University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master's thesis

Security of Distributed Cloud Systems

Plzeň 2024

Bc. Petr Urban

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Petr URBAN**
Osobní číslo:	**A22N0061P**
Studijní program:	**N0613A140040 Softwarové a informační systémy**
Téma práce:	**Zabezpečení distribuovaných cloudových systémů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s problematikou microservices, webových služeb a možnostmi komunikace s nimi.
2. Analyzuje v kontextu těchto architektur možnosti bezpečné komunikace, zajištění důvěry (trust management), správy identit a přístupu (identity and access management), autentizace a autorizace. Navrhněte vhodné komplexní řešení těchto aspektů pro alespoň jednu vybranou konkrétní aplikaci.
3. Implementujte navržené řešení a demonstrujte několik ostatních formou jednoduchých Proof-of-Concept prototypů.
4. Řešení řádně otestujte a kriticky zhodnoťte přínosy, nevýhody a důsledky návrhu.

Rozsah diplomové práce:	**doporuč. 50 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Ing. Jakub Daněk** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**8. září 2023**
Termín odevzdání diplomové práce:	**16. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2023

Declaration

Abstract

The goal of this thesis was to explore microservices and web services, focusing on their communication capabilities. Furthermore, an analysis of safe communication, trust, identity, and access management, along with authentication and authorization within these architectures was made in order to choose the best model for user access and management for the SPADe project, which is a research tool invented at the Department of Computer Science and Engineering. Key issues, particularly authentication and authorization protocols like OIDC and SAML, were applied to a demo application. Based on the analysis and implementation results, the OIDC protocol was chosen using the IAM tool Keycloak.

Keywords: owasp, nist, cybersecurity, oauth2, oidc, saml, microservices, soa communication

Abstrakt

Cílem této práce bylo zkoumat mikroslužby a webové služby s důrazem na jejich komunikační schopnosti. Byla provedena analýza bezpečné komunikace a správy přístupu, včetně autentizace a autorizace, pro výběr optimálního modelu pro projekt SPADe z katedry informatiky a výpočetní techniky. Hlavní zaměření bylo na protokoly autentizace a autorizace jako OIDC a SAML, které byly testovány na demonstrační aplikaci. Na základě analýzy a implementace byl pro projekt vybrán protokol OIDC pomocí nástroje IAM Keycloak.

Acknowledgement

I would like to thank my thesis supervisor, Ing. Jakub Daněk, from the Faculty of Applied Sciences of the University of West Bohemia. His office door was always open whenever I encountered obstacles or had questions about my research or writing. He consistently guided me in the right direction whenever necessary. This thesis has confirmed my interest in the field of cybersecurity, a domain I want to explore further in the future.

Contents

1 Introduction **10**

2 Microservices (SOA) **11**

2.1 Definition and Characteristics of Microservices . . . 12

2.2 Advantages and disadvantages . . . 12

2.2.1 Advantages . . . 12

2.2.2 Disadvantages . . . 13

2.3 Commonly used approaches in communication . . . 14

3 Communication Between Microservices **16**

3.1 Rest API . . . 16

3.1.1 Advantages and Disadvantages of REST API . . . 17

3.2 Message Queue . . . 17

3.2.1 Definition and Characteristics of Message Queue . . . 17

3.2.2 Message Queue Protocols . . . 18

4 Security Approaches and Risks **19**

4.1 Application Security Verification Standard (ASVS) . . . 19

4.1.1 ASVS Overview . . . 19

4.1.2 ASVS Related Risks . . . 20

4.2 Data Protection Techniques: Encryption and Hashing . . . 22

4.2.1 Essentials of Data Security . . . 22

4.2.2 Hashing . . . 22

4.2.3 Attacks to Hashes . . . 23

4.2.4 Encryption . . . 24

4.3 Transport Layer Security (TLS) / Secure Sockets Layer (SSL) 27

4.3.1 Definition of TLS / SSL . . . 27

4.3.2 Importance in Cybersecurity . . . 28

4.4 Identity and Access Management (IAM) . . . 28

4.4.1 Authentication . . . 28

4.4.2 Process of Authentication . . . 29

4.4.3 User Identification and Management . . . 32

4.4.4 Username & Password Authentication . . . 32

4.4.5 IP Address Authentication . . . 35

4.4.6 X.509 Certificates . . . 36

4.4.7 WebAuthn 38
4.4.8 Risk-Based Authentication (RBA) Method 40
4.4.9 Authorization Overview 41
4.4.10 Process of Authorization 41
4.4.11 Recommendations for Effective Authorization 42
4.4.12 Identity and Access Management Technologies 42
4.5 Authentication and Authorization Protocols 44
4.5.1 JSON Web Tokens (JWT) 44
4.5.2 Open Authorization (OAuth) 2.0 46
4.5.3 OpenID Connect (OIDC) 49
4.5.4 Security Assertion Makrup Language (SAML) 2.0 . . 51
4.5.5 Kerberos 55
4.5.6 LDAP in Cloud Computing Environments 57
4.6 Trust Management 58
4.6.1 Trust Management Technologies 59
4.7 Further Security Practices 59
4.7.1 Input Validation 60
4.7.2 Common Attacks on the Internet 60
4.7.3 Common Security Risks and Attacks 61

5 Practical Implementation **63**
5.1 Introduction of the Applications 63
5.2 Design and Architecture 65
5.3 Brief Description of Components 65
5.3.1 Docker 65
5.3.2 Keycloak 66
5.3.3 Overview of the Core Application 66
5.3.4 Simple Broker 68
5.3.5 ReactJS Application for User Interface 69
5.3.6 Databases 71
5.3.7 Nginx 71
5.3.8 Apache Web Server 72
5.4 Chosen Authentication Protocol For
SPADe Project 73
5.4.1 Authentiction and Authorization 73
5.4.2 User Management in the SPADe 76
5.5 Illustration of Implemented Security Approaches 77
5.5.1 Authentication 77
5.5.2 Authorization 82
5.5.3 Message Signing 82

5.5.4 Encryption and Hashing 83
5.5.5 User Management and Trust Management 84

6 Testing and Quality Assurance of The Demo Application 86
6.1 ZAP Proxy 87
6.1.1 Vulnerabilities Identified and Addressed by ZAP Proxy Tool 87
6.1.2 Corrections For Vulnerabilities 88

7 Conclusion 92

Used Shortcuts 96

Bibliography 97

A Description of the directory structure of the submitted file 104

B Images of the graphical user interface of the demo application 106

1 Introduction

In the current technological landscape, the shift towards decomposing services into smaller, independently operating microservices has garnered significant attention from developers of large-scale systems. This trend, along with the advent of cloud computing, offers numerous advantages. For instance, it eliminates the need for powerful hardware by hosting systems in the cloud, allows for smaller, more manageable codebases through the use of thin clients, and supports language independence, enabling services to be developed in various programming languages and hosted in different locations. However, this architectural style necessitates intricate communication between services, each with its distinct responsibilities, thereby introducing complexity not only in development but also in maintaining security. As automation and remote service operation become more prevalent, particularly in cloud-based environments, the demand for stringent security measures escalates in response to the increasing frequency of cyberattacks.

This thesis provides an initial exploration of microservices, examining the array of protocols that enable these services to communicate effectively while highlighting crucial security aspects. It leverages respected sources in cybersecurity, like the Open Web Application Security Project (OWASP) and the National Institute of Standards and Technology (NIST), to underline the significance of secure practices, including but not limited to authentication, authorization, various security protocols, Identity and Access Management (IAM), and Trust Management (TM). By applying some of these concepts from the analysis to a demonstration application, the study presents a practical application of selected security measures, notably focusing on authentication protocols. The ultimate goal is to select and integrate superior security mechanisms, especially authentication protocol into the SPADe application, aiming to significantly improve its security, especially in terms of authentication and management of user identities to determine their privileges not only to allow an access to the application's resources, but also to trigger the application's processes, thereby providing a secure model for similar applications. The SPADe (Software Process Anti-patterns Detector) is used to analyze projects to detect activities that may have a negative impact on the overall project.

2 Microservices (SOA)

Service Oriented Architecture (SOA) is a standard way of developing robust systems and smaller services that communicate with each other in the end. There is an assumption that the usage of the SOA approach eliminates monolithic applications, as the core of each service is separated and independent. This may result in better maintainability and scalability of the whole system, which not only leads to potentially better testing methods that can be used to improve the quality of service but also makes them easier to replace in some cases. Not surprisingly, these components often communicate with each other, either via a defined protocol or via an Application Programming Interface (API). This service-oriented architecture also brings another benefit associated with cloud computing, as each service can be hosted in a different environment under different circumstances, which raises a very important question - how to secure these systems, as each service may run in a different environment, may be implemented in different programming languages, and each service may require different security policies to access specific resources and do specific tasks. Cybersecurity plays a pivotal role with terms like authentication, authorisation, data consistency, encryption and more, which is the main topic of this thesis.

This chapter discusses the fundamentals of microservices, including their advantages and disadvantages, as well as various approaches to communication between them. Additionally, it discusses the technologies commonly used to develop these services and the final environment in which they can be run.

2.1 Definition and Characteristics of Microservices

Microservices[1] are small applications with only one responsibility independent of the rest of the system. A simple example may by illustrated by this diagram:

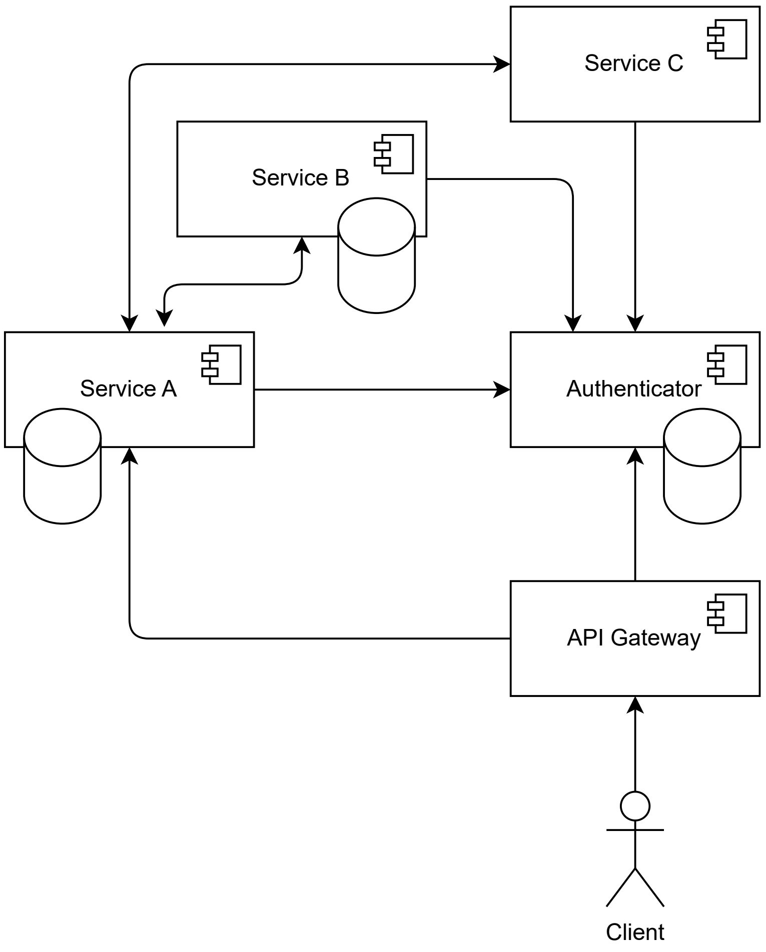


Figure 2.1: Illustration of SOA

2.2 Advantages and disadvantages

2.2.1 Advantages

[2],[3] Before delving into the specifics of microservices, it is crucial to understand the myriad benefits they bring to the table. This section illuminates the advantages of adopting a microservices architecture, from enhancing

scalability and technological flexibility to bolstering system resilience and deployment efficiency. Each point is articulated to provide readers with insights into why microservices have become the preferred architectural pattern for many contemporary software development projects.

Here is a brief list of the most important advantages from the point of Microservices:

- **Scalability**: Microservices enable scaling of individual components as needed, without the necessity to scale the entire application, making it more efficient and cost-effective.
- **Flexibility in Technology**: Teams can select the most suitable technology stack for each service, fostering innovation and efficiency.
- **Resilience**: Failure in one service does not necessarily compromise the entire system, enhancing overall reliability.
- **Deployment Velocity**: Independent services allow for quicker updates and deployments, facilitating continuous delivery and integration.

2.2.2 Disadvantages

Despite their benefits, microservices also present certain challenges and drawbacks that must be navigated carefully. This section aims to highlight the complexities and potential pitfalls of implementing a microservices architecture. These include management difficulties, data consistency issues, communication overheads and increased infrastructure requirements.

Here is a brief list of some significant disadvantages from the point of Microservices:

- **Complexity in Management**: The distributed nature of microservices introduces complexity in deployment, monitoring, and management.
- **Data Consistency**: Ensuring data consistency across services can be challenging due to decentralized data management.
- **Inter-service Communication**: The reliance on network calls between services can introduce latency and potential points of failure.
- **Overhead**: The architectural overhead can increase with the need for more infrastructure and tools to manage the microservices ecosystem.

