`. The user's claims and roles are assembled, and any outstanding consent requirements are evaluated against the application's configured consent mode (implicit, explicit, external, or systematic). If additional consent is necessary, a Razor-based consent page enumerates the requested scopes, otherwise, the flow advances directly to token issuance.

```

1 // Get authentication result from http context
2 var result = await HttpContext.AuthenticateAsync();
3
4 // Check the authentication result and OpenIdDict request
5 if (result is not { Succeeded: true } ||
6     ((request.HasPromptValue(PromptValues.Login) || request.
7         MaxAge is 0 ||
8         (request.MaxAge != null && result.Properties?.IssuedUtc
9             != null &&
10             TimeProvider.System.GetUtcNow() - result.Properties.
11                 IssuedUtc > TimeSpan.FromSeconds(request.MaxAge.
12                     Value))) &&
13     TempData["IgnoreAuthenticationChallenge"] is null or
14     false))
15 {
16     // If the user is not logged in, redirect to the login
17     // page
18     TempData["IgnoreAuthenticationChallenge"] = true;
19     // Return a challenge to the OpenIdDict server
20     // authentication scheme
21     return Challenge(new AuthenticationProperties
22     {
23         RedirectUri = Request.PathBase + Request.Path +
24             QueryString.Create(
25                 Request.HasFormContentType ? Request.Form :
26                 Request.Query)
27     });
28 }

```

Listing 5.1 Authentication example [9]

The authenticated identity with subject identifier, name, email, and role claims is finally passed to OpenIdDict, which issues an ID token, an access token, and, if requested, a refresh token. Embedding the user's claims within these tokens enables downstream APIs to enforce fine-grained authorization policies without secondary lookups, while adherence to OIDC ensures interoperability with other standards compliant clients and external identity providers (e.g., Google) integrated through federation.

5.2.1 Authorization logic

Authorization governs which authenticated principals may access specific client applications and scopes. The implementation address to the OpenID Connect specification and employs OpenIdDict's extensibility mechanisms to enforce consent-driven, fine-grained access control [33].

Each client is assigned a consent type: implicit, explicit, external, or systematic. After successful authentication, the controller retrieves any existing authorizations binding

the user, client, and requested scopes.

- If a matching authorization exists and the consent type allows silent approval (e.g., implicit), the consent step is skipped and tokens are issued immediately.
- If no authorization exists and consent is required (explicit or external), the user is prompted unless the request contains `prompt=none`, in which case the flow terminates with `error=consent_required`.

When consent is satisfied, the system constructs a `ClaimsIdentity` containing subject, email, preferred username, and role claims. Requested scopes are applied to constrain the claim set and determine token destinations (access versus ID token). The finalized identity is passed to `OpenIddict`, which signs and issues the tokens: access token (15 min), ID token (60 min), and refresh token (30 days).

Listing 5.2 summarizes the consent decision tree: external clients without prior authorization are denied; implicit or previously consented requests proceed; explicit or systematic clients with `prompt=none` are refused; all other cases render the consent page.

```

1 switch (await _appManager.GetConsentTypeAsync(application))
2 {
3     case ConsentTypes.External when authorizations.Count ==
4         0:
5         return Forbid(...);
6     case ConsentTypes.Implicit:
7     case ConsentTypes.External when authorizations.Count !=
8         0:
9     case ConsentTypes.Explicit when authorizations.Count != 0
10         &&
11         !request.HasPromptValue(PromptValues.Consent):
12         // Issue tokens without extra consent
13         return SignIn(...);
14     case ConsentTypes.Explicit when request.HasPromptValue(
15         PromptValues.None):
16     case ConsentTypes.Systematic when request.HasPromptValue(
17         PromptValues.None):
18         return Forbid(...);
19     default:
20         // Render consent UI
21         return View(...);
22 }
```

Listing 5.2 Consent handling in the authorization controller [24]

This design keeps authorization decisions user-centric, leverages standard OIDC constructs,

and provides a flexible foundation for secure interactions across distributed client–server deployments.

5.3 Service Endpoints

The secure interaction between client applications and the AaaS server is realized through several HTTP endpoints that together implement the OpenID Connect and OAuth 2.0 flows. Each endpoint enforces the security and performance requirements set out in Chapter 4. All service endpoints are summarized in table 5.1.

Table 5.1 Service endpoints, methods, and their functions

Endpoint	Methods	Description
/connect/authorize	GET, POST	Initiates OIDC authorization flow (authentication and consent)
/connect/token	POST	Exchanges code or refresh token for access/ID tokens
/connect/logout	GET, POST	Signs the user out and terminates session
/connect/userinfo	GET	Returns authenticated user's profile claims
/connect/revoke	POST	Revokes a given refresh token
/callback/login	GET, POST	Handles external OIDC provider callback (e.g. Google)
/Applications/Clients	GET, POST	Lists and manages registered client applications
/Applications/ClientRegistration	GET, POST	Provides form to onboard new clients
/Home/Users	GET	Displays all users and their roles
/Home/EditRoles	GET, POST	Allows modification of user role assignments
/Home/Tokens	GET	Shows active refresh tokens per user
/Home/Authorizations	GET	Lists active user–client authorizations
/Health	GET	Liveness probe indicating service health
/Health/Ready	GET	Liveness probe indicating cild services health (e.g. database)
/.well-known/openid-configuration/	GET	Provides OIDC discovery metadata and JWKS

Authentication and authorization endpoints are described the following paragraphs.

5.3.1 Authorization endpoint /connect/authorize

This endpoint initiates the OIDC authorization code flow and accepts both GET and POST verbs. After parsing the incoming request containing client identifier, requested scopes, PKCE challenge, and prompt parameters the controller either redirects unauthenticated users to the local login page, evaluates consent rules for authenticated users, or issues OIDC-compliant error responses (e.g. `login_required`, `consent_required`). Successful requests progress to the token stage or render a consent form when explicit approval is mandated.

5.3.2 Consent handlers /connect/authorize (Accept / Deny)

Form submissions from the consent page are routed back to the authorization endpoint and distinguished by the presence of `submit.Accept` or `submit.Deny`. Acceptance creates (or reuses) a permanent authorization record and triggers token issuance. Denial returns an OIDC Forbid result, signalling `error=access_denied` to the client.

5.3.3 Token endpoint /connect/token

Accessible via POST method, this endpoint exchanges authorization codes or refresh tokens for signed access, ID, and refresh tokens. It validates the grant type, PKCE verifier, client credentials, and user sign-in status before generating new tokens embedded with the user's claims and approved scopes. Failure to satisfy any validation rule yields an `invalid_grant` or `invalid_client` error.

5.3.4 Logout endpoint /connect/logout

Supporting GET (initiation) and POST (confirmation) methods, this endpoint signs the user out of ASP.NET Core Identity, instructs OpenIddict to propagate front-channel logout, and finally redirects the browser to the post-logout URI. This procedure ensures that both local and federated sessions are invalidated, eliminating residual authentication state.

Together, these endpoints constitute the public API of the service and form the basis for the authentication and authorization scenarios.

5.3.5 Client-server communication

Secure and interoperable interaction between relying parties and the AaaS server is achieved through the OpenID Connect protocol, which augments OAuth 2.0 with an identity layer. The following paragraphs describe how the implementation applies OIDC messages, manages user consent, and handles error conditions.

Client applications initiate authentication by redirecting the browser to a protected endpoint `/connect/authorize` with parameters such as `client_id`, `redirect_uri`, `scope`, and `response_type`. The server validates the client registration and, after successful authentication, exchanges the resulting authorization code at `/connect/token` endpoint for an access token and an ID token containing user claims and approved scopes. All requests are conveyed over HTTPS, safeguarding confidentiality and integrity.