2.3 Commonly used approaches in communication

[4] Today, there are two basic service-to-service communications that can be chosen when designing SOA architecture. Here is a list of the most common ones:

- **Synchronous**: As defined by Microsoft's own source, this type of communication follows a pattern where a service calls a API exposed by another service, using a protocol such as Hypertext Transfer Protocol (HTTP) or gRPC Remote Procedure Calls (gRPC). This approach is called synchronous because the caller waits for a response from the receiver.

- **Asynchronous**: In other words, asynchronous messaging. A type of communication in which (not only) a service sends a message without waiting for a response. This sent message is then processed asynchronously by one or more services. For example, Message Queue (MQ) is one of the candidates for the asynchronous type of communication.

It is typical today that many (not only) distributed applications run in cloud-based environments, making them accessible to users over the Internet. It is also typical that these applications, or in other words services, can communicate with other services. It does not matter what type of communication is used, at this point it is very important to take a look at the security aspects and make sure that the communication is secure enough to prevent cyber threats that may occur.

To achieve a secure communication not only between client and service, but also between service to service communication, it is important to use security approaches such as Transport Layer Security (TLS) which includes encryption of the transferred data to keep the data safe against unwanted leakage so that only the right participants communicating with each other are able to obtain the data. Not only encryption is required, but also some sort of mechanism to ensure that the data being transferred are not compromised. This is called integrity checking. Not only the communication itself must be protected, but also the resources that are accessible to others. This is where authentication and authorization play a key role.

All these approaches, from data encryption, data integrity, authentication, authorization and more, are discussed later in the theoretical part of this thesis, in the **chapter 4**. Now follows a **chapter 3** that briefly introduces selected communication approaches, such as Representational State

Transfer (REST), gRPC, and MQ. Especially MQ and REST have been selected and implemented in the practical part of this thesis described in the **chapter 5**.

3 Communication Between Microservices

3.1 Rest API

[5] Roy Fielding introduced REST in his doctoral dissertation, defining it as Representational State Transfer. A RESTful application, as Surwase explains, adheres to six architectural constraints, namely:

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand

These principles facilitate the development of scalable, flexible, and efficient services. RESTful services expose endpoints, such as:

```
http://example.com/api/resource
```

offering Application Programming Interfaces (APIs) that provide a means for service-to-service and service-to-client communication mediated via HTTP, which means that the REST also shares the security parameters of HTTP that must be considered. Also, since this approach is stateless by definition, it means that the server itself does not store any session information about the client, but processes the operations that the client requires. The server simply accepts the client's requests and processes them. This raises an important question - how to authenticate the user and decide whether the requested operation is allowed or not? This is a problem that is discussed in more detail in the **section 4.5**.

The REST has also been used in the implementation in the demo application in the **chapter 5**, illustrating the protection of resources located on specific Uniform Resource Locator (URL) and also controlling the access of each user.

3.1.1 Advantages and Disadvantages of REST API

The REST architectural style offers several advantages, including scalability, flexibility, and a wide range of supported data formats. However, it also presents challenges such as potential over-fetching or under-fetching of data and the need for careful design to avoid these issues[6].

3.2 Message Queue

Since the MQ communication approach is demonstrated in the practical part of this thesis (**chapter 5**), this chapter discusses what the message queue is, its advantages, disadvantages and, most importantly, its security implications for a system that needs to be properly secured.

3.2.1 Definition and Characteristics of Message Queue

[7],[8] A message queue serves as a key mechanism for asynchronous inter-service communication in both serverless[1] and microservices architectures. By allowing messages to be held in a queue until they are ready to be processed and deleted, it ensures that each message is handled once by a single consumer. This setup facilitates the decoupling of processing tasks, enables the buffering or batching of work, and aids in managing fluctuating workloads.

[7] In practice, message queues allow applications to send and receive messages without requiring the producer and consumer to interact with the message at the same time. This flexibility is crucial for applications that are structured as a collection of smaller, independent services, enhancing their development, deployment, and maintenance efficiency. Key features often associated with message queue implementations include message durability options, security policies, and various delivery and routing policies, among others.

[1]"Serverless computing is a method of providing backend services on an as-used basis. A serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure" [9]

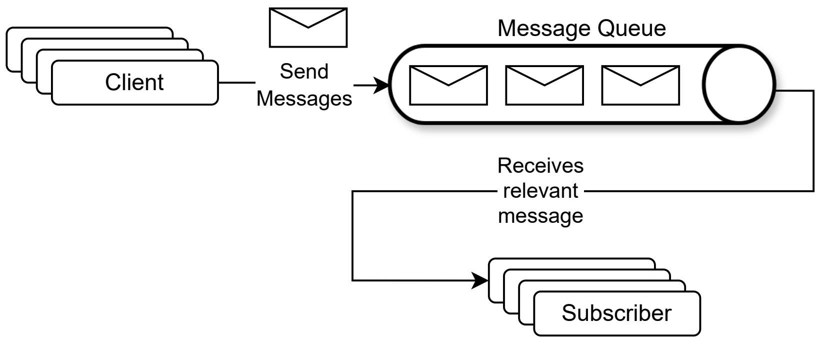


Figure 3.1: Illustration of MQ

As shown on the **image 3.1**, clients (a user or a system) push messages into the queue that offers a specific endpoint. On the other hand, subscribers (clients or other systems that listen to the queue, or so-called, they have subscribed to it and that want to receive the messages from the queue) receive the message as soon as there are any.

3.2.2 Message Queue Protocols

[10] Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and Advanced Message Queuing Protocol (AMQP) are typically used within the MQ protocol suite. STOMP is known for its simplicity and adaptability, making it suitable for a wide range of applications from web messaging to complex enterprise environments. MQTT, designed for low-bandwidth and high-latency situations such as Internet of Things (IoT) contexts, provides efficient message delivery under constrained conditions. AMQP is used because of its reliability, flexibility, security and open nature.

4 Security Approaches and Risks

This chapter is a comprehensive guide to contemporary security approaches and the risks prevalent in today's digital landscape, with a focus on practical application and prevention strategies. It meticulously outlines various security measures, each bolstered by credible sources like OWASP and others within the cybersecurity domain. These approaches are not only detailed theoretically but some of them are also illustrated through a demo application to show their practical implementation.

Central to the thesis, this chapter aims to consolidate essential security knowledge vital for developers of cloud-based applications or any system that interacts with users and potentially handles sensitive data. In an era where even minor applications are vulnerable to attacks and stringent regulations like General Data Protection Regulation (GDPR) demand rigorous data protection, understanding and applying these security measures becomes imperative.

Covering a broad spectrum of cybersecurity practices, this section delves into authentication, authorization, encryption, hashing, the use of IAM tools, and many more. It also highlights common attacks, offering insights into their prevention or mitigation. The ultimate goal is to safeguard the integrity of server-client communications, protect data from breaches, and ensure system availability. Selected security approaches are further explored in **chapter 5**, demonstrating their real-world applicability.

4.1 Application Security Verification Standard (ASVS)

4.1.1 ASVS Overview

Application Security Verification Standard (ASVS) [11] is a project designed to help developers assess the effectiveness of security measures in their web applications. It provides a guide and checklist for a complex security evaluation, from the process flow in the system to the technical aspects in general. By standardizing the verification process, ASVS aims to ensure consistent and thorough security assessments across different applications and environments.

One of the greatest advantages of the OWASP ASVS project is its accessibility. Hosted on the OWASP Foundation's GitHub repository, the standard is freely available to not only developers. This accessibility allows developers to review the documentation and adopt standardized security practices for assessing the security quality of specific applications in many aspects such as: development process flow, architectural styles (patterns) used, and more.

However, it is essential to acknowledge a fundamental reality: no big application can be completely protected against every cyber threat. Developers must recognize that achieving comprehensive security involves more than just adhering to standardized guidelines. Considerations such as application-specific use cases and compliance with industry or governmental regulations play crucial roles. For instance, applications handling sensitive personal data or those with potential real-world impacts demand heightened security measures to prevent data breaches or mitigate potential risks to individuals' lives.

This standard is highlighted in the thesis for its comprehensive enumeration of common vulnerabilities and attacks that need to be addressed to ensure the security of applications. It serves as an important resource, especially for developers, as it emphasises the importance of being familiar with these security risks in order to develop applications that are robust and secure against many potential threats. Developers, especially security professionals who are charged with ensuring the security of software, can use this standard as a checklist to ensure that OWASP-recommended security practices are incorporated into their software.

4.1.2 ASVS Related Risks

While the Application Security Verification Standard (ASVS) provides comprehensive guidelines for securing web applications, it's also essential to contextualize these within the broader landscape of prevalent security risks. The OWASP Top Ten API Security Risks for 2019 and 2023 outline the most critical web application security risks currently identified by security professionals globally. Acknowledging these risks underscores the dynamic and evolving nature of web application security, particularly in the domain of Rest APIs, which are becoming increasingly popular. This thesis cannot track updated problems due to the unpredictable and rapidly changing nature of vulnerabilities and risks; hence, more information should be sought from up-to-date and trustworthy sources like OWASP and NIST.

The list of the Top Ten API Security Risks for 2019 includes [12] :

1. API1:2019 - Broken Object Level Authorization
2. API2:2019 - Broken User Authentication
3. API3:2019 - Excessive Data Exposure
4. API4:2019 - Lack of Resources & Rate Limiting
5. API5:2019 - Broken Function Level Authorization
6. API6:2019 - Mass Assignment
7. API7:2019 - Security Misconfiguration
8. API8:2019 - Injection
9. API9:2019 - Improper Assets Management
10. API10:2019 - Insufficient Logging & Monitoring

And for 2023 [13] :

1. API1:2023 - Broken Object Level Authorization
2. API2:2023 - Broken Authentication
3. API3:2023 - Broken Object Property Level Authorization
4. API4:2023 - Unrestricted Resource Consumption
5. API5:2023 - Broken Function Level Authorization
6. API6:2023 - Unrestricted Access to Sensitive Business Flows
7. API7:2023 - Server Side Request Forgery
8. API8:2023 - Security Misconfiguration
9. API9:2023 - Improper Inventory Management
10. API10:2023 - Unsafe Consumption of APIs

The upcoming sections will address key challenges frequently encountered in cybersecurity. A major concern in distributed systems is the protection of data, underscoring the importance of robust data security measures. The following section will detail encryption and hashing techniques, which are critical components of data protection, as they are used in other approaches such as TLS/Secure Sockets Layer (SSL), data signing, authorization protocols, and trust management in general.

4.2 Data Protection Techniques: Encryption and Hashing

4.2.1 Essentials of Data Security

In the digital age, information security is paramount for privacy and reliability, especially with legislation dictating how users' sensitive data should be handled when it is stored. Encryption plays a key role in converting data into a coded format to prevent unauthorised viewing, accessible only with the appropriate decryption key. This is often combined with hashing, which creates a digital print of the current snapshot of the data. It can be said that encryption and hashing techniques are now crucial to protecting data transmissions over the Internet, as much of the data may be sensitive and users may wish to keep it private.

4.2.2 Hashing

[14] Hashing transforms data into a consistent-sized value or signature, serving as a digital fingerprint. It is a unidirectional technique, meaning the original information cannot be derived from its hash. It is mainly used for **integrity verification**, as even a small change to the source being hashed by a strong hash function will significantly alter the resulting hash. This function is essential for confirming the unaltered state of data, usually to validate the signature during the authentication process or software via checksums, which can prevent some suspicious software from being downloaded that could be used to attack a system.

Incorporating Salting and Peppering: [15] Salting is a technique where additional data added to the hashing process is mainly used to process the user's credentials, in this case mainly passwords, which are stored in the database. Credentials must not be stored in their plain-text form. This would make the system more potentially vulnerable to data breach and the attacker would have easier access to the user's information. Security is improved by adding a salt, or random data, to the data before it is hashed. Each salt should be unique, ensuring that even if several passwords are the same, each password will have a different hash thanks to the additional data. At the same time, the salt values must be stored with the passwords in the database to be able to reproduce the hash to verify its correctness, especially during the authentication process when the user enters the credentials. Peppering, on the other hand, consists of adding another secret, but in this case not to the data before the hash, but to the generated hash after salting. As mentioned by OWASP, one peppering strategy is to apply, for example, Hash-Based Message Authentication Codes (HMAC) to the original password hash before storing the password hash in the database, in which case the pepper acts as a HMAC key known only to the application. The fact that the value used for peppering is known only to the application, and is not stored in the database, makes it more difficult for an attacker who would steal the contents of the database containing the user's credentials to crack the passwords, since the attacker would have to know the peppering value first. This makes it even harder for the attacker to guess the original password, even using brute force or dictionary techniques.

4.2.3 Attacks to Hashes

As with any security approach used in the system, the hash is subject to some common attacks that are used to extract the original data from it, although the definition of the hash function remains that the hash cannot be transformed back into the original appearance. For this reason, various approaches are used on a daily basis to try to crack the crucial information hidden behind the hash, or the so-called fingerprint of the data. Here is a list of common attacks [16], [17]:

- **Brute Force Attacks:** A brute force attack is applied mainly in a situation when an attacker retrieved the database of (perharps hashes) of users' passwords, by for example an Structured Query Language (SQL) Injection attack, and is trying to guess the passwords by attempting to calculate the hash for random words combinations that may be taken from for example database of known passwords used among many users[1].
- **Rainbow Table Attacks:** A special table (a "rainbow table") of pre-computed password hash values for each plaintext character. An attacker can then use the stolen hashes of encrypted passwords and try to find a match with the rainbow table. This technique is often used when an attacker finds vulnerabilities in a system, such as outdated hashing algorithms. As mentioned by the OWASP in the **section 4.2.2**, where the salting and peppering techniques are described, this can make this type of attack significantly more difficult, since using

[1]Those are password such as: mom, dad, 123456789 and more. These passwords are recommended as not to be used in any system as these are the first attempts to be tried when trying to crack them.

salting requires an attacker to crack hashes one at a time using the appropriate salt, rather than calculating the hash once and comparing it to each stored hash. Peppering is an addition to salting that makes it impossible for an attacker to crack any of the hashes if the attacker only has access to the database without knowing the peppering value.