For each client the server enforces a consent type (implicit, explicit, external, or systematic). Where explicit consent is required and no prior authorization exists, the user is shown a consent view summarized in Listing 5.3. The decision is persisted and reused on subsequent requests, thereby aligning with privacy regulations and user-centric security principles.

```
1 return View(new AuthorizeViewModel {  
2     ApplicationName = await _applicationManager.  
3         GetLocalizedDisplayNameAsync(app),  
4     Scope = request.Scope  
5 });
```

Listing 5.3 Rendering the consent page when explicit approval is needed

In case of errors the server returns OIDC compliant error responses to ensure clients can react deterministically:

- Parameter `login_required` issued when `prompt=none` is set and no authenticated session exists.
- Parameter `consent_required` issued when explicit consent is mandatory but absent or denied.
- Parameter `invalid_grant` / `invalid_client` issued for expired codes, revoked refresh tokens, or untrusted client credentials.

These errors are delivered via appropriate HTTP status codes and structured JSON bodies, enabling clients to present clear feedback and avoiding ambiguous failure modes, thereby strengthening the overall security posture of the system.

5.3.6 Token types and lifecycles

OAuth2.0 introduces three tokens: access token, ID token, and refresh token that together support secure, stateful communication between clients and the authorization server. Each token fulfills a distinct role and is governed by an independent expiration policy.

- **Access token** grants the client temporary authority to call protected APIs on the user's behalf. In the implementation it is RS256-signed and expires 15 minutes after issuance, thereby limiting exposure if the token is leaked.
- **ID token** conveys the user's identity (subject identifier, name, email, roles) to the client. It is consumed only by the client, not by downstream APIs, and therefore carries a longer 60 minutes lifetime.
- **Refresh token** enables the client to obtain new access and ID tokens without re-authentication, supporting long-lived sessions. A validity of 30 days balances convenience with rapid incident response. Tokens are revoked on logout or security events.

Tokens are issued exclusively by the authorization server and transmitted over TLS. Clients are expected to store access and ID tokens in ephemeral, XSS-resistant locations (e.g., memory or secure browser storage) [38]. Refresh tokens, being more sensitive, undergo server-side checks on every use: user status, client credentials, and scope alignment are validated prior to re-issuance (Listing 5.4).

```

1 if (request.IsAuthorizationCodeGrantType() || request.
   IsRefreshTokenType())
2 {
3     var result = await HttpContext.AuthenticateAsync(
        OpenIddictServerAspNetCoreDefaults.
        AuthenticationScheme);
4     var user = await _userManager.FindByIdAsync(result.
        Principal.GetClaim(Claims.Subject));
5
6     if (user is null)
7         return Forbid(...);
8
9     if (!await _signInManager.CanSignInAsync(user))
10        return Forbid(...);
11
12    // Create a new identity based on the claims from the
13    authentication result
14    var identity = ...
15    // Return a SignInResult will ask OpenIddict to issue the
16    appropriate access/identity tokens.
17    return SignIn(new ClaimsPrincipal(identity),
        OpenIddictServerAspNetCoreDefaults.
        AuthenticationScheme);
18 }

```

Listing 5.4 Validation logic for authorisation-code and refresh-token grants

All exchanges occur over encrypted channels, and refresh tokens are never forwarded to third-party services. Revocation and logout endpoints allow users or administrators to invalidate tokens proactively, ensuring confidentiality, integrity, and availability throughout the token lifecycle.

5.3.7 Token creation implementation

Token issuance constitutes the final step of the authentication flow, translating the authenticated principal and the client's requested permissions into cryptographically signed artifacts. The implementation relies on OpenIddict's token factory while ClaimsIdentity is assembling that captures user attributes and approved scopes.

Claims represent verifiable statements about the user and are propagated to downstream ser-

vices to support authorization decisions. Immediately after authentication the server constructs an identity object and populates it with subject, email, name, preferred username, and role claims retrieved from the ASP.NET Core Identity store as shown in listing 5.5.

```
1 var identity = new ClaimsIdentity(  
2     authenticationType: TokenValidationParameters.  
3         DefaultAuthenticationType,  
4     nameType: Claims.Name,  
5     roleType: Claims.Role);  
6     identity.SetClaim(Claims.Subject, await _userManager.  
7         GetUserIdAsync(user));  
8     identity.SetClaim(Claims.Email, await _userManager.  
9         GetEmailAsync(user));  
10    identity.SetClaim(Claims.Name, await _userManager.  
11        GetUserNameAsync(user));  
12    identity.SetClaim(Claims.PreferredUsername, await  
13        _userManager.GetUserNameAsync(user));  
14    identity.SetClaims(Claims.Role, [... (await _userManager.  
15        GetRolesAsync(user))]);
```

Listing 5.5 Claim population prior to token issuance

Scopes delimit the breadth of access requested by the client application. They are extracted from the incoming OIDC request (openid, profile, offline_access, etc.) and applied to the identity as shown in listing 5.6.

```
1 identity.SetScopes(request.GetScopes());
```

Listing 5.6 Set scope example

Assigning scopes at this stage enables OpenIddict to include only the necessary claims in each token and to populate the corresponding audience / resource declarations. The resulting tokens therefore conform to the principle of least privilege, carrying just enough information for the relying party to enforce its own access-control rules.

Once the claim set and scope list are finalized, OpenIddict signs the payload and returns an access token, ID token, and (when requested) a refresh token, each with the lifetime policies defined in Section 5.3.6.

5.3.8 Refresh Token Support

Refresh tokens extend session longevity by allowing clients to obtain fresh access (and ID) tokens without re-prompting the user for credentials. In the implemented solution, refresh-token grants are processed at the `/connect/token` endpoint alongside the authorization-code flow.

When a client submits a valid refresh token via a POST to `/connect/token`, the server re-authenticates the token and resolves the associated user principal (Listing 5.7). If the user record is missing or sign-in is no longer permitted, an `invalid_grant` error is returned. Otherwise, the system issues a new access token (15 min) and, if requested, an ID token (60 min), each bearing the original scopes and claims.

```
1 if (request.IsAuthorizationCodeGrantType() ||
2     request.IsRefreshTokenGrantType())
3 {
4     var result = await HttpContext.AuthenticateAsync(
5         OpenIddictServerAspNetCoreDefaults.
6         AuthenticationScheme);
7     var user = await _userManager.FindByIdAsync(result.
8         Principal.GetClaim(Claims.Subject));
9
10    if (user is null)
11        return Forbid(...);
12
13    if (!await _signInManager.CanSignInAsync(user))
14        return Forbid(...);
15
16    // Issue new tokens with updated expiry...
17 }
```

Listing 5.7 Core logic for refresh-token grant

Refresh tokens are protected by multiple controls:

- **Validation against account state:** Tokens are rejected if the user account is disabled or deleted.
- **Revocation and expiry:** Tokens expire after 30 days and can be revoked on logout or by administrators.
- **Secure channels and storage:** All refresh-token exchanges occur over HTTPS. Clients must store tokens in non-persistent, XSS-resistant storage.

- **Scope enforcement:** Newly issued tokens inherit only the scopes and audiences originally granted to prevent privilege escalation.

These measures ensure that the refresh-token mechanism enhances usability without undermining the overall security posture of the service.

5.3.9 OIDC external integration

Federating with third-party identity providers via the OpenID Connect protocol enhances both usability and security by allowing users to authenticate with familiar accounts (e.g. enterprise directories or social logins). The implemented solution employs OpenIddict to act simultaneously as an OpenID Connect provider for internal clients and as a relying party for external identity sources. Example of a external OIDC connection to Google auth services is shown in listing 5.8.

```
1 options.UseWebProviders()
2     .AddGoogle(options =>
3     {
4         options.SetClientId("google-client-id")
5         .SetClientSecret("google-client-secret");
6         .SetRedirectUri("callback/login/google");
7     });
8 }
```

Listing 5.8 External OIDC integration example

OpenIddict encapsulates the OAuth 2.0 and OIDC workflows within the same ASP.NET Core application, managing endpoints such as /connect/authorize and /connect/token in a standard compliant manner. When configured as a relying party, the system redirects users to an external provider (for example, Google), validates the returned ID token, maps external claims into a local ApplicationUser, and issues its own tokens without exposing underlying protocol complexity to the client.