4.2.4 Encryption

[18] Encryption is a technique used to hide information from an unauthorized user. During this process, two participants must share a secret that is used to encrypt and decrypt the crucial information, the secret is called a key. There are two types of keys [19], [20]:

- **Symmetric Encryption:** This approach employs a singular key for encrypting and decrypting data. Advanced Encryption Standard (AES) and Data Encryption Standard (DES) are notable examples utilized for this encryption style.

- **Asymmetric Encryption:** Known too as public-key encryption, this strategy involves a public key for encryption and a private key for decryption, with RSA being a widely recognized method [21].

Application Scenarios

- **Safeguarding Data:** Applying encryption to data stored in databases or on physical storage. For example, AES encryption to secure files stored on a drive.

- **Securing Data in Motion:** Guaranteeing the safe transport of data over networks. TLS protocol, for instance, is used for securing web traffic.

- **Data Signing:** [22] One of the processes of data signing is illustrated in the **picture 4.3**. A participant who wants to verify the signature must obtain a public key from a participant who created the digital signature. Another way to create a digital signature is to use HMAC - in this case only one secret is needed to verify the signature. This process of creating a digital signature is also illustrated in the practical part of this thesis, in the **section 5.8**.

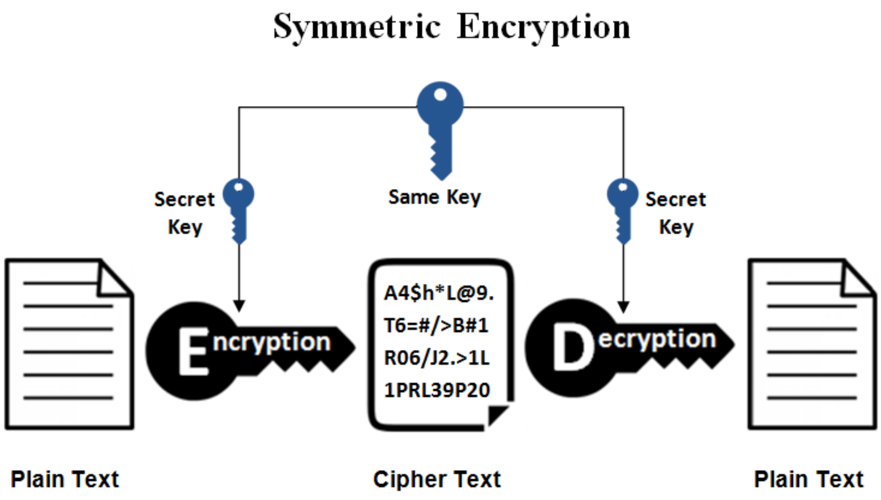


Figure 4.1: Symmetric encryption[23]

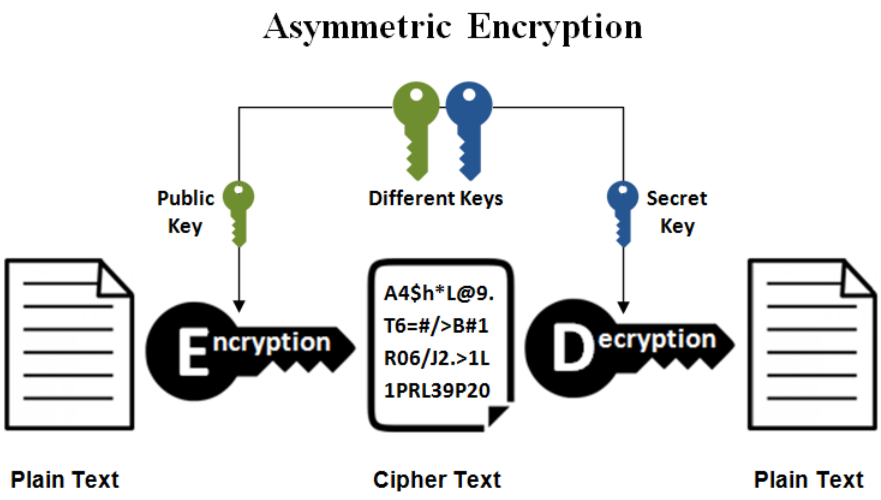


Figure 4.2: Asymmetric encryption[23]

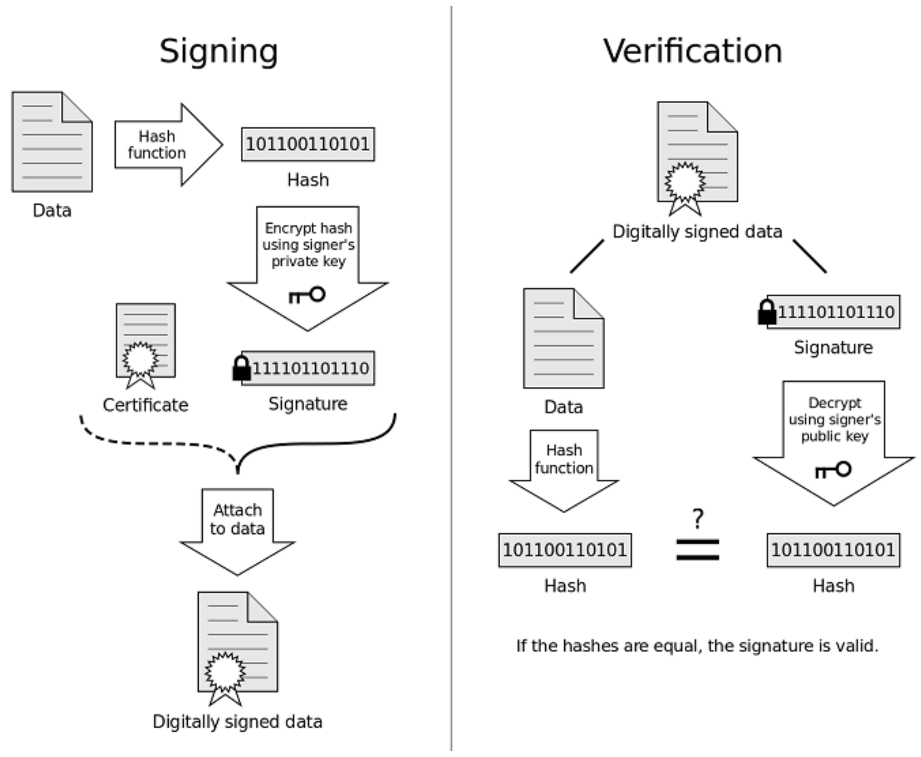


Figure 4.3: Digital Signature[24]

The illustrations **4.1** and **4.2** emphasize the central role of keys in encryption processes. The challenge, however, lies in the secure transmission of these keys. Protocols like TLS illustrate this challenge well, as they involve a negotiation process for secret keys to encrypt and decrypt communications securely. The integrity of the entire communication can be compromised if this key exchange is intercepted or manipulated, leading to vulnerabilities such as man-in-the-middle attacks. Therefore, the method of key transfer is as crucial as the encryption itself, underscoring the need for robust mechanisms to protect against unauthorized interception and ensure the confidentiality of the communication.

Difference between Hash and Encryption Techniques: [25] The main difference between the hashing techniques mentioned in the **section 4.2.2** and the encryption is the fact that it is possible to revert encrypted data to the previous state, this is called the process of decryption, unlike the hashing which is a one-way function only that creates unique string that is not reversible back to the previous value. Another difference is the speed of the process. Hashing functions are considered to be faster algorithms than the encryption algorithms. In particular, signing algorithms take advantage of the idempotency and speed of hash functions by first creating a much smaller hash from a (presumably large) input, and then using a slower encryption algorithm to create a digital signature of the hash.

4.3 Transport Layer Security (TLS) / Secure Sockets Layer (SSL)

[26] In the digital era, ensuring secure communication across the internet is paramount, particularly for web applications and services. TLS, along with its predecessor SSL, stands at the forefront of addressing this need by providing a robust mechanism for encrypting communications. This section delves into the significance of TLS/SSL within the realm of cybersecurity.

4.3.1 Definition of TLS / SSL

TLS and its precursor, SSL, are fundamental to safeguarding data transmitted over the Internet. Originally developed for encrypting web communications, their applications now extend to email, messaging, and Voice Over IP (VoIP) among others. The inception of TLS by the Internet Engineering Task Force (IETF) marked a significant advancement in internet security protocols, with TLS 1.3 being the latest iteration, emphasizing enhanced privacy and data security.

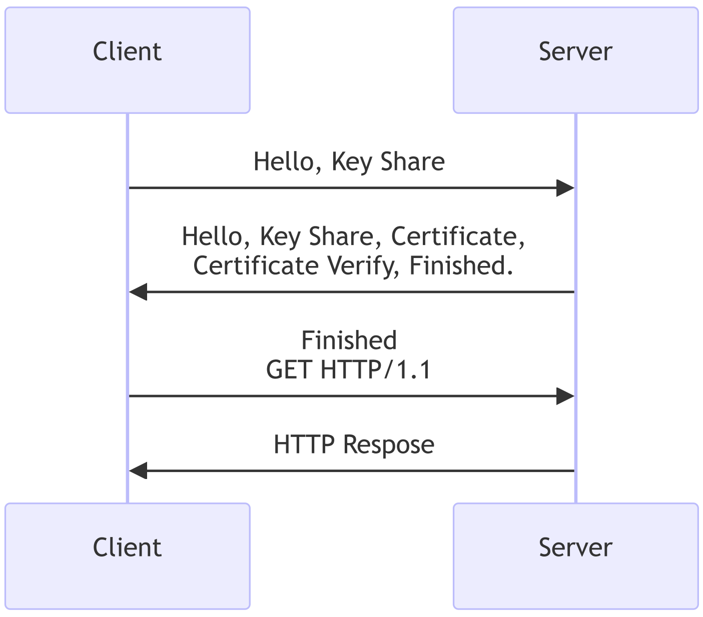


Figure 4.4: TLS 1.3 Handshake Illustration

4.3.2 Importance in Cybersecurity

At its core, TLS/SSL serves three key purposes: encryption, authentication, and integrity. Encryption conceals the data exchange between parties, preventing unauthorized access. Authentication confirms the identity of the entities involved in the communication, ensuring that users are actually interacting with the intended servers. Lastly, integrity checks guarantee that the data transmitted remains unaltered and secure from tampering.

The utilization of TLS/SSL is particularly critical for web applications, where Hypertext Transfer Protocol Secure (HTTPS) employs TLS encryption to secure web traffic. The widespread adoption of HTTPS, driven by initiatives from major web browsers and heightened user awareness, underscores the importance of TLS/SSL in establishing a secure and trustworthy internet ecosystem.

4.4 Identity and Access Management (IAM)

This section discusses IAM and its related technologies and tools that are mainly used in cloud computing and distributed systems. It includes a wide range of cybersecurity methods like authentication, authorization, digital certificates, Risk-Based Authentication (RBA), encryption, hashing, and many more. Some of these methods are shown in a demo application described in Chapter **5**. It aims to mention at least some of standard security technologies, tools, and practices, using reliable sources like the OWASP Foundation and NIST.

Oracle defines IAM as a critical framework for protecting user information and managing access within an organization's systems, including both cloud-based and on-premise[2] resources. [27].

This subsection describes the basics of authentication, authorization and protocols where these approaches are used, as well as the principles of secure data management to prevent data breaches.

4.4.1 Authentication

This section examines prevalent methods of authentication and authorization deployed across various applications. These methods are critical not

[2] An on-premise solution is a replacement for a cloud-based solution, where the software typically runs in the customer's own environment rather than in a remote facility such as the cloud.

only in globally accessible cloud environments but also within intranets. Ensuring the security of both externally facing APIs and internal applications is essential for safeguarding business-critical processes. This thesis describes the concepts of digital identity, identity verification, and session control, offering guidelines primarily advocated by the OWASP Foundation. Practical illustrations are provided through links to a demo application, demonstrating some of these principles in action.

4.4.2 Process of Authentication

The process of authentication in this concept is defined by trustful sources in the area of cybersecurity as follows:

- “Security measures designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual’s authorization to receive specific categories of information.“[28]
- “Authentication (AuthN) is the process of verifying that an individual, entity, or website is who or what it claims to be by determining the validity of one or more authenticators (like passwords, fingerprints, or security tokens) that are used to back up this claim.“[29]

[29] In this context of authentication, it is important to define also the following subjects: **Digital Identity**, **Identity Proofing**, and lastly **Session Management**:

- **Digital Identity**: serves as a distinct marker for individuals participating in online activities. It is designed to be unique within the digital realm, though it is not always directly linked to an identifiable person in the physical world.
- **Identity Proofing**: The process of identity verification, or identity proofing, is crucial in confirming that an individual is who they claim to be. This process parallels the principles of Know Your Customer (KYC), striving to associate a digital presence with an actual person.
- **Session Management**: Managing sessions involves a server’s ability to keep track of a user’s interaction over time. This capability is essential for the server to consistently respond to a user’s actions during an online session.