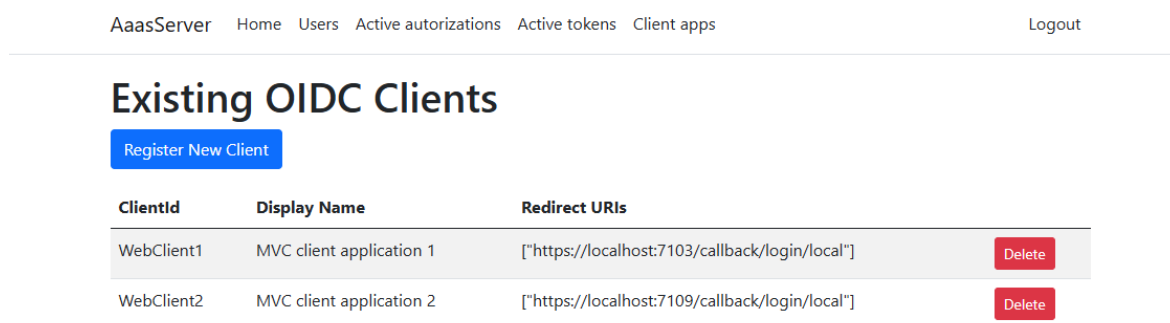
External clients are governed by the same consent model as first-party applications. If a client's consent type is External and no valid authorization record exists, the flow terminates with an OIDC compliant error.

This enforcement ensures that only users with explicit external authorization may proceed, preserving a consistent, user-centric consent experience across both internal and federated scenarios. Administrative APIs allow revocation and review of external authorizations, granting users and operators precise control over which external applications can access their data.

5.3.10 Service frontend

The administrative interface is implemented with ASP.NET Core Razor Pages, offering a set of multiple views for client and token management as well as user and role administration. Each page is described below.

Registered clients list is accessible at `/Applications/Clients` endpoint. This page displays a table of all client applications, showing `client_id`, display name, and status. A “Register New Client” button invokes the client registration form as shown in the screenshot 5.2.



Existing OIDC Clients			
Register New Client			
ClientId	Display Name	Redirect URIs	
WebClient1	MVC client application 1	["https://localhost:7103/callback/login/local"]	Delete
WebClient2	MVC client application 2	["https://localhost:7109/callback/login/local"]	Delete

Figure 5.2 Clients list UI (source: own)

New client registration is the `/Applications/ClientRegistration` page presents a form for administrators to enter the application name, allowed redirect URIs, and required grant types. Upon submission, the server generates a `client_id` and `client_secret`, displays them once, and persists the hashed secret as shown in figure 5.3.

AaaSServer

Home

Users

Active authorizations

Active tokens

Client apps

Logout

New OIDC client registration

Client ID

BinktlvD

Display Name

BinktlvD

Client Secret

.....

Generate

Show

Copy

Allowed redirect URIs

https://example.com/singin

Allowed post Logout Redirect URIs

https://example.com/singout

Register client

Figure 5.3 Register new client form (source: own)

Active authorizations is the page under /Home/Authorizations. The UI lists all valid consent records (user–client–scope tuples), indicating which users have approved which applications. Administrators may revoke individual authorizations directly from the table as shown in figure 5.4.

AaaSServer

Home

Users

Active authorizations

Active tokens

Client apps

Logout

Active authorizations

Id	Subject	Application	Scopes	Creation Date	Status	Action
73670db7-18ff-4c34-89ad-e53305a2c6be	447314da-d522-47be-9698-b552da5e3fda	WebClient2	["openid","email","profile","roles","offline_access"]	05/29/2025 16:06	valid	Revoke
925d12bc-29c6-4b14-93fe-fefef0683aa0	447314da-d522-47be-9698-b552da5e3fda	WebClient1	["openid","email","profile","roles","offline_access"]	05/29/2025 16:06	valid	Revoke

Figure 5.4 List of active authorizations (source: own)

Active tokens is the /Home/Tokens page tabulates all non-expired refresh tokens, showing associated user, client, issue date, and expiration. A “Revoke” action invalidates the selected token and immediately blocks its further use as shown on figure 5.5.

AaaSServer Home Users Active autorizations Active tokens Client apps							Logout
Active tokens							
Id	Subject	Application	Type	Creation Date	Expiration	Status	Action
54e3f451-9283-47f3-8e5e-322604b0b902	447314da-d522-47be-9698-b552da5e3fda	WebClient1	authorization_code	05/29/2025 16:14	05/29/2025 16:19	redeemed	
450b151a-91f4-49be-b74e-8f8e3bfa46d4	447314da-d522-47be-9698-b552da5e3fda	WebClient1	access_token	05/29/2025 16:14	05/29/2025 16:29	valid	
9219f238-3210-49bc-8dae-edf1ac5b9fe0	447314da-d522-47be-9698-b552da5e3fda	WebClient1	refresh_token	05/29/2025 16:14	06/28/2025 16:14	valid	Revoke
9ce5e6cd-efb2-4eb4-84cc-02c16fa12193	447314da-d522-47be-9698-b552da5e3fda	WebClient1	id_token	05/29/2025 16:14	05/29/2025 17:14	valid	
a0e9bdbc-cb64-47d7-9c03-3c5fb7a09c0e	447314da-d522-47be-9698-b552da5e3fda	WebClient2	authorization_code	05/29/2025 16:14	05/29/2025 16:19	redeemed	
249acc8c-34df-4c07-b4db-c0f712e96478	447314da-d522-47be-9698-b552da5e3fda	WebClient2	access_token	05/29/2025 16:14	05/29/2025 16:29	valid	
4b5b1349-568c-427b-88b7-7e796d05c508	447314da-d522-47be-9698-b552da5e3fda	WebClient2	refresh_token	05/29/2025 16:14	06/28/2025 16:14	valid	Revoke
df510070-d35f-4d6d-b300-af5e82e4d385	447314da-d522-47be-9698-b552da5e3fda	WebClient2	id_token	05/29/2025 16:14	05/29/2025 17:14	valid	

Figure 5.5 List of active tokens (source: own)

Users and roles is a /Home/Users page where administrators see a user directory with assigned roles. Each row includes an “Edit Roles” action, opening a modal where roles (admin, user, etc.) can be toggled. Changes take effect on the next token issuance. Each user can be deleted as shown on the figure 5.6.

AaaSServer Home Users Active autorizations Active tokens Client apps							Logout
List of users							
Id	E-mail	Role					
447314da-d522-47be-9698-b552da5e3fda	test@test.com	user_admin	Delete				
b11fc886-1644-472f-9f97-6323e915371f	martin.gabriel91@gmail.com	admin	Delete				
3ee2f1ba-deff-437b-9e1f-d869e13f8083	test@utb.cz	user	Delete				

Figure 5.6 List of users (source: own)

Access denied example shows when an authenticated user attempts to access a protected resource for which they lack permission, the Razor Pages UI returns a standard error view (e.g. Error/403.cshtml) displaying a clear “Access Denied” message and a link back to the dashboard as shown the figure 5.7 with example of a user with user role trying to access Table of Users, which is restricted only to users with admin role.

Access denied

You do not have access to this resource.

Figure 5.7 List of active authorizations (source: own)

All pages enforce the admin or user role policies as appropriate and integrate client-side validation for all forms, ensuring both security and responsiveness.

5.3.11 Service deployment

The entire solution is packaged and deployed as containers using Docker and orchestrated via Docker Compose in a single `docker-compose.yml` file. Each component runs in its own service, which is described in diagram 5.8.

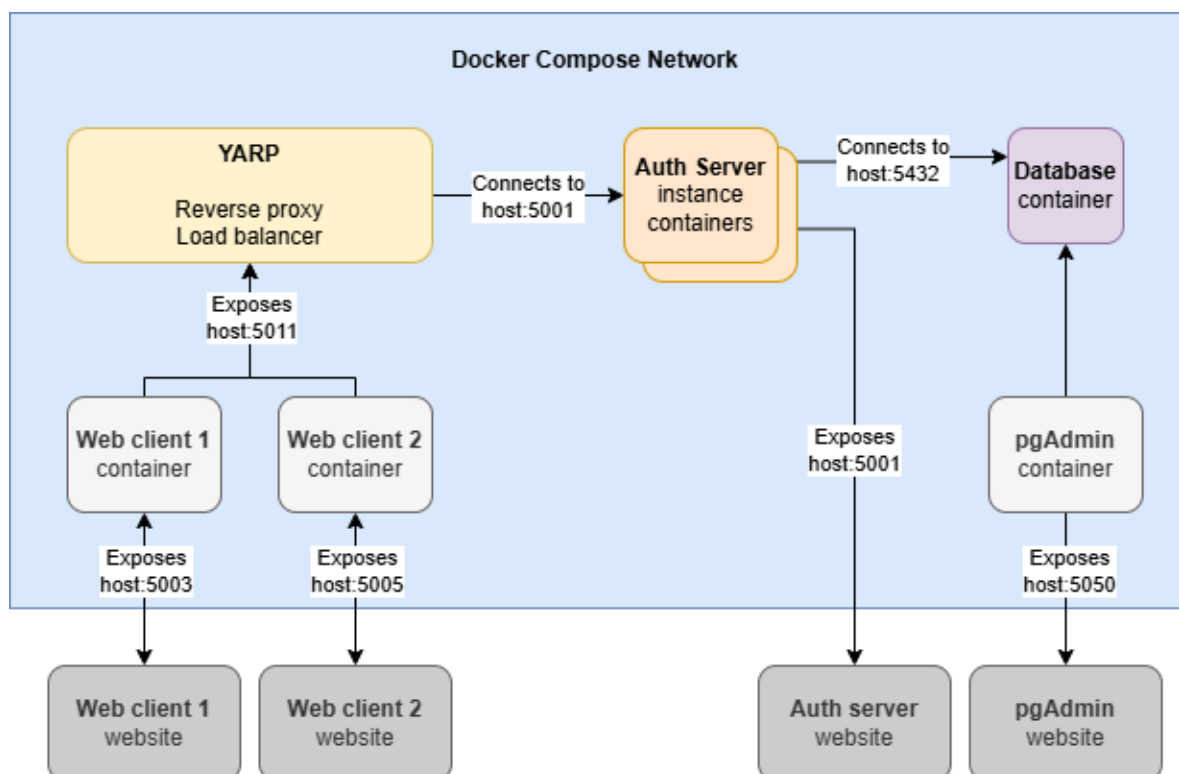


Figure 5.8 Docker compose network diagram (source: own)

- **YARP gateway:** Built from `YARP_LoadBalancer/Dockerfile`, this container ex-

poses port 5011 on the host. It is a reverse proxy and load balancer between clients and auth server. All clients communicate through this gateway.

- **Auth server:** Built from `AaaSServer/Dockerfile`, this container exposes port 5001 (API) on the host and is available on `https://aaasserver:5001`. Receives requests from clients via YARP gateway and communicate with database inside the Compose network.
- **Example web client 1:** Built from `WebClient1/Dockerfile`, this container exposes port 5003 on the host. It relies on Server through YARP gateway being available at `https://webclient1:5003` in the Compose network.
- **Example web client 2:** Built from `WebClient2/Dockerfile`, this container exposes port 5005 on the host. It relies on Server through YARP gateway being available at `https://webclient2:5005` in the Compose network.
- **Database:** A PostgreSQL database container (using the official `postgres:latest` image) listens on host port 5432:5432. It initializes the database with user and password, and persists data in a Docker volume.

By running the docker compose image all services start in a dedicated network, with automatic DNS resolution (e.g. `webclient1:5003`). This approach fulfills non-functional deployment requirements and ensures:

- **Consistency:** Every developer and environment uses the same container images and configuration [19].
- **Isolation:** Services run independently, preventing host-level conflicts [19].
- **Scalability:** Individual services can be scaled horizontally (e.g. multiple `AaaSServer` instances) behind a YARP load balancer [39].
- **CI/CD Integration:** Containers can be built, tested, and deployed automatically via pipelines, enabling zero-downtime rolling updates [19].

6 SECURITY AND RESILIENCE

In the modern digital ecosystem, where web applications increasingly serve as gateways to critical services and sensitive data, the security and resilience of authentication and authorization mechanisms are important points in system security. As primary entry points into systems, these components are frequent targets of cyberattacks aimed at exploiting vulnerabilities in identity verification and access control.

Common threats include brute force attacks, credential stuffing, phishing, and session hijacking, each capable of undermining trust and compromising system integrity. Attackers often exploit weaknesses in session management, insufficient input validation, and insecure HTTP configurations to gain unauthorized access or disrupt service availability. Beyond technical flaws, even auxiliary features such as account recovery can introduce critical vulnerabilities when improperly implemented [37].

Effective defense requires a layered approach that integrates secure authentication protocols, robust input validation, and proactive session management, which were designed in the final solution and are described in the following subsections.

6.1 Threat modeling

To systematically assess the security of the proposed solution, it is needed to apply the STRIDE model [40], identifying representative threats in each category:

- **Spoofing:** An attacker forges a client's identity by stealing its client id and client secret, or by replaying a valid access token, thus gaining unauthorized access to protected resources.
- **Tampering:** Malicious modification of the authorization request parameters (e.g. manipulation of `redirect_uri` or `state`), or of JWT assertion contents, may bypass validation checks and redirect tokens to an attacker-controlled endpoint.
- **Repudiation:** Insufficient logging or lack of cryptographic non-repudiation allows a

user, client, or administrator to deny having performed a sensitive action (e.g. issuing or revoking tokens), complicating audit and forensics.

- **Information disclosure:** Exposure of tokens or user claims through insecure transport (e.g. missing TLS), verbose error messages, or overly permissive Cross-origin resource sharing policies can leak credentials or Personally Identifiable Information to eavesdroppers.
- **Denial of Service:** Flooding the `/connect/authorize` or `/connect/token` endpoints with mock requests can exhaust CPU or database resources, preventing legitimate authentication and authorization operations.
- **Elevation of privilege:** A compromised or misconfigured client or server instance could issue tokens with excessive scopes or roles, allowing an attacker to escalate privileges beyond their intended access.

The mitigation of these threats is described in the following subchapters.

6.2 Endpoint security and resilience

The robustness of the authentication and authorization service is underpinned by multiple layers of endpoint protection, combining declarative attributes, middleware policies, and explicit validation to enforce authentication, authorization, and input integrity.

6.2.1 Declarative protection

All sensitive controllers and actions are annotated with the `[Authorize]` attribute, ensuring that only authenticated users may invoke them as can be seen in listing 6.1.

```
1 [Authorize]
2 public class ApplicationController : Controller
3 {
4     // ...
5 }
```

Listing 6.1 Endpoint authorize notation example

Any unauthenticated request to these actions results in an automatic challenge (redirect to login) or a 401 Unauthorized response, consistent with the OIDC specification.

6.2.2 Anti-forgery and input validation

State-changing POST endpoints employ the `[ValidateAntiForgeryToken]` attribute to mitigate CSRF attacks. For instance, client registrations are implemented as shown in listing 6.2.

```
1 [HttpPost, ValidateAntiForgeryToken]
2 public async Task<IActionResult> ClientRegistration(
3     ApplicationViewModel model)
4 {
5     // ...
6 }
```

Listing 6.2 Endpoint AntiForgery notation example

This requirement that incoming forms include a valid anti-forgery token prevents malicious cross-site submissions.

6.2.3 Rate limiting and abuse prevention

Critical endpoints such as `/connect/authorize` and `/connect/token` are guarded by the ASP.NET Core rate-limiting middleware via the `[EnableRateLimiting]` attribute as shown in listing 6.3.

```
1 [EnableRateLimiting("TokenEndpointPolicy")]
2 [HttpPost("~/connect/token"), IgnoreAntiforgeryToken]
3 public async Task<IActionResult> Exchange()
4 {
5     // ...
6 }
```

Listing 6.3 Endpoint rate limiter example

Configured thresholds (e.g. five login attempts and ten token requests per minute per IP) protects endpoints to credential stuffing and volumetric DoS attacks while preserving legitimate traffic.