The following subsections are dedicated to the authentication approaches (Authentication General Guidelines). Although the main resource is OWASP Foundation, the guidelines do not differ from the guidelines that can be found at other sources, such as NIST, National Cyber Security Centre (NCSC), and more, as those principles are already well known among cybersecurity experts. However, this paragraph does not necessarily cover all possible approaches that are available to be used. The cybersecurity scene is really fast and new approaches are most likely to be found and some old ones can be already outdated due to their insufficiency.

All the guidelines are linked with the demo application that starts from chapter no. 5 (Practical Implementation).

Authentication Risks

One critical vulnerability in digital security is broken authentication, as mentioned in the OWASP Top 10. This flaw arises when authentication procedures are incorrectly implemented, allowing attackers to assume other users' identities or access unauthorized information. Authentication mechanisms, being openly accessible, present a tempting target for exploitation. The complexity and common misconceptions about authentication contribute to the widespread occurrence of these issues. Attackers can potentially gain full control over user accounts, access personal data, and undertake actions indistinguishable from those of legitimate users[30].

Identifying Vulnerabilities An application may be deemed vulnerable to broken authentication if it exhibits (not only) any of the following characteristics described by the OWASP Foundation:

- It is vulnerable to credential stuffing, a type of attack where the attacker uses brute force with a list of possibly valid usernames and passwords. Very similar to the rainbow table attack mentioned in the **section 4.2.2**.
- It lacks adequate measures, like captcha or account lockout mechanisms, to prevent repeated brute-force attempts on the same user account. This problem is related to the RBA solution which is mentioned later in the **section 4.4.8**.
- It allows the use of weak passwords, increasing the risk of successful brute-force attacks.

- It transmits sensitive authentication details, such as tokens or passwords, in URLs. All the data transmitted in the URLs are traceable as they are not hidden from anyone.
- It does not require password confirmation for critical operations, such as changing email addresses or passwords. Which may lead to account theft. If such an account could have higher roles in the system as a bonus, the attacker would have more opportunities to misuse the system and its data.
- It fails to verify the authenticity of tokens or accepts tokens with weak or no signatures. This can lead to access by users who may compromise or steal the token to gain easy access to the system.
- It stores passwords in plain text, without encryption, or employs weak hashing techniques. Which leads to easier theft with attack techniques that have been mentioned for example in the **section 4.2.2**.

Practical Measures for Prevention To mitigate the risks of broken authentication, several preventive measures can be adopted:

- Understanding all possible authentication flows and ensuring they are secure.
- Adhering to standard practices for authentication, token generation, and password storage.
- Treating credential recovery endpoints with the same security measures as direct authentication pathways.
- Implementing multi-factor authentication where feasible to add an extra layer of security.
- Establishing anti-brute force mechanisms to protect against various types of automated attacks.
- Requiring re-authentication for sensitive operations within the application.

These strategies, along with insights from the OWASP Authentication Cheat Sheet, form the cornerstone of secure authentication practices. Chapters 5 and 6 of this thesis, which detail the Practical Implementation and Testing and Quality Assurance of the Demo Application, respectively, will further explore how tools like Zed Attack Proxy (ZAP) can be utilized to identify and address broken authentication vulnerabilities in real-world scenarios[30].

4.4.3 User Identification and Management

[29] As highlighted by the OWASP Foundation, safeguarding the user's identifier within a system is crucial. A direct method to enhance security is by randomly generating the user identifier. This approach helps in avoiding the creation of predictable or sequential Identifiers (IDs), which could elevate security risks. Such precautions are particularly vital in environments where User IDs may be exposed or deduced from external sources.

Definition of Username and Email Within System: A username is a memorable identifier that is chosen by individuals to represent themselves when they log on to a system or service. It is recommended that individuals are allowed to use their email address as a username, subject to email verification during the registration process. In addition, users should be allowed to choose a username that is different from their email address.

However, as the username or email address is the user's prompt, it is important to keep in mind that all the inputs that are inserted must be properly validated to avoid the potential risk that could be caused by an attacker. Further details on approaches to input validation can be found in Section **4.7.1** of this document.

Additionally, it is imperative to enforce specific security measures concerning account access and authentication practices that are strongly recommended by the OWASP Foundation:

- Sensitive accounts that can be used for the more advanced operations within the system (e.g. direct access to the back-end / middle-ware / database) should not be allowed to log in to any front-end user interface.
- An Authentication solution (e.g. Identity Provider (IDP)[3] / Active Directory (AD))

The following subsections discuss various authentication mechanisms such as Username & Password, Internet Protocol (IP) Address, X.509 and more.

4.4.4 Username & Password Authentication

The Username and Password is a fundamental method for user authentication. While simple and straightforward to implement, this method neces-

[3]The identity provider can be a banking system, for example. The user's identity is verified by logging into the system, as the identity has already been verified by the bank when the account was created.

sitates careful handling by developers to protect against password breaches and other potential attacks.

As described in article [31], and in the subsection above (**4.4.3**), username and password authentication involves verifying a user's identity through a unique identifier (username) and a secret (password) against a stored database.

To bolster security when employing this method, developers can adopt several practices:

- Implementing a robust password policy,
- Utilizing hashing with salting for password storage,
- Employing RBA to identify anomalies.

Some of the key challenges associated with password authentication:

- **Password Reuse:** A significant risk arises when credentials from one system are compromised, as attackers may attempt to use these credentials across various platforms.
- **Password Guessing:** Attackers often employ dictionaries of common passwords in their attempts to breach user accounts. Techniques like RBA can mitigate such risks by detecting and preventing suspicious login attempts.

Password Policy

Ensuring password strength is critical to secure authentication. A strong password policy makes it difficult for attackers to guess or crack passwords using manual or automated methods. Key elements of a strong password include:

- **Password length**: It is important to have a password policy in place that limits the length of a password and prevents users from creating passwords that are too weak or unnecessarily too complicated.
- **Character diversity**: Encourage the use of a variety of characters, including Unicode and spaces, without imposing restrictions on character types. This flexibility supports the creation of stronger passwords.

- **Compromise Response**: Prompt password changes in the event of a security breach or when a compromise is detected to protect user accounts.
- **Password strength meter**: Include tools such as the zxcvbn-ts library to help users generate complex passwords and prevent the selection of common or previously compromised passwords.
- **Vulnerability checking**: Use services such as Pwned Passwords to check against a database of known compromised passwords. This service can be integrated via an API or directly hosted.

Storing the Password

[32] It is very important to keep user credentials safe, especially if someone breaks into an application. As mentioned in the previous **section 4.2**, the way passwords are stored in the database is crucial for the whole application. Only password hashes with the addition of salting and peppering, as mentioned in the **section 4.2.2**, should be stored. There are some pre-existing methods that are deliberately slow to slow down any attempt at password cracking. Such methods are called mixing methods. Some of these are, for example

- **bcrypt:** is older, but still very reliable, especially for systems that can not use the latest methods yet.
- **Argon2id:** is the newest and is recommended for its ability to effectively fend off hacker attacks. It allows you to adjust the settings according to how much security you need.
- **PBKDF2:** is often required to meet certain security standards and is a solid choice for many applications.

The name “mixing” comes from the principle of mixing passwords to increase their strength. Each has its own set of suggestions, but they all add a bit of unique data to each password before mixing it, making it harder for attackers to figure out the passwords.

Risks with Passwords

Not having a strong rule for passwords can lead to dangers like brute-force attacks, where hackers try many passwords to get in. This can let them into user accounts without permission. Having a variety of checks for user logins can greatly lower this danger[32].

In summary, this section on username/password authentication, including the password policy, is intended to cover basic aspects and considerations for developers implementing this method. Despite its widespread usage, the shift towards passwordless methods, including Single Sign-On (SSO), signifies evolving security paradigms in application development. The username and password approach is illustrated in the practical part, **subsection 5.5.1**.

4.4.5 IP Address Authentication

[33] IP address authentication allows users access to certain services based on their computer or device's IP address, usually managed by an organization's Information Technology (IT) department. This method, often used for securing access to specific online resources within an organization's physical location like campuses or offices, requires additional tools like EZproxy[4] or HAN proxy[5] for off-site access.

This form of authentication is favored for its ease of setup and maintenance, providing a frictionless experience for users. However, it falls short in supporting modern remote access needs without additional remote authentication tools. Moreover, it limits user actions, such as placing holds on library items, unless they're within the recognized IP range.

Users accessing resources from a registered IP address enjoy seamless entry to organization-specific resources. Conversely, access from unregistered IPs necessitates navigating through a proxy tool, initiated from the library's website or similar platforms, potentially complicating the user experience.

For organizations with static IP ranges, IP authentication remains a viable option. Yet, those with dynamic or frequently changing IP ranges might find maintenance burdensome. In such cases, exploring more stable authentication methods or considering Personal User Authentication for remote access could prove beneficial. Ensuring your authentication settings are up-to-date, including proxy configurations, is also recommended for a smoother user experience.

[4]A technology that provides remote access to Electronic Information Resources by means of EZProxy links: `https://ezdroje.muni.cz/vzdaleny_pristup/ezproxy.php?lang=en`

[5]HAN is based on the technological concept of a reverse proxy: `https://www.hh-han.com/en/concept.cfm`

4.4.6 X.509 Certificates

[34] An X.509 certificate is like a digital passport for websites or electronic identities. It confirms that the public key contained within the certificate belongs to the person, organization, or device it claims to. These certificates rely on a standard format for Public Key Infrastructure (PKI) to work across the internet. The main structure of the certificate is shown below [35]:

```
Certificate  ::=  SEQUENCE  {
  tbsCertificate       TBSCertificate,
  signatureAlgorithm   AlgorithmIdentifier,
  signature            BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
  version         [0]  Version DEFAULT v1,
  serialNumber         CertificateSerialNumber,
  signature            AlgorithmIdentifier,
  issuer               Name,
  validity             Validity,
  subject              Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                      -- If present, version MUST be v2 or v3
  subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                      -- If present, version MUST be v2 or v3
  extensions      [3]  Extensions OPTIONAL
                      -- If present, version MUST be v3 --  }
```

Usage of X.509 Certificate

As mentioned in the source [36], the X.509 may also be used as an authentication service. This digital certificate is a certificate-based authentication security framework that may be used for securing transaction processing and private information. In particular, it is used to handle security and identity in computer networks and Internet-mediated communications in general.

The core of this authentication service is the public key certificate associated with each user. Each key is assumed to have been generated by a trusted certification authority and provided to a user by the certified authority. This certificate is then used as a credential.

The X.509 Authentication Service Certificate has many applications because many protocols depend on this standard. Some of these are illustrated below:

- Document signing and Digital signatures,
- TLS / SSL certificates,
- Email certificates,
- Code signing,
- Secure Shell Protocol (SSH),
- Digital Identities

The Working Mechanism of X.509 Certificates

At its core, an X.509 certificate's job is to establish a secure link. This process involves:

1. **Issuing the Certificate**: A trusted authority checks and then issues a certificate to the requester.
2. **Validating the Certificate**: When someone connects to a service, like a website, their browser checks the site's certificate is valid.
3. **Establishing Secure Communication**: Once the certificate is verified, a secure connection is made, protecting the data exchanged[37].

Why X.509 Certificates Matter

These certificates are foundational to online trust and security:

- **Building Trust**: It helps users feel secure, knowing they are possibly talking to the website they think they are.
- **Protecting Data**: It is used during the handshake between a client and a server to establish secure communication when adding an extra layer of security to the communication: SSL / TLS.
- **Proving Identities**: They ensure that the parties in any digital communication are who they claim to be.

4.4.7 WebAuthn

[38] WebAuthn, or Web Authentication, stands as a modern API standard by W3C aimed at enhancing user login methods for online services and websites. It supports authentication via biometrics, like fingerprint or facial recognition, and hardware authenticators such as USB or NFC tokens.

Features of WebAuthn

- **Phishing Resistance:** By employing public key cryptography, where the private key remains on the user's device, WebAuthn effectively guards against phishing. Credentials are domain-specific, preventing their misuse on fraudulent sites.
- **High Security Standards:** WebAuthn adheres to stringent cryptographic protocols, ensuring secure user verification through either roaming or platform authenticators, depending on the device's capabilities.
- **Seamless User Experience:** WebAuthn facilitates a frictionless login process, allowing authentication with a simple gesture, thereby balancing convenience with heightened security.

Roles in WebAuthn

- **Relying Party:** The online service or website requiring user access, initiating the authentication process.
- **WebAuthn Client Device:** The user's device, hosting private keys securely, supporting user authentication.
- **User:** The individual seeking access to the relying party's resources.
- **Authenticator Types:**
  - **Platform Authenticators:** Built-in device features like biometrics.
  - **Roaming Authenticators:** External devices, e.g., YubiKey, enhancing cross-platform security.

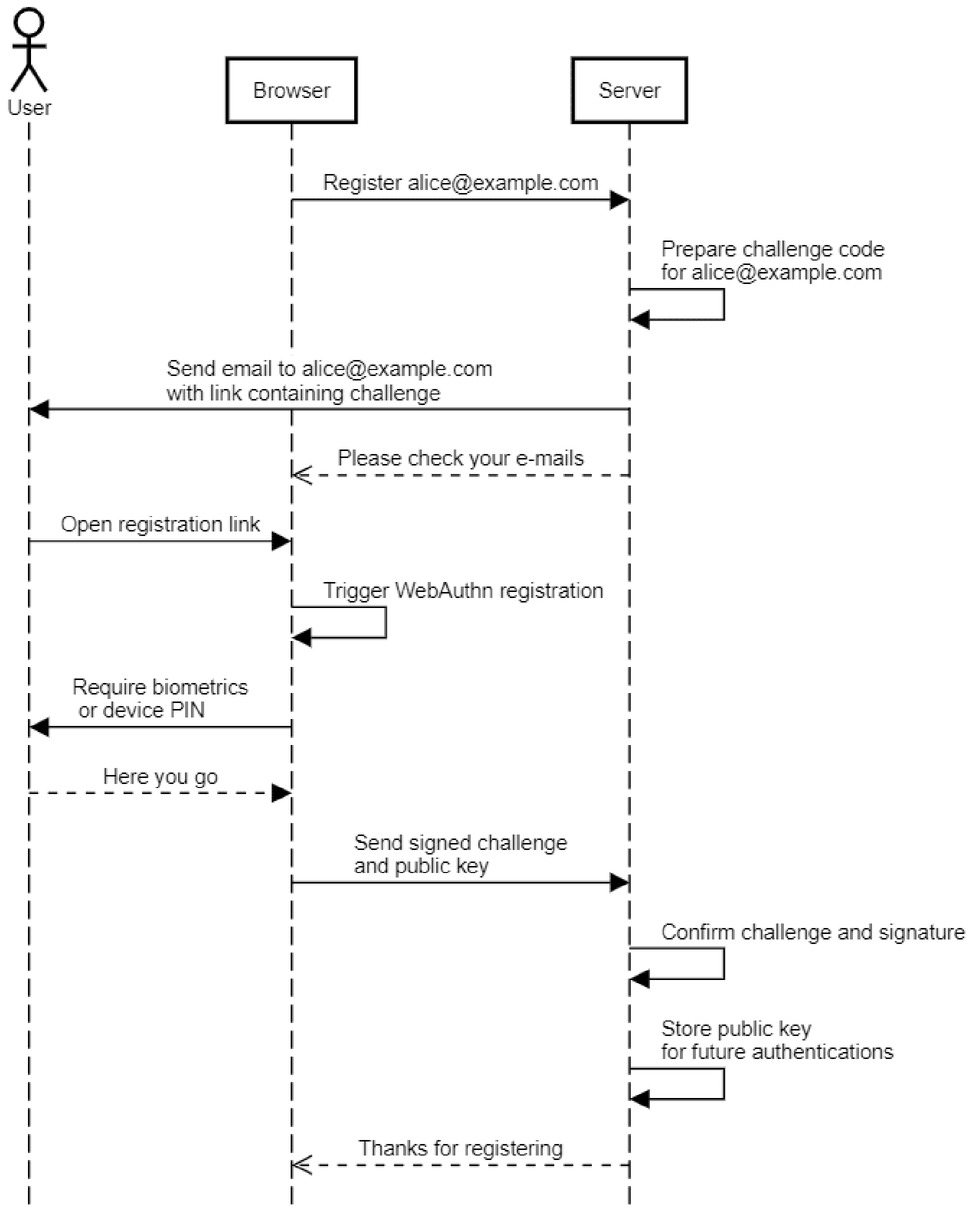
How WebAuthn Operates

WebAuthn authentication involves two primary flows:

- Registration (Sign Up Operation),
- Authentication (Sign In Operation).

Both operations are illustrated in the images provided on the next page of this thesis, which are taken from the source cited in the title of the images:

Registration (Sign Up Operation) Flow



User
Browser
Server
Register alice@example.com
Prepare challenge code for alice@example.com
Send email to alice@example.com with link containing challenge
Please check your e-mails
Open registration link
Trigger WebAuthn registration
Require biometrics or device PIN
Here you go
Send signed challenge and public key
Confirm challenge and signature
Store public key for future authentications
Thanks for registering

Figure 4.5: WebAuth Sign Up Flow [39]

Authentication (Sign In Operation) Flow

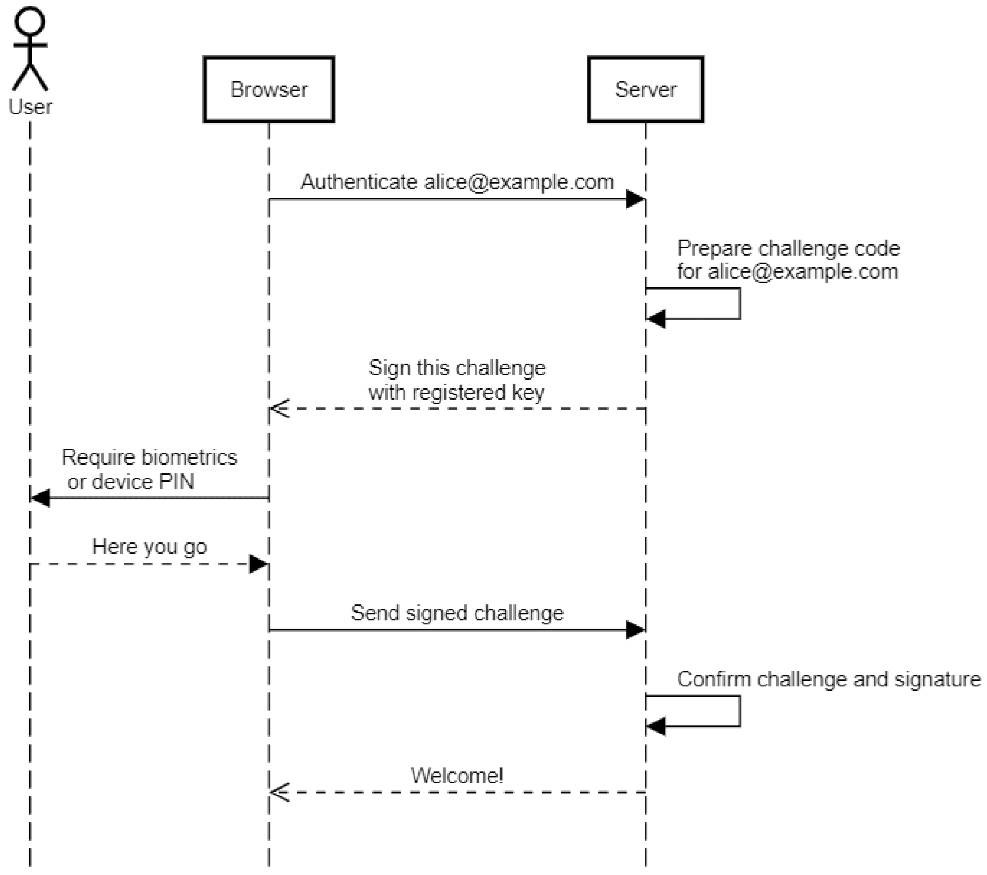

User
Browser
Server
Authenticate alice@example.com
Prepare challenge code for alice@example.com
Sign this challenge with registered key
Require biometrics or device PIN
Here you go
Send signed challenge
Confirm challenge and signature
Welcome!

Figure 4.6: WebAuth Sign In Flow [39]

WebAuthn is a revolutionary approach because it eliminates the need to create and manage passwords. This new method improves security because it requires a special device, like a Universal Serial BUS (USB) or Near Field Communication (NFC) token, to log in. Without this device, attackers can not get into the application. These devices are important for creating secure keys, which is a way to keep information safe, as have been mentioned in section **(4.2)**. This shows the importance of the traditional security approaches in the field of cybersecurity, especially in today's world where cloud computing is increasingly used and security plays a key role in protecting not only the data but also the users.

4.4.8 Risk-Based Authentication (RBA) Method

RBA is a sophisticated approach to enhancing traditional password authentication systems. It works by monitoring extra details, like the type of device being used or where the user is located, while they enter their password. If RBA spots anything unusual that could mean a higher security risk, it asks for more proof that the person is who they say they are, such as a code sent to their phone[40].

The idea behind RBA is to make it harder for bad actors to get into someone's account, even if they have somehow figured out the password. This method is backed by guidelines from NIST on digital identity and is already being used by many big online platforms. It is particularly good at stopping a range of attacks aimed at stealing user data, like when hackers try to break into many accounts at once using stolen password lists, guess passwords, or take advantage of weak passwords. It is not just about preventing account theft, but also about the security of the system as a whole, since any client could be affected by an attacker who might try to make the system unavailable.

Unfortunately, the exact way RBA works is not usually shared by companies that use it. This lack of openness means there is not much detailed research out there about how RBA is put into practice. This situation slows down progress in making RBA better and more widely understood. For users, it means it is tougher to know just how secure the online services they rely on really are. It also makes it harder for RBA to be adopted more broadly, even though it can significantly improve online security[40].

4.4.9 Authorization Overview

Authorization plays a critical role in securing applications by ensuring that users can only access resources for which they have permissions. It is distinct from authentication, which verifies user identity, whereas authorization verifies user permissions after their identity has been confirmed.

4.4.10 Process of Authorization

The process of authorization involves determining user rights and permissions to various resources within an application. This process is pivotal in maintaining the security and integrity of the application's data.

Authorization Risks

Authorization risks arise when users are granted access to resources or data for which they do not have legitimate need or permission. These risks can lead to unauthorized data access, data manipulation, or exposure of sensitive information.

4.4.11 Recommendations for Effective Authorization

To mitigate authorization risks and ensure a robust security posture, the following recommendations should be considered:

Least Privileges

Employ the principle of least privilege by ensuring that users have the minimum level of access, or permissions, needed to perform their duties. This minimizes the potential impact of a breach by limiting the resources an attacker can access or compromise.

Deny by Default

Adopt a deny-by-default stance where access permissions are not granted unless explicitly defined. This approach ensures that users are only able to access resources that have been specifically allowed, reducing the risk of unauthorized access. This is based on the least privileges principle.

Validate Permissions on Every Request Permissions should be validated for each request to ensure that users have appropriate access rights.

Attribute and Relationship-Based Access Control

Under Role-Based Access Control (RBAC), individuals are organized into roles, with each role granted certain permissions. This setup simplifies the management of access rights, as any changes to roles or their permissions automatically reflect on all users linked to those roles. For instance, within any IAM system, roles might encompass permissions such as the ability to read, write, or delete content, and users are allocated roles aligning with their job functions.

4.4.12 Identity and Access Management Technologies

Well known cloud service providers, including Microsoft, Amazon Web Services (AWS), and International Business Machines Corporation (IBM), deliver complete IAM solutions that integrate a broad spectrum of cybersecurity practices. They generally offer greater security and reliability than custom development, which can expose vulnerabilities or lead to bad design decisions. Comprehensive IAM solutions incorporate advanced authentication protocols, such as OpenID Connect (OIDC) and Security Assertion Markup

Language (SAML 2.0), along with Multi-Factor Authentication (MFA), resource access policies, user groups, sign-in and sign-up processes, SSO, and extensive monitoring features.

The adoption of prebuilt IAM solutions is highly recommended to mitigate the risk of common cybersecurity threats effectively and to avoid the pitfalls associated with custom-built systems. These platforms provide a robust framework for managing digital identities, enhancing system security, and ensuring compliance with evolving cybersecurity norms.

[27], [41] This is a list of the most common technologies that are included in the IAM:

- **Single Sign-On (SSO):** Or in other words Identity Federation is a technology that allows users to log in once and access multiple systems without having to re-authenticate to each system. This reduces the need for users to maintain multiple passwords for each service. Another benefit is that the systems do not need to maintain the user's credentials, as they should already be authenticated and authorized by some third parties.

- **Multi-Factor Authentication (MFA):** MFA enhances security by requiring users to provide multiple forms of verification before gaining access. This may be necessary to allow the user to perform some critical operations, such as accessing resources that require higher privileges or operations (not only) related to the account settings, such as changing password and email. Nowadays this solution is crucial for example on the social media sites (e.g. Facebook), as it would be really easy to steal someone's account if there would be possibility to change the user's settings like phone number, email and more without any additional validation steps required. This makes it much more difficult for the attacker.

- **Directory Services:** Utilizing protocols like Lightweight Directory Access Protocol (LDAP), directory services manage identity information for users, groups, and devices without requiring additional custom implementation by default, eliminating the risk of creating potentially dangerous parts of the application that may be incorrectly implemented and exposed to attackers.

- **User Provisioning:** It automates the creation, modification, and deletion of user accounts across multiple systems. In other words, it allows system administrators to manage users across multiple systems from a single location. User management includes managing their roles to perform actions and access resources according to their privileges.

- **RBA:** RBA manages authentication challenges based on the risk profile associated with a user's behavior. This is useful for early detection of a malicious user based on the user's actions within the system(s). This approach is described in more detail in the **section 4.4.8**.
- **Cloud Identity Management Solutions:** Technologies like AWS Cognito, Azure Active Directory, and Google Cloud Identity provide cloud-based solutions for managing user identities and access rights efficiently.

4.5 Authentication and Authorization Protocols

In cybersecurity, authentication protocols are crucial for verifying the identities of users, systems, or entities before granting access to sensitive information, services, or networks. These protocols form the backbone of security mechanisms, ensuring that access to digital assets is appropriately controlled. The evolution of cyber threats necessitates robust authentication protocols to protect against unauthorized access.