6.2.4 Explicit error handling

Validation failures and protocol errors are spotted through feedback from the model state or responses that are OIDC compliant [33]. A duplicate client identifier during registration triggers as shown in listing 6.4.

```
1 if (client != null)
2 {
3     ModelState.AddModelError(nameof(model.ClientId), "Cannot
      use this ClientId");
4     return View(model);
5 }
```

Listing 6.4 Duplicate ClientId check example

Similarly, authentication and authorization failures return standardized error codes (such as `invalid_grant`, `consent_required`, etc.), enabling clients to implement precise error recovery paths.

6.2.5 Secure credential generation

Client credentials are produced via server-side generator to ensure unpredictability and client-side eavesdropping as shown in listing 6.5.

```
1 private static string GenerateClientSecret()
2     => Guid.NewGuid().ToString("N");
3
4 private static string GenerateClientId(int length)
5 {
6     const string chars = "ABCDEFGHJKLMNOPQRSTUVWXYZ
7     abcdefghijklmnopqrstuvwxyz0123456789";
8     using var rng = RandomNumberGenerator.Create();
9     var data = new byte[length];
10    rng.GetBytes(data);
11    var result = new char[length];
12    for (int i = 0; i < length; i++)
13    {
14        result[i] = chars[data[i] % chars.Length];
15    }
16    return new string(result);
17 }
```

Listing 6.5 Server-side client secret generation example

This approach prevents enumeration and brute-force attacks on client identifiers and secrets.

Collectively, these measures satisfy the non-functional security requirements (Section 4) and provide a resilient endpoint architecture that upholds confidentiality, integrity, and availability under adversarial conditions.

6.3 Client validation

Client validation prevents unauthorized or misconfigured applications from participating in the authorization flows, thereby preserving the integrity of the service and compliance with the OpenID Connect standard [9].

6.3.1 Uniqueness and existence checks

During registration the service verifies that the supplied client identifier is not already in use. The OpenIdDict application manager is queried, and any duplicate causes the operation to abort with a model-state error as shown in listing 6.6.

```
1 var manager = scope.ServiceProvider.GetRequiredService<  
    IOpenIdDictApplicationManager>();  
2 var existing = await manager.FindByClientIdAsync(model.  
    ClientId);  
3 if (existing is not null)  
4 {  
5     ModelState.AddModelError(nameof(model.ClientId),  
6         "This ClientId is already registered.");  
7     return View(model);  
8 }
```

Listing 6.6 ClientId uniqueness check

This measure ensures that each client application remains uniquely identifiable and eliminates conflicts or token misrouting arising from identifier reuse.

6.3.2 Redirect URI validation

As part of registration, each redirect URI provided by the client is parsed and validated to confirm well formed and trusted scheme (HTTPS). The URIs are split on newline delimiters, converted to `System.Uri` instances as shown in listing 6.7.

```
1 var redirectUris = (model.RedirectUris ?? string.Empty)
2   .Split(new[] { "\r\n", "\n" }, StringSplitOptions.
3     RemoveEmptyEntries)
4   .Select(url => new Uri(url))
5   .ToHashSet();
6 appDescriptor.RedirectUris.AddRange(redirectUris);
```

Listing 6.7 Redirect URI parsing and validation

Strict validation of redirect endpoints mitigates open-redirect vulnerabilities and conforms to the OIDC requirement that only pre-registered URIs be accepted during authorization requests.

6.3.3 Integration with OpenIddict

Once identifier and URI checks succeed, the client is registered with OpenIddict via an `OpenIddictApplicationDescriptor` that encapsulates its grant types, redirect URIs, and other permissions. This tight coupling guarantees that only validated clients are recognized by the token issuance and authorization endpoints, enforcing end-to-end security and protocol compliance.

6.4 Client registration

Client registration establishes the identity and permissions of applications interacting with the authorization server. It comprises a user-driven workflow with strong server-side validation and seamless integration into OpenIddict.

6.4.1 Registration workflow

Authenticated users with a admin role can access `/Applications/ClientRegistration` endpoint, where the service generates a random client identifier and secret and pre-populates the registration form as shown in listing 6.8.

```
1 public IActionResult ClientRegistration()
2 {
3     var id      = GenerateClientId(8);
4     var secret  = GenerateClientSecret();
5     var model   = new ApplicationViewModel
6     {
7         ClientId      = id,
8         DisplayName   = id,
9     };
10    return View(model);
11 }
```

Listing 6.8 Initial client registration form pre-population

ClientId and ClientSecret generation is explained in subsection 6.2.5.

6.4.2 Server-side validation and persistence

Upon submission, the service verifies model validity, checks for duplicate identifiers, and validates redirect URIs before persisting as shown in listing 6.9.

```
1 var manager = scope.ServiceProvider
2     .GetRequiredService<IOpenIdDictApplicationManager>();
3
4 var existing = await manager.FindByClientIdAsync(model.
5     ClientId);
6 if (existing != null)
7 {
8     ModelState.AddModelError(nameof(model.ClientId),
9         "This ClientId is already in use.");
10    return View(model);
11 }
12 var redirectUris = (model.RedirectUris ?? string.Empty)
13     .Split(new[] { "\r\n", "\n" }, StringSplitOptions.
14         RemoveEmptyEntries)
15     .Select(url => new Uri(url))
16     .ToHashSet();
```

Listing 6.9 ClientId uniqueness and redirect URI validation

6.4.3 Client creation and configuration

After validation, a new OpenIdDictApplicationDescriptor is constructed, specifying grant types, scopes, consent type, and PKCE requirement as shown in listing 6.10.

```

1 var appDescriptor = new OpenIdDictApplicationDescriptor
2 {
3     ClientId      = model.ClientId,
4     ClientSecret  = model.ClientSecret,
5     ConsentType   = ConsentTypes.Explicit,
6     DisplayName   = model.DisplayName,
7     Permissions   =
8     {
9         Permissions.Endpoints.Authorization,
10        Permissions.Endpoints.EndSession,
11        Permissions.Endpoints.Token,
12        Permissions.GrantTypes.AuthorizationCode,
13        Permissions.GrantTypes.RefreshToken,
14        Permissions.ResponseTypes.Code,
15        Permissions.Scopes.Email,
16        Permissions.Scopes.Profile,
17        Permissions.Scopes.Roles
18    },
19    Requirements   =
20    {
21        Requirements.Features.ProofKeyForCodeExchange
22    }
23 };
24 appDescriptor.RedirectUris.AddRange(redirectUris);
25 appDescriptor.PostLogoutRedirectUris.AddRange(postLogoutUris);
26 ;
27 await manager.CreateAsync(appDescriptor);

```

Listing 6.10 OpenIdDict application descriptor creation

6.4.4 User feedback and completion

Upon successful registration, the user is redirected to the home page. Any errors, such as duplicate identifiers or malformed URIs, are communicated via model-state messages and re-rendering of the registration view, ensuring transparency and immediate corrective guidance.

6.5 PKCE Validation

Proof Key for Code Exchange (PKCE) enhances the OAuth 2.0 authorization code grant by requiring a dynamic, per-request secret that prevents intercepted codes from being reused. In this solution, PKCE is mandated at client registration and enforced automatically by the OpenIdDict middleware.

6.5.1 Enforcing PKCE at client registration

New clients are configured to require PKCE by adding the `ProofKeyForCodeExchange` feature to their application descriptor. This declarative requirement instructs OpenIddict to reject any authorization request that omits or miscalculates the PKCE parameters as shown in listing 6.11.

```
1 var appDescriptor = new OpenIddictApplicationDescriptor
2 {
3     // ... other properties ...
4     Requirements =
5     {
6         Requirements.Features.ProofKeyForCodeExchange
7     }
8 };
```

Listing 6.11 Enforcing PKCE in client descriptor

6.5.2 PKCE validation at runtime

At authorization time, OpenIddict's server component validates the `code_challenge` supplied in the initial `/connect/authorize` request and the corresponding `code_verifier` sent to `/connect/token`. Non-conforming or missing values result in an `invalid_request` or `invalid_grant` error, ensuring that only clients possessing the original verifier can redeem an authorization code. This mechanism effectively refuses interception and replay attacks without additional application logic.

6.5.3 Password management

Secure password storage and verification are achieved through the use of the robust Argon2id memory-hard hashing algorithm, ensuring the highest level of protection for user credentials. Argon2id, as defined in RFC9106 [32], is employed for both password storage and verification due to its advanced memory-hard properties that significantly enhance resistance to brute-force attacks and ensure robust protection of user credentials.

Password storage ensures that when a user registers, the system securely stores their password by first generating a random salt using a secure random number generator (typically 16 bytes in size). This salt is then combined with the user's password and processed with the Argon2id hashing algorithm, configured with specific parameters (such as degree of parallelism, iterations, and memory size). The resulting hash (e.g., 16 bytes) is then converted to a Base64 string, and both the salt and hash are concatenated (often separated by a delimiter like a dot) to form the final password hash, which is subsequently stored in the database as described in listing 6.12.

```

1 byte[] salt = RandomNumberGenerator.GetBytes(16);
2
3 var argon2 = new Argon2id(Encoding.UTF8.GetBytes(password))
4 {
5     Salt = salt,
6     DegreeOfParallelism = 8,
7     Iterations = 4,
8     MemorySize = 1024 * 64
9 };
10
11
12 byte[] hashBytes = await argon2.GetBytesAsync(16); // 16
13     bytes hash
14 string saltBase64 = Convert.ToBase64String(salt);
15 string hashBase64 = Convert.ToBase64String(hashBytes);
16 string passwordHash = $"{saltBase64}.{hashBase64}";
17
18 var newUser = new User
19 {
20     UserName = username,
21     Email = username,
22     FirstName = firstname,
23     LastName = lastname,
24     PasswordHash = passwordHash
25 };
26 db.Users.Add(newUser);
27 await db.SaveChangesAsync();

```

Listing 6.12 Password storage code

This approach ensures that even if the database is compromised, the actual passwords remain protected due to the unique salt and the computational difficulty of reversing the hash.

Password verification ensures that during login with user credentials, the system validates the user's credentials by splitting the stored password hash to extract the original salt and hash values. The extracted salt is then used to recompute the hash of the provided password using the same Argon2id algorithm and parameters as during the storage process. The newly computed hash is compared with the stored hash, and if they match, the password is verified

successfully. Otherwise, the authentication is rejected as shown in listing 6.13.

```
1 var parts = user.PasswordHash.Split('.');
2 if (parts.Length != 2)
3     return Results.Unauthorized();
4
5 var salt = Convert.FromBase64String(parts[0]);
6 var storedHash = parts[1];
7
8 var argon2 = new Argon2id(Encoding.UTF8.GetBytes(password))
9 {
10     Salt = salt,
11     DegreeOfParallelism = 8,
12     Iterations = 4,
13     MemorySize = 1024 * 64
14 };
15
16 byte[] computedHashBytes = await argon2.GetBytesAsync(16);
17 string computedHash = Convert.ToBase64String(
18     computedHashBytes);
19 if (!computedHash.Equals(storedHash))
20     return Results.Unauthorized();
```

Listing 6.13 Password verification code

This method leverages Argon2's resistance to modern cracking techniques and ensures that only users providing the correct password can be authenticated [32].

6.6 Rate limiter

Rate limiting is employed to protect critical endpoints from brute-force attacks, automated abuse, and denial-of-service conditions by constraining the frequency of incoming requests per client.

6.6.1 Implementation and configuration

The solution utilizes the ASP.NET Core rate-limiting middleware, defining distinct policies for token issuance and login endpoints in the service configuration (5 requests / minute / IP address for login endpoints, 10 requests / min / IP address for token endpoints) described in Section 4.3. Rate limiter is configured as a service as shown in listing 6.14.

```

1 services.AddRateLimiter(options =>
2 {
3     options.AddPolicy("TokenEndpointPolicy", context =>
4         RateLimitPartition.GetFixedWindowLimiter(
5             context.Connection.RemoteIpAddress ?? IPAddress.
6                 Loopback,
7             => new FixedWindowRateLimiterOptions
8             {
9                 PermitLimit = 10,
10                Window = TimeSpan.FromMinutes(1),
11                QueueProcessingOrder = QueueProcessingOrder.
12                    OldestFirst,
13                QueueLimit = 2
14            });
15    options.AddPolicy("LoginEndpointPolicy", context =>
16        RateLimitPartition.GetFixedWindowLimiter(
17            context.Connection.RemoteIpAddress ?? IPAddress.
18                Loopback,
19            => new FixedWindowRateLimiterOptions
20            {
21                PermitLimit = 5,
22                Window = TimeSpan.FromMinutes(1),
23                QueueProcessingOrder = QueueProcessingOrder.
24                    OldestFirst,
25                QueueLimit = 1
26            });
27 });

```

Listing 6.14 Fixed-window rate-limiting policy definitions

Each policy is applied to the corresponding controller action, which corresponds to each endpoint, via the `EnableRateLimiting` attribute as shown in listing 6.15.

```

1 [EnableRateLimiting("TokenEndpointPolicy")]
2 [HttpPost("~/connect/token")]
3 public async Task<IActionResult> Exchange()
4 {
5     // Token exchange logic...
6 }

```

Listing 6.15 Applying rate-limiting policy to token endpoint

6.6.2 Security and resilience

By limiting requests to predefined thresholds and incorporating a small queue for burst tolerance, the rate limiter enforces fair usage and reduces the viability of rapid automated attacks. More stringent limits on the login policy mitigate credential stuffing, whereas the token policy preserves system responsiveness under peak load. This layered control strengthens the availability and integrity of the authentication service without imposing significant overhead on legitimate clients [41].

6.6.3 Token expiration

Token expiration policies constitute a vital element of the service's security architecture. Expiration policies limit the usefulness of compromised credentials while preserving an acceptable user experience. The configuration in listing 6.16 below satisfies the acceptance criteria defined for Authenticate and authorize end-users.

```
1 options.SetAuthorizationCodeLifetime(TimeSpan.FromMinutes(5))
2 options.SetAccessTokenLifetime(TimeSpan.FromMinutes(15))
3 options.SetIdentityTokenLifetime(TimeSpan.FromHours(1))
4 options.SetRefreshTokenLifetime(TimeSpan.FromDays(30));
```

Listing 6.16 OpenIddict token lifetime setting

- **Authorization codes** (*auth codes*) expire after 5 minutes, preventing intercepted codes from being redeemed beyond a brief window.
- **Access tokens** are valid for 15 minutes, so any stolen token is short-lived and cannot be replayed for long.
- **ID tokens** carry a 60-minute lifetime; because they are consumed only by the client for session management and never presented to APIs, a longer validity avoids needless re-authentication prompts without materially increasing risk.
- **Refresh tokens** remain valid for 30 days, providing seamless access across sessions while still giving administrators time to detect and revoke a compromised token.

These differentiated intervals ensure that short-lived artifacts (auth codes, access tokens) limit exposure to replay attacks, whereas longer-lived artifacts (ID and refresh tokens) maintain usability by reducing repeated logins [4].

6.7 Token revocation

The ability to revoke refresh tokens is a security control that enables immediate invalidation of credentials when compromise is suspected or when a user's privileges change. Without a

revocation mechanism, a stolen or leaked refresh token could be used to obtain new access tokens indefinitely, undermining the security of the entire system. RFC 7009 [42] specifies a standardized token revocation endpoint, allowing clients to inform the authorization server that a given token, especially long-lived refresh tokens, should no longer be accepted. By implementing this revocation endpoint, the service ensures that refresh tokens can be proactively invalidated upon user logout, credential rotation, or detection of anomalous activity, thereby minimizing the risk of unauthorized access and maintaining compliance with best security practices [42].