Authentication protocols are agreements on how credentials are exchanged between a requesting party and the system providing access. They specify methods for presenting and verifying credentials, such as passwords, digital certificates, or biometric data, with their effectiveness measured by their resilience to attacks, ease of implementation, and usability. The main advantage of these protocols is lack of need to send user's credentials, especially password.

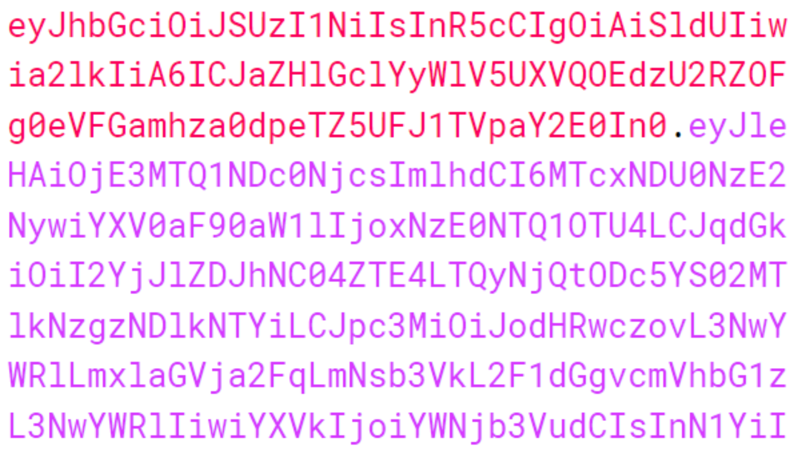
The diversity of authentication protocols mirrors the broad spectrum of applications and security requirements in today's interconnected environment, from simple password-based methods to advanced cryptographic systems, all aiming to secure digital identities and facilitate secure transactions and communications.

4.5.1 JSON Web Tokens (JWT)

This format deserves its own subsection, as it is very often used to hold information about the current client, especially in the OIDC protocol, and is also sent with every request, typically in the authorization header when interacting with a web application that requires this type of token. The application can then use this token to decide whether or not the user can access the resource. The token has three parts:

1. `Header`: Contains information about the algorithm and token type
2. `Payload`: Contains the main data describing the client
3. `Verify Signature`: This part is important to prevent token compromise. This is a digital footprint of the token to verify that the token has not been modified by a malicious client.

Each part of the token is then separated by the "dot (.)" character. The representation of the token is shown in an image **4.7**.



```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
ia2lkIiA6ICJaZHlGclYyWlV5UXVQOEdzU2RZOF
g0eVFGamhza0dpeTZ5UFJ1TVpaY2E0In0.eyJle
HAiOjE3MTQ1NDc0NjcsImlhdCI6MTcxNDU0NzE2
NywiYXV0aF90aW1lIjoxNzE0NTQ1OTU4LCJqdGk
iOiI2YjJlZDJhNC04ZTE4LTQyNjQtODc5YS02MT
lkNzgzNDlkNTYiLCJpc3MiOiJodHRwczovL3NwY
WRlLmxlaGVja2FqLmNsb3VkL2F1dGgvcmVhbG1z
L3NwYWRlIiwiYXVkIjoiYWNjb3VudCIsInN1YiI
```

Figure 4.7: Part of the JWT Token

The content of the decoded token may then look like this (only part of it):

```
...
"resource_access": {
  "account": {
    "roles": [
      "manage-account",
      "manage-account-links",
      "view-profile"
    ]
  },
  "client": {
    "roles": [
      "role1",
      "role2"
    ]
    }
  },
  "scope": "openid email profile",
  "sid": "2bf75c36-5a94-4612-b472-13db68331b0a",
  "email_verified": false,
  "name": "<name>",
  "preferred_username": "test",
  "given_name": "<given_name>",
  "family_name": "<faimly_name>",
  "email": "<email_address>"
  ...
}
```

Listing 4.1: Decoded JWT Token

4.5.2 Open Authorization (OAuth) 2.0

[42] Defined by the IETF in RFCs 6749 and 6750, Open Authorization (OAuth) 2.0 is a framework that lays down a standardized method for secure authorization. Unlike its predecessor, OAuth 1.0, which was more focused on client authentication and cryptographic signatures, OAuth 2.0 introduces a more flexible and extensible framework primarily aimed at authorizing third-party applications to access server resources on behalf of a user. It allows services like Facebook, GitHub, and Google to grant application access to user information over HTTP, without sharing the user's credentials.

[43] It is important to note once again that OAuth 2.0 is an authorization protocol and not an authentication protocol. This is due to the fact that this protocol is primarily designed for granting access to specific resources on the Internet, such as APIs or user data in general. In order to achieve this, so-called access tokens are used, which contain the necessary information about the current authorization of a specific client that receives this token. This token is often represented as a JSON Web Token (JWT) token[6], which is described in more detail in the **subsection 4.5.1**. This type of token allows the insertion of a lot of information about the client, which can then be used in systems to determine whether or not the client is allowed to access the resources. A diagram **4.8** illustrates the flow of how OAuth 2.0 works in general.

[6]The content of the JWT token can be viewed by using `https://jwt.io` for example. This is a very useful debugging tool.

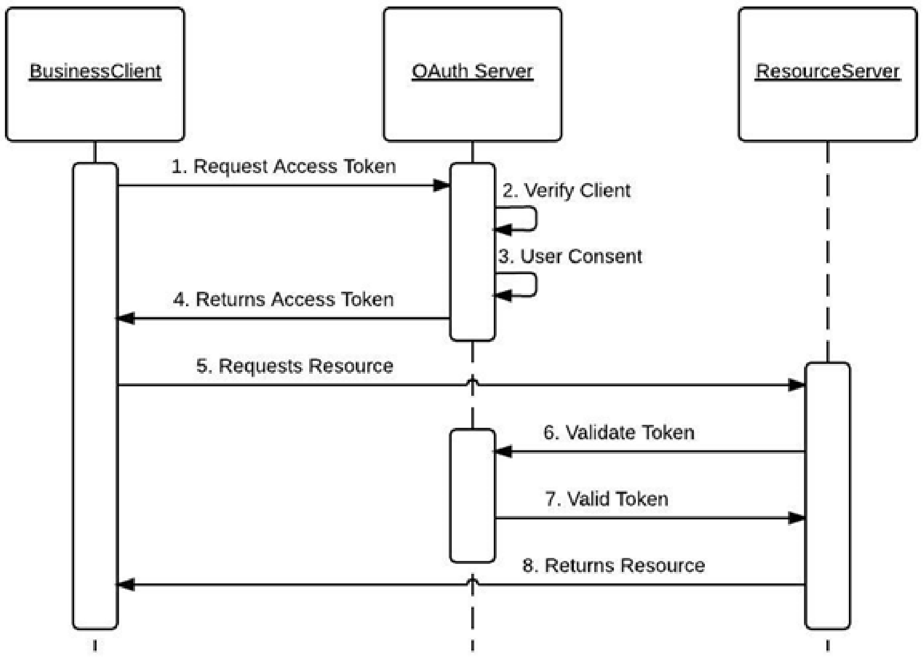


Figure 4.8: OAuth 2.0 Flow Illustration[44]

OAuth 2.0 Roles

The OAuth 2.0 framework specifies four roles:

- **Resource Owner**: Typically the user, who can grant access to their resources (data).
- **Resource Server**: The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- **Client**: The application requesting access to the resource owner's data.
- **Authorization Server**: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Authorization Grants

As also described in the [43] source, in OAuth 2.0, grants are the set of steps that a client must perform to obtain authorization to access a resource. The authorization framework provides several types of grants to address different scenarios:

- **Authorization Code Grant:** The Authorization Server issues a one-time Authorization Code to the Client. This code is then exchanged for an Access Token. This method is optimal for traditional web applications that handle the token exchange securely on the server side. Although also applicable to Single Page Application (SPA)s and mobile apps, these cannot securely store the client secret, thus limiting authentication to only the client ID during token exchange. An enhanced alternative for these applications is the Authorization Code Grant with PKCE.

- **Implicit Grant:** This simplified procedure directly sends the Access Token to the Client. Traditionally, the token might be delivered through the callback Uniform Resource Identifier (URI) or as a response to a form submission. The former method is generally discouraged due to the risk of token exposure.

- **Authorization Code Grant with PKCE (Proof Key for Code Exchange):** [45] This method extends the Authorization Code Grant by incorporating additional security measures, making it more secure for mobile apps and SPAs, as these types of applications cannot securely store a client secret. This approach introduces a secret created by the calling application that can be verified by the authorization server itself. The calling application also creates a transient value of the code verifier, called a code challenge, which is sent within the request over HTTPS to retrieve an authorization code. This prevents a malicious attacker from obtaining a token because the attacker does not have the code verifier.

- **Resource Owner Password Credentials Grant:** In this grant, the Client must first acquire the resource owner's credentials to request the token directly from the Authorization Server. This grant is only suitable for highly trusted clients due to the sensitive nature of handling user credentials directly. It is particularly useful where redirecting to the Authorization server is not viable.

- **Client Credentials Grant:** This grant is intended for applications acting on their own behalf rather than on behalf of a user, such as automated processes or microservices. The client is authenticated using its client ID and secret.

- **Device Code Grant:** Designed for devices with limited input capabilities, like smart TVs, this grant allows a secondary device to facilitate the authorization process.

- **Refresh Token Grant:** This involves exchanging a Refresh Token for a new Access Token, enabling extended access without requiring the user to re-authenticate.

4.5.3 OpenID Connect (OIDC)

Built on OAuth 2.0, OIDC extends this authorization framework to include user authentication, thereby enabling clients to verify users' identity efficiently and access basic profile information via a standardized protocol [46].

[47] To define the OIDC, it is also important to mention another terminologies that are related to this protocol:

- `Relying Parties (RP)`: Commonly web applications, visited by clients, that requires users to be authenticated to perform specific actions (e.g. fetch data from other other applications, or access specific sources on the web application and more).

- `OpenID Connect Provider (OP)`: Provider, or in other words, a service that uses IDP to obtain user's information once the user is authenticated successfully. The IDP may be own solution, typically database with user's credentials, or external services such as LDAP, SAML 2.0, Google and many more.

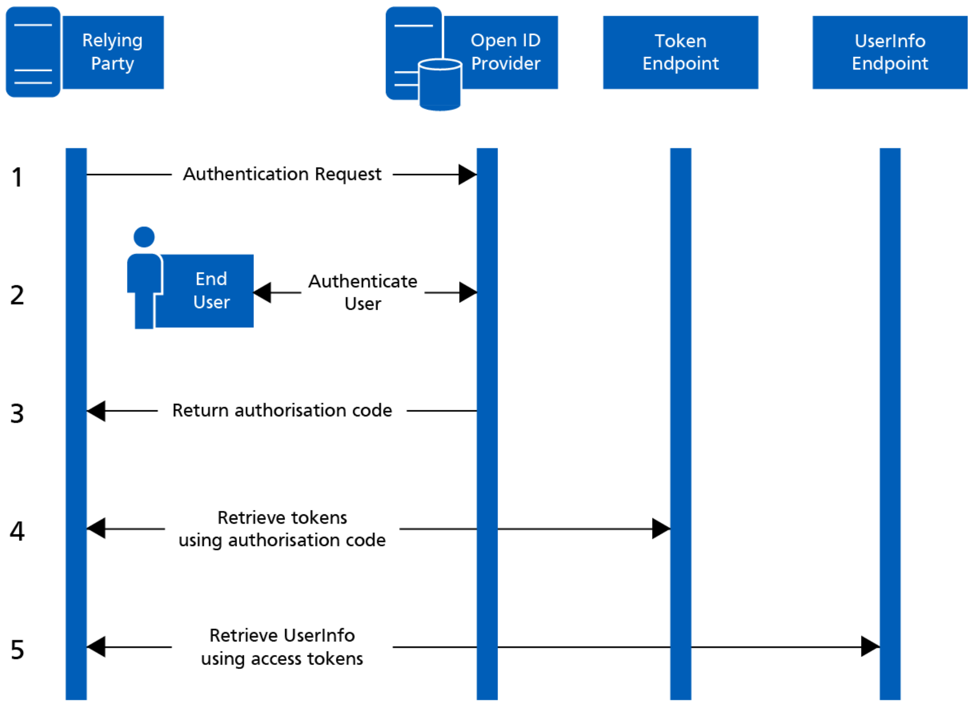


Figure 4.9: OIDC Authentication Illustration[48]

[49] As mentioned at the beginning of this subsection, the OIDC is an extension of the OAuth 2.0 in that it adds an identity layer. This is achieved by abstracting an ID token from this layer using JWT tokens. The Identifier (ID) token then represents the user's identity, which is used for authentication, or in other words, proving the user's identity, since the OAuth 2.0 itself is used for authorization, which involves what the user can do.

The client receives an ID token along with the access token. The ID token is retrieved thanks to the OP mentioned at the beginning of this subsection. The client then sends this token with each request to appropriate sources, which use it to perform necessary actions to determine whether the user is allowed to access the resource.

This approach with OIDC and the tokens takes the security aspects to another level, as the applications relying on this protocol do not need to store the user's credentials, and the solution is so-called passwordless. The JWT token contains all the necessary information that applications need to know to decide whether the client is authenticated and authorized for access. This also allows the SSO approach between many services, which means that the user only needs to be logged in once in one place and can access multiple services, reducing the pressure on users to remember their passwords for multiple applications, but they can be authenticated from one place because each service handles all the processes (authentication and authorization) thanks to the tokens. This process is illustrated in the picture **4.10**. Not only the theoretical illustration has been done in this thesis, but the OIDC protocol has also been implemented in the practical part of this thesis, which is illustrated in the **section 5.5.1**.