CONCLUSION

This thesis has presented the research, design and realization of an Authentication-as-a-Service solution built on the .NET platform, with the goal of enabling small and medium-sized projects to integrate secure authentication and authorization with minimal effort. The work began with a survey of existing commercial and open-source offerings (Auth0, Azure AD B2C, Firebase Authentication, Okta, Keycloak, FusionAuth), identifying gaps in .NET support, deployment flexibility and cost that motivated a custom, lightweight service. Guided by industry standards: OAuth 2.0, OpenID Connect and JSON Web Tokens, and drawing on best current practices (PKCE, Argon2id, rate limiting, token revocation), the thesis defined a coherent set of user stories and requirements that drive both functional scope and non-functional quality attributes.

In Chapter 3, “Solution Goals and User Stories,” nine developer-centric scenarios were defined, ranging from automated credential-based and federated login to client application onboarding, consent management, token lifecycle operations and role administration. Each user story was linked to concrete acceptance criteria, ensuring that the service design would yield measurable outcomes. These functional requirements were complemented in Chapter 4 by non-functional demands, security, performance, resilience and deployability, with explicit thresholds for token lifetimes, hashing latency, request throughput and container image size.

The heart of the implementation, described in Chapter 5, relies on ASP.NET Core Identity and the OpenIddict library to stand up a standards compliant authorization server. The code exhibits a clear separation of concerns: Razor Pages for the administrative UI, controller actions for OIDC endpoints (`/connect/authorize`, `/connect/token`, `/connect/logout`), and middleware policies for anti-forgery, rate limiting and TLS enforcement. Access tokens (15 min), ID tokens (60 min) and refresh tokens (30 days) follow a differentiated expiration strategy that balances security with usability. Proof Key for Code Exchange (PKCE) and a unique JWT ID (`jti`) defend against interception and replay, while Argon2id hashing parameters (64 MiB, 4 iterations) resist brute-force attacks without excessive login latency.

Chapter 6 evaluated the service’s security and resilience through STRIDE threat modeling,

demonstrating how declarative attributes, explicit input validation and fixed-window rate limiting (5 requests per minute for login, 10 requests per min for token exchange) mitigate common attack vectors. The design also integrates token revocation and back-channel logout to ensure that compromised credentials or global sign-out requests are propagated within seconds.

Throughout this work, particular attention was paid to ease of adoption: the AaaS solution is packaged as a Docker image, deployable via Docker Compose. It exposes a minimal set of REST and OIDC endpoints discoverable via the `/.well-known/openid-configuration` document and it includes a self-service administration console for client registration, user management and token introspection. By adhering to fully documented industry standards and offering an open-source reference implementation, the service empowers developers to onboard authentication and authorization capabilities without in-depth security expertise.

By targeting the unique needs of smaller projects, rapid integration, minimal operational overhead and predictable costs, this thesis delivers a practical blueprint for Authentication-as-a-Service on the .NET platform. Future work may explore multi-tenant isolation, adaptive rate limiting, richer consent management (e.g. GDPR-driven scope summaries) and integration with emerging decentralized identity schemes, further extending the service's applicability and trustworthiness in an evolving security landscape.

REFERENCES

- [1] WILSON, Yvonne, 2019. *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0*. Abhishek Hingnikar. ISBN 978-1-4842-5095-2. Available from: <https://doi.org/10.1007/978-1-4842-5095-2>.
- [2] MICROSOFT, 2024. *Overview of ASP.NET Core Authentication*. Online. Microsoft Learn. Available from: <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-9.0#authentication-scheme>. [cit. 2025-04-13].
- [3] OKTA, 2024. *Authentication vs. Authorization*. Online. Okta. Available from: <https://www.okta.com/identity-101/authentication-vs-authorization/>. [cit. 2025-04-30].
- [4] HARDT, D., October 2012. *The OAuth 2.0 Authorization Framework*. RFC 6749. Available from: <https://tools.ietf.org/html/rfc6749>.
- [5] OKTA, 2024. *RBAC vs. ABAC: Definitions & When to Use*. Online. Okta. Available from: <https://www.okta.com/identity-101/role-based-access-control-vs-attribute-based-access-control/>. [cit. 2025-04-15].
- [6] RICHER, Justin and SANSONO, Antonio, 2017. *OAuth 2 in Action*. Manning Publications. ISBN 978-1-61729-327-6. Available from: <https://www.manning.com/books/oauth-2-in-action>.
- [7] OAUTH, 2025. *OAuth 2.0*. Online. Available from: <https://oauth.net/2/>. [cit. 2025-03-01].
- [8] PARECKI, Aaron, 2019. *It's Time for OAuth 2.1*. Online. Aaron Parecki Blog. Available from: <https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1>. [cit. 2025-04-27].
- [9] CONNECT, OpenID, 2025. *What is OpenID Connect*. Online. Available from: <https://openid.net/developers/how-connect-works/>. [cit. 2025-03-01].
- [10] PEYROTT, Sebastián E., 2024. *The JWT Handbook*. Online. Auth0. Available from: <https://auth0.com/resources/ebooks/jwt-handbook>. [cit. 2025-03-29].

- [11] JWT, 2025. *Introduction to JSON Web Tokens*. Online. Available from: <https://jwt.io/introduction>. [cit. 2025-03-02].
- [12] MICROSOFT, 2025. *What is SAML?* Online. Available from: <https://www.microsoft.com/en-us/security/business/security-101/what-is-security-assertion-markup-language-saml>. [cit. 2025-03-01].
- [13] AUTH0, 2025. *Secure Access for Everyone. But Not Just Anyone*. Online. Available from: <https://auth0.com/>. [cit. 2025-02-25].
- [14] MICROSOFT, 2025. *What is Azure Active Directory B2C?* Online. Available from: <https://learn.microsoft.com/en-us/azure/active-directory-b2c/overview>. [cit. 2025-03-01].
- [15] FIREBASE, 2025. *Firestore Authentication*. Online. Available from: <https://firebase.google.com/docs/auth>. [cit. 2025-03-01].
- [16] OKTA, 2025. *Okta and Auth0 Platforms*. Online. Available from: <https://www.okta.com/>. [cit. 2025-02-25].
- [17] KEYCLOAK, 2025. *Open Source Identity and Access Management*. Online. Available from: <https://www.keycloak.org/>. [cit. 2025-03-01].
- [18] FUSIONAUTH, 2025. *Getting Started with FusionAuth: A Beginner's Guide*. Online. Available from: <https://fusionauth.io/docs/get-started/start-here>. [cit. 2025-03-01].
- [19] HOFFMAN, Kevin, 2017. *Building Microservices with ASP.NET Core: Develop, Test, and Deploy Cross-Platform Services in the Cloud*. O'Reilly Media. ISBN 978-1-491-96173-5.
- [20] FREEMAN, Adam, 2020. *Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages*. Apress. 8th edition. ISBN 978-1-4842-5440-0. Available from: <http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&AN=2494147&authtype=ip,shib&custid=s3936755>.

- [21] PRICE, Mark J., 2021. *C# 10 and .NET 6 - Modern Cross-Platform Development: Build Apps, Websites, and Services with ASP.NET Core 6, Blazor, and EF Core 6 Using Visual Studio 2022 and Visual Studio Code*. Packt. 6th edition. ISBN 978-1-80107-736-1.
- [22] MICROSOFT, 2024. *What's New in .NET 9*. Online. Microsoft Learn. Available from: <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-9/overview>. [cit. 2025-05-20].
- [23] MICROSOFT, 2022. *What Is An Identity Provider, And Why Do Companies Need One?* Online. Microsoft. Available from: <https://www.crypitr.co/blog/what-is-an-idp>. [cit. 2025-05-20].
- [24] PROJECT, OpenIddict, 2025. *OpenIddict Introduction*. Online. Available from: <https://documentation.openiddict.com/introduction>. [cit. 2025-05-24].
- [25] POSTGRESQL, 2025. *PostgreSQL 17.4 Documentation*. Online. PostgreSQL.org. Available from: <https://www.postgresql.org/files/documentation/pdf/17/postgresql-17-A4.pdf>. [cit. 2025-03-30].
- [26] CRUDU, Valeriu, 2024. *What Are the Most Popular Frameworks Used by .NET Developers?* Online. MoldStud. Available from: <https://moldstud.com/articles/p-what-are-the-most-popular-frameworks-used-by-dotnet-developers>. [cit. 2025-05-20].
- [27] CRUDU, Valeriu, 2025. *Overview of YARP*. Online. Microsoft Learn. Available from: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/yarp-overview?view=aspnetcore-9.0&source=recommendations>. [cit. 2025-05-20].
- [28] AUTHGEAR, 2025. *Authentication-as-a-Service: What Is It and Why You Need It*. Online. Authgear. Available from: <https://www.authgear.com/post/authentication-as-a-service>. [cit. 2025-05-22].
- [29] GORBACHENKO, Pavel, 2025. *What are Functional and Non-Functional Requirements and How to Document These*. Online. Enkonix. Available from: <https://enkonix.com/blog/functional-requirements-vs-non-functional/>. [cit. 2025-05-24].

- [30] LODDERSTEDT, Torsten; BRADLEY, John; LABUNETS, Andrey and FETT, Daniel, January 2025. Best Current Practice for OAuth 2.0 Security. RFC 9700. Available from: <https://www.rfc-editor.org/info/rfc9700>.
- [31] PRAJAPATI, Sudhanshu, July 2023. *Why Adaptive Rate Limiting is a Game-Changer*. Online. Available from: <https://medium.com/fluxninjaq/why-adaptive-rate-limiting-is-a-game-changer-79f130e6ec71>. [cit. 2025-05-24].
- [32] BIRYUKOV, Alex; DINU, Daniel; KHOVRATOVICH, Dmitry and JOSEFSSON, Simon, September 2021. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106. Available from: <https://www.rfc-editor.org/info/rfc9106>.
- [33] SAKIMURA, Nat; BRADLEY, John; JONES, Michael; MEDEIROS, Breno and MORTIMORE, Chuck, December 2023. OpenID Connect Core 1.0 incorporating errata set 2. Available from: https://openid.net/specs/openid-connect-core-1_0.html.
- [34] NIELSEN, Jakob, June 2010. *Website Response Times*. Online. Available from: <https://www.nngroup.com/articles/website-response-times/>. [cit. 2025-05-24].
- [35] GITHUB, 2025. *Rate limits for the REST API*. Online. Available from: <https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api>. [cit. 2025-05-24].
- [36] MORIARTY, Kathleen; KALISKI, Burt; JONSSON, Jakob and RUSCH, Andreas, November 2016. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017. Available from: <https://www.rfc-editor.org/info/rfc8017>.
- [37] ANDERSON, Ross, 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley. 3rd edition. ISBN 978-1-119-64278-7.
- [38] FOUNDATION, OWASP, 2025. *Cross Site Scripting Prevention Cheat Sheet*. Online. Available from: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. [cit. 2025-05-24].

- [39] DEJONGHE, Derek and NUGARA, Arlan, 2021. *Application Delivery and Load Balancing in Microsoft Azure: Practical Solutions with NGINX and Microsoft Azure*. O'Reilly. ISBN 978-1-098-11586-9.
- [40] CONKLIN, Larry, 2025. *Threat Modeling Process*. Online. OWASP. Available from: https://owasp.org/www-community/Threat_Modeling_Process. [cit. 2025-04-26].
- [41] MICROSOFT, 2025. *Rate Limiting Middleware in ASP.NET Core*. Online. Microsoft Learn. Available from: <https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit?view=aspnetcore-8.0>. [cit. 2025-04-14].
- [42] LODDERSTEDT, Torsten; DRONIA, Stefanie and SCURTESCU, Marius, August 2013. OAuth 2.0 Token Revocation. RFC 7009. Available from: <https://www.rfc-editor.org/info/rfc7009>.

LIST OF FIGURES

Figure 1.1	Authentication diagram	16
Figure 1.2	Authorization diagram.....	16
Figure 1.3	JWT structure.....	18
Figure 2.1	OAuth abstract protocol flow.....	26
Figure 2.2	OAuth access token with refresh token	27
Figure 5.1	Solution architecture	45
Figure 5.2	Clients list UI	58
Figure 5.3	Register new client form	59
Figure 5.4	List of active authorizations	59
Figure 5.5	List of active tokens	60
Figure 5.6	List of active users.....	60
Figure 5.7	List of active authorizations	61
Figure 5.8	Docker compose network diagram (source: own).....	61

LIST OF TABLES

Tabulce 1.1	Comparison of AaaS solutions (source: own).....	22
Tabulce 3.1	Overview of the defined user stories	34
Tabulce 4.1	User story coverage across thesis chapters.....	41
Tabulce 5.1	Service endpoints, methods, and their functions.....	50

LIST OF LISTINGS

Listing 5.1	Authentication example [9]	48
Listing 5.2	Consent handling in the authorization controller [24]	49
Listing 5.3	Rendering the consent page when explicit approval is needed	52
Listing 5.4	Validation logic for authorisation-code and refresh-token grants	54
Listing 5.5	Claim population prior to token issuance	55
Listing 5.6	Set scope example	55
Listing 5.7	Core logic for refresh-token grant	56
Listing 5.8	External OIDC integration example	57
Listing 6.1	Endpoint authorize notation example	64
Listing 6.2	Endpoint AntiForgery notation example	65
Listing 6.3	Endpoint rate limiter example	65
Listing 6.4	Duplicate ClitentId check example	66
Listing 6.5	Server-side client secret generation example	66
Listing 6.6	ClientId uniqueness check	67
Listing 6.7	Redirect URI parsing and validation	67
Listing 6.8	Initial client registration form pre-population	69
Listing 6.9	ClientId uniqueness and redirect URI validation	69
Listing 6.10	OpenIddict application descriptor creation	70
Listing 6.11	Enforcing PKCE in client descriptor	71
Listing 6.12	Password storage code	72
Listing 6.13	Password verification code	73
Listing 6.14	Fixed-window rate-limiting policy definitions	74
Listing 6.15	Applying rate-limiting policy to token endpoint	74
Listing 6.16	OpenIddict token lifetime setting	75

LIST OF ABBREVIATIONS

SSO	Single-Sign On
XML	Extensible Markup Language
SaaS	Software as a Service
AaaS	Authentication as a Service
OIDC	OpenID Connect
JWT	JSON Web Token
SAML	Security Assertion Markup Language
LDAP	Lightweight Directory Access Protocol
AD	Active Directory
API	Application Programming Interface
MVC	Model–View–Controller
YARP	Yet Another Reverse Proxy
SDK	Software Development Kit
B2C	Business-2-Customer
REST	REpresentational State Transfer
RSA	Rivest–Shamir–Adleman algorithm
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
CSRF	Cross-Site Request Forgery
PKCE	Proof Key for Code Exchange
ORM	Object–Relational Mapping
LINQ	Language Integrated Query
CI/CD	Continuous Integration / Continuous Deployment
JSON	JavaScript Object Notation
GPU	Graphics Processing Unit
TLS	Transport Layer Security
RFC	Request For Comments
GUID	Globally Unique Identifier
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
XSS	Cross Site Scripting

LIST OF APPENDICES

APPENDIX A I: Solution source code

APPENDIX A I: SOLUTION SOURCE CODE

The appendix contains source code for this thesis and source code, configuration and deployment files for AaaSSolution. The directory structure is as follows:

- `fulltext.pdf` - diploma thesis text in PDF/A format
- `src` - main directory
- `src/Thesis/` - diploma thesis source code
- `src/Source/AaaSSolution/` - AaaSSolution source code directory
- `src/Source/AaaSSolution/readme.md` - AaaSSolution readme file
- `src/Source/AaaSSolution/AaaSServer/` - auth server source code
- `src/Source/AaaSSolution/WebClient1/` - example web client source code
- `src/Source/AaaSSolution/WebClient2/` - example web client source code
- `src/Source/AaaSSolution/YARP_LoadBalancer/` - load balancer source code