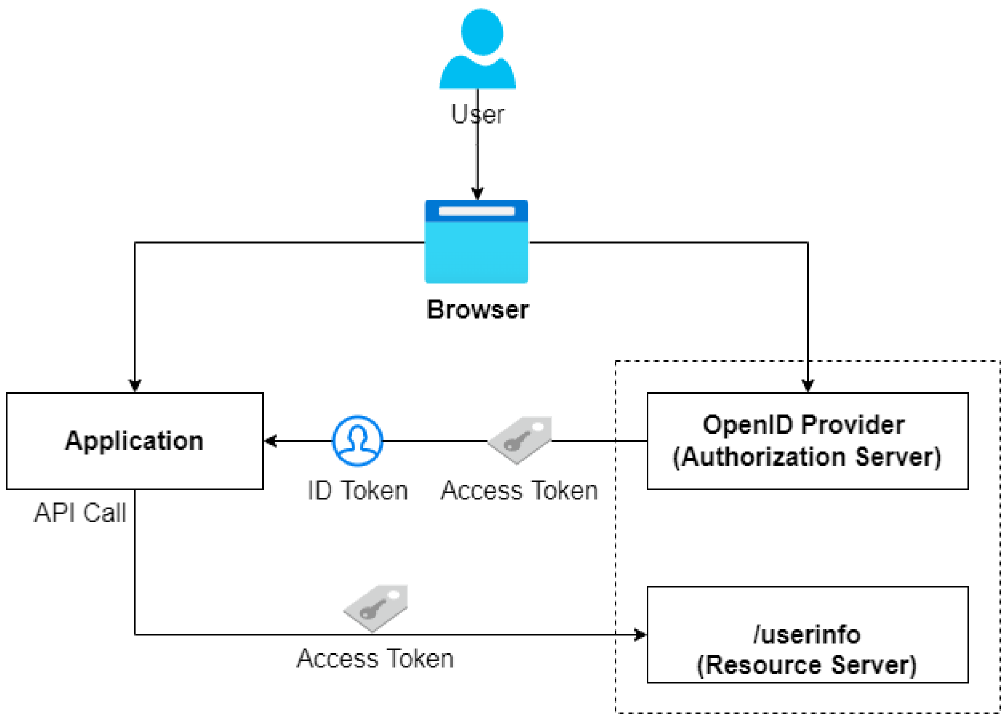


Figure 4.10: OIDC Access and ID Tokens in API calls[49]

4.5.4 Security Assertion Makrup Language (SAML) 2.0

[50] SAML 2.0 is an Extensible Markup Language (XML)-based standard for exchanging authentication and authorization data between parties. Before diving into the description of this protocol in more details, it is important to mention the basic actors that play pivotal role in this approach:

- **IDP**: SAML 2.0 IDP is an authority used to verify a user's identity. The IDP typically provides public endpoints to obtain necessary data to mediate the authentication flow. The IDP is the same as defined in the OIDC section (if allows the SAML 2.0 authentication).

- **Service Provider (SP)**: A typical example of a SP used in the SAML 2.0 based authentication may be `ShibbolethSP`. Service provider is most commonly a component, or a middleware in a web application that triggers the authentication requests to a SAML 2.0 IDP.

The authentication and authorization mediated between IDP and SP is done by exchanging the digitally signed XML documents, in this case SAML 2.0 Assertions. The whole process may be illustrated by the **picture 4.11**. The SAML 2.0 response exchanged between IDP and SP may be illustrated like this (shortened version):

```
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" ID="_8e8dc5f69a98cc4c1ff3427e5ce34606fd672f91e6" Version="2.0" IssueInstant="2014-07-17T01:01:48Z" Destination="http://sp.example.com/demo1/index.php?acs" ...>
  <saml:Issuer>http://idp.example.com/metadata.php</saml:Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
  </samlp:Status>
  <saml:EncryptedAssertion>
    <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" Type="http://www.w3.org/2001/04/xmlenc#Element"><xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/><dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"><xenc:EncryptedKey><xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/><xenc:CipherData><xenc:CipherValue>...</xenc:CipherValue></xenc:CipherData></xenc:EncryptedKey></dsig:KeyInfo>
   <xenc:CipherData>
      <xenc:CipherValue>...</xenc:CipherValue>
   </xenc:CipherData>
</xenc:EncryptedData>
  </saml:EncryptedAssertion>
</samlp:Response>
}
```

Listing 4.2: SAML Response

The IDP provides an endpoint that contains all the necessary information for the SP. In this case, an example using IAM Keycloak, which is used in the practical part of this thesis. The endpoint is located on this URI:

`http://<domain>/auth/realms/<realm>/protocol/saml/descriptor`

After calling this endpoint, all other necessary endpoints are described within this saml descriptor. The response is also XML based and looks like this

```
This XML file does not appear to have any style information
    associated with it. The document tree is shown below.
<md:EntityDescriptor xmlns="urn:oasis:names:tc:SAML:2.0
    :metadata" xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata"
     xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#" entityID="
    http://localhost:9080/auth/realms/SAMLRealm">
<md:IDPSSODescriptor WantAuthnRequestsSigned="true"
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0
    :protocol">
<md:KeyDescriptor use="signing">
<ds:KeyInfo>
<ds:KeyName>a2Z9TtLbPk6TfGVR7e-5hI0TcQfh7UPFUC5nCz8bZ6E</
    ds:KeyName>
<ds:X509Data>
<ds:X509Certificate>...</ds:X509Certificate>
</ds:X509Data>
</ds:KeyInfo>
</md:KeyDescriptor>
<md:ArtifactResolutionService Binding="
    urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http:
    //localhost:9080/auth/realms/SAMLRealm/protocol/saml/
    resolve" index="0"/>
<md:SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0
    :bindings:HTTP-POST" Location="http://localhost:9080/auth/
    realms/SAMLRealm/protocol/saml"/>
<md:SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0
    :bindings:HTTP-Redirect" Location="http://localhost:9080/
    auth/realms/SAMLRealm/protocol/saml"/>
<md:SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0
    :bindings:HTTP-Artifact" Location="http://localhost:9080/
    auth/realms/SAMLRealm/protocol/saml"/>
<md:SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0
    :bindings:SOAP" Location="http://localhost:9080/auth/
    realms/SAMLRealm/protocol/saml"/>
<md:NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
    format:persistent</md:NameIDFormat>
<md:NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-
    format:transient</md:NameIDFormat>
```

```
<md:NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
   format:unspecified</md:NameIDFormat>
<md:NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-
   format:emailAddress</md:NameIDFormat>
<md:SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0
   :bindings:HTTP-POST" Location="http://localhost:9080/auth/
   realms/SAMLRealm/protocol/saml"/>
<md:SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0
   :bindings:HTTP-Redirect" Location="http://localhost:9080/
   auth/realms/SAMLRealm/protocol/saml"/>
<md:SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0
   :bindings:SOAP" Location="http://localhost:9080/auth/
   realms/SAMLRealm/protocol/saml"/>
<md:SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0
   :bindings:HTTP-Artifact" Location="http://localhost:9080/
   auth/realms/SAMLRealm/protocol/saml"/>
</md:IDPSSODescriptor>
</md:EntityDescriptor>
}
```
Listing 4.3: SAML Response

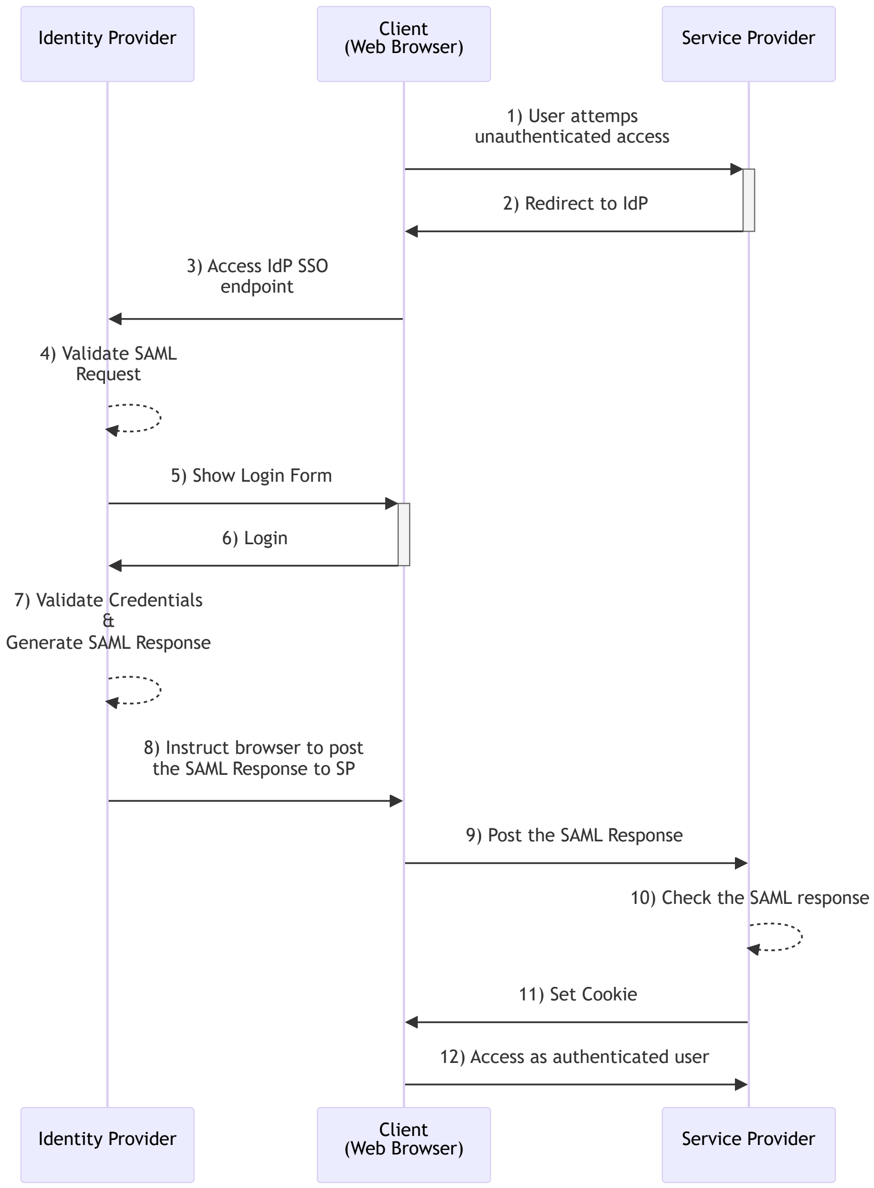

Identity Provider
Client (Web Browser)
Service Provider
1) User attemps unauthenticated access
2) Redirect to IdP
3) Access IdP SSO endpoint
4) Validate SAML Request
5) Show Login Form
6) Login
7) Validate Credentials & Generate SAML Response
8) Instruct browser to post the SAML Response to SP
9) Post the SAML Response
10) Check the SAML response
11) Set Cookie
12) Access as authenticated user
Identity Provider
Client (Web Browser)
Service Provider

Figure 4.11: SAML authentication flow[50]

More detail on how this protocol works from a practical perspective and how it can be configured is illustrated in the practical part of this thesis, in the **section 5.5.1.**

4.5.5 Kerberos

[51] Kerberos is a network authentication protocol designed to provide strong authentication for client/server applications by using secret key cryptography. It is primarily used in secure network environments where identity

verification is crucial, such as corporate intranets and secure web applications. One of the technological examples where Kerberos takes its part is Microsoft Active Directory that employs Kerberos for authentication across Windows networks.

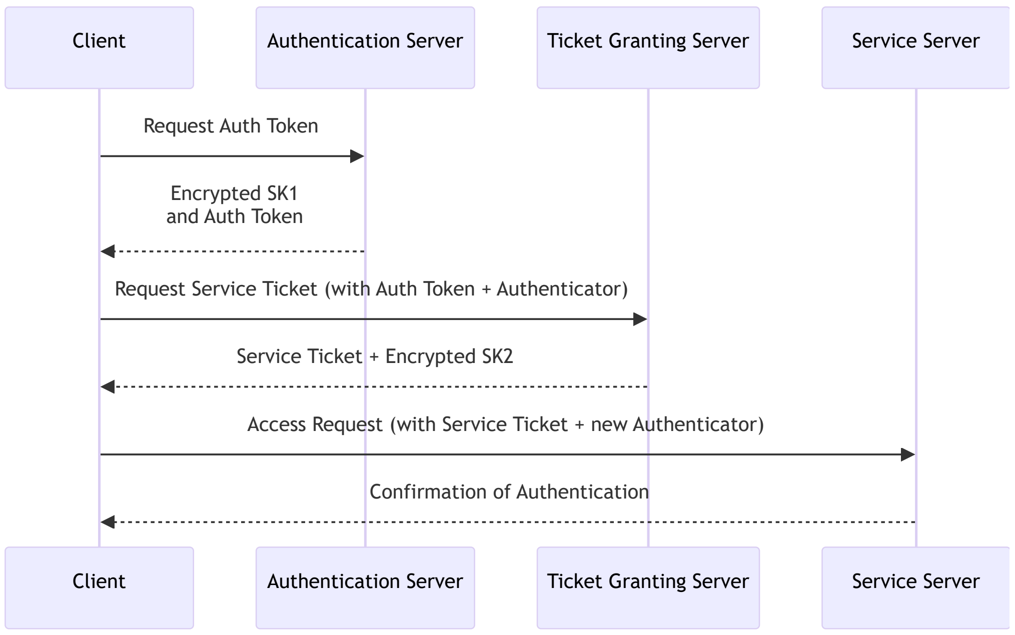

Client
Authentication Server
Ticket Granting Server
Service Server
Request Auth Token
Encrypted SK1 and Auth Token
Request Service Ticket (with Auth Token + Authenticator)
Service Ticket + Encrypted SK2
Access Request (with Service Ticket + new Authenticator)
Confirmation of Authentication
Client
Authentication Server
Ticket Granting Server
Service Server

Figure 4.12: Simplified Kerberos Authentication Flow

Advantages

- **Enhanced Security**: Encrypts data to protect against eavesdropping and replay attacks.
- **Single Sign-On (SSO)**: Allows users to log in once and access multiple services without re-authenticating.
- **Scalability**: Well-suited for large organizations with extensive network security requirements.

Disadvantages

- **Complexity**: Implementation and management can be complex and resource-intensive.
- **Dependency**: Relies on a central authentication server, creating a single point of failure.

- **Limited Cross-Domain Capability**: Cross-realm authentication can be challenging to configure.

4.5.6 LDAP in Cloud Computing Environments

[52] LDAP serves as a cornerstone in managing directory services across IP networks, facilitating efficient user and resource management. This section explores LDAP's functionality, its critical role in cloud computing, particularly in Azure, and delves into inherent security risks and mitigation strategies.

Overview of LDAP

LDAP, a streamlined version of the Directory Access Protocol (DAP), is pivotal for querying and modifying directory services via an open-source application protocol. Its compatibility with both public and private networks enhances its utility across diverse directory services, making it indispensable for accessing and authenticating user information in a multitude of environments.

LDAP and Cloud Computing

The integration of LDAP with cloud services, such as Microsoft's Azure, exemplifies the protocol's versatility. LDAP functions as the communicative protocol for Azure Active Directory, enabling seamless authentication and access management in cloud infrastructures. This synergy underscores the protocol's adaptability in bridging on-premises directory services with cloud-based environments.

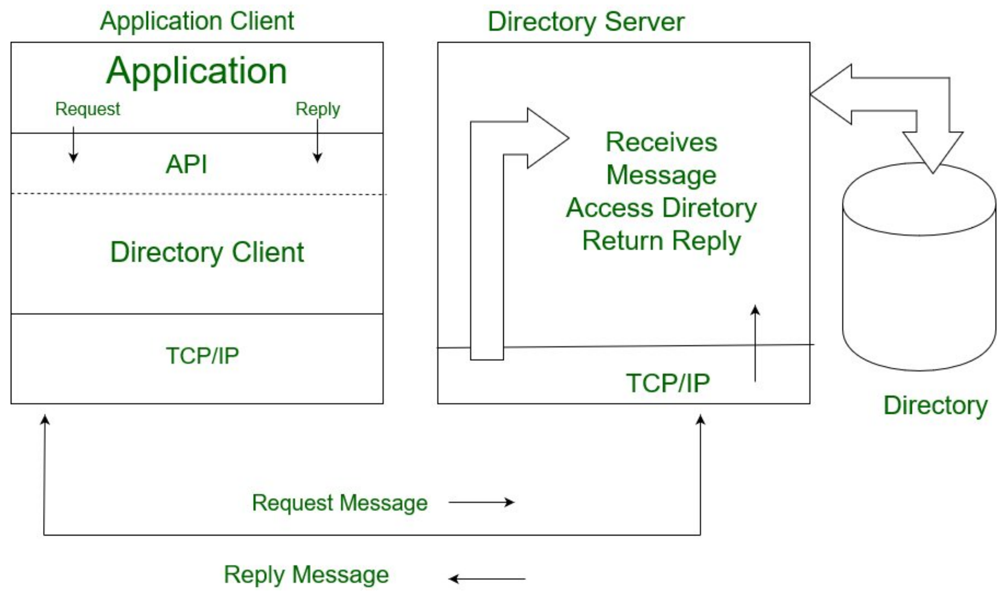


Figure 4.13: LDAP Workflow[53]

4.6 Trust Management

Trust Management plays a crucial role in enhancing the security of distributed systems within cloud computing environments. It is particularly relevant when systems incorporate services from third parties or manage access to sensitive data and functionalities that must be protected against unauthorized or malicious access. Trust Management System automates the process of validating access across varied services with potentially different access requirements.

Originally introduced by Matt Blaze, Trust Management aids in the automated verification of actions against established security policies. It assesses whether actions comply with policies based on presented credentials, regardless of the entity's actual identity. This approach decouples the symbolic representation of trust from the actual identity, focusing on aspects such as honesty, truthfulness, competence, and reliability [54], [55].

Key principles of Trust Management include:

- Formulating and managing security policies and credentials.
- Determining if a specific combination of credentials satisfies relevant policies.
- Deferring trust to third-party entities.

- Dynamically adjusting trust levels based on behavior or other factors, with the possibility of increasing or decreasing trust based on observed actions.

Trust can be illustrated by analogy with ticketing systems, such as those used in Kerberos (**section 4.5.5**), where a ticket grants entry or access rights, embodying distributed trust. However, systems may reject a ticket if further verification is required, demonstrating the need for adaptive trust levels.

4.6.1 Trust Management Technologies

Trust Management technologies are integral to the effective implementation of the above principles within information systems. These technologies include for example:

- **Kerberos**: As mentioned in the **section 4.5.5**.
- **JWT**: As mentioned in the **section 4.5.1**, JWT plays a pivotal role in the TM in general, as the JWT may be used across the systems to evaluate whether the operation is allowed or fobidden.
- **RBA**: As mentioned in the **section 4.4.8**.
- **RBAC (Role-Based Access Control)**: This is more of a process than a technology, but this approach also plays a central role in the TM topic, as access to resources is automatically maintained by the roles of each user trying to access them. This approach has been also illustrated in the practical part of this thesis **5.5.2**.

4.7 Further Security Practices

As mentioned in the previous section discussing the IAM and IAM technologies, it is also important to mention another security aspect that must be considered when developing any application that would be available to users online. The first and very important aspect, which can help prevent some dangerous attacks that can lead to data breaches (but not only), can be prevented by proper input validation. Nevertheless, this section also mentions some of the most common attacks that occur daily to many applications that are accessible online, especially larger systems and applications in general.

4.7.1 Input Validation

It is widely recognized that any application allowing user input must implement stringent input validation. Specifically, techniques from extreme programming prove invaluable in this context. Users should never be fully trusted, as even minor vulnerabilities can lead to significant system failures, primarily due to attackers exploiting every small oversight in the application.

Proper input validation is crucial for preventing attacks such as [56]:

- SQL Injection,
- Cross-Site Scripting (XSS): This attack can lead to the compromise of redirect links used for example in the OIDC protocol (but not only). Therefore, those IAM systems that mediate this protocol, and are used for SSO approach, explicitly require a list of allowed URLs that are valid to increase the security of the user that would use the specific application, most likely the web application.
- In today's digital landscape, even DOM-based XSS attacks[7],
- Denial of Service (DoS), which can occur, for instance, by uploading malicious files. Such files might overwrite crucial server files (e.g., .htaccess) or exhaust server storage, severely impairing server availability.

The examples listed above highlight just a few of the many potential issues associated with inadequate input validation. The validation process must be tailored to the type of data being submitted, encompassing files, text, numbers, and any other conceivable input types. While it's impractical to catalog all issues related to input validation here, it's essential to consult resources like the OWASP Foundation's cheat sheets for comprehensive guidelines. This mention serves as a reminder that input validation is a critical security aspect in the development of any application open to public interaction, especially web applications engaging users through various input forms.

4.7.2 Common Attacks on the Internet

As each application is unique, employing its own or external libraries, contributing to the complexity of securing it. Importantly, threats to system

[7]For more details about preventing DOM-based XSS, refer to the OWASP Cheat Sheet: `https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html`

stability are not solely external; incorrect input validation, as highlighted in **section 4.7.1**, or unforeseen failures, such as runtime exceptions, can also lead to service denial.

The development of robust and secure applications highly depends on the specific use case. It is acknowledged that larger systems cannot remain without any vulnerabilities; no system is entirely bulletproof. Therefore, the security measures prioritized will vary based on system functionality. For instance, systems focusing on data storage and analysis prioritize safeguarding data against breaches or loss. Systems for streaming video content, among others, also have specific security needs.

In all cases, a fundamental understanding of good programming paradigms and patterns that mitigate risks is crucial. Knowledge in areas such as authentication, authorization, encryption, data integrity checks, digital signatures, certificates, and input validation is indispensable for securing applications. With new vulnerabilities discovered weekly in various libraries, staying informed through technical cybersecurity news, updating systems regularly, checking for library vulnerabilities, monitoring for malicious activities, establishing static connections through Media Access Control (MAC) or IP address restrictions, and more, becomes imperative. Moreover, utilizing established security frameworks, like Spring Security in Java, is advisable over custom-built security solutions, unless one possesses extensive security expertise and is involved in developing advanced security tools.

This master's thesis does not aim to transform readers into cybersecurity experts. Instead, it seeks to outline fundamental security principles and remind readers that cybersecurity requires staying informed through reputable sources such as the OWASP Foundation, NIST, and others. Continual education, attending conferences, and adhering to the latest recommendations and guidelines are all part of maintaining a strong security posture.

4.7.3 Common Security Risks and Attacks

This section outlines prevalent security risks and attacks, recognized widely both in online sources and cybersecurity literature. The focus here is not to exhaustively list all potential security threats but to highlight common ones identified by reputable sources, including cybersecurity and web application security books.

Cloudflare[8] enumerates several widespread security risks, underscoring the varied nature of threats facing web applications today:

[8]"Cloudflare operates one of the largest networks on the Internet, providing services that enhance the security and performance of websites and web services [57]."

- Zero-Day Vulnerabilities,
- XSS,
- SQL Injection,
- DoS/Distributed Denial of Service (DDoS) Attacks,
- Memory Corruption,
- Buffer Overflow,
- Cross-Site Request Forgery (CSRF),
- Credential Stuffing,
- Page Scraping,
- API Abuse,
- Shadow APIs,
- Third-Party Code Abuse,
- Attack Surface Misconfigurations.

This compilation serves as an overview of frequent challenges encountered every day by developers, particularly within large-scale systems. It is crucial to acknowledge that this list is not exhaustive. Cybersecurity professionals continually confront emerging threats not specified here, necessitating ongoing vigilance and innovative preventive strategies.

5 Practical Implementation

5.1 Introduction of the Applications

This Master's thesis is primarily analytical, delving into cybersecurity recommendations, trends, and essential concepts pertinent to the subject matter. To complement the theoretical analysis, a demonstrative application has been developed to showcase practical implementations of selected authentication protocols, authorization approaches, and encryption and hashing algorithms within specific use cases. This application serves as a bridge between theoretical concepts and real-world application, particularly within the realms of cloud computing and microservice architectures.

A full-stack real-time chat application was chosen as the demonstration platform. This application, developed using the ReactJS framework, reflects current trends in web application development, favoring ReactJS or Angular for their robust capabilities. The real-time functionality is achieved through a websocket connection to a broker, implemented in the Spring Boot framework and utilizing STOMP for communication. Additionally, a core application, also crafted in Spring Boot, plays a pivotal role in authentication and authorization. This application acts as a pseudo API gateway, facilitating direct communication with Keycloak and each service, thereby ensuring a comprehensive demonstration of both websocket and Representational State Transfer Application Programming Interface (REST API) secured endpoints for basic CRUD operations.

The selection of authentication protocols for demonstration purposes includes:

- Custom Username & Password management with JWT Bearer token integration in the Core application.
- OIDC Authentication Protocol, managed by Keycloak, with communications routed through the Core and ReactJS applications.
- SAML 2.0 Authentication Protocol, demonstrated using a Shibboleth module in an Apache Server to illustrate SSO capabilities, also under the management of Keycloak but not utilizing the Core application. This setup represents a distinct service within the entire infrastructure, aimed solely at demonstrating the SAML 2.0 protocol.

Furthermore, the application embodies specific authorization approaches, simultaneously demonstrating encryption and hashing techniques through message signatures that incorporate both symmetric and asymmetric encryption methods.

The development of this demonstration application was also motivated by the desire to underscore the significance of penetration tools. These tools are invaluable not only for integration into automated Continuous Integration/Continuous Delivery (CI/CD) pipelines but also for identifying vulnerabilities and performance bottlenecks. For this thesis, the ZAP proxy was primarily utilized to highlight these aspects.

Each service and technology is discussed individually in separate sections, with details of the approaches used and specific references to the theoretical part of this thesis, so that the correlation between theory and practical use can be seen, with its pros and cons. It is also important to mention that this is only a demo application, so it could not be used in a real production. Some implementation approaches are really a proof of concept and their real use would suggest a more complex implementation with additional steps in the code for specific use cases.

With a successful proof of concept of the OIDC Authentication Protocol, an integration of this protocol has also been realised in a real project called SPADe[1], which also consists of a core application and a ReactJS client that is used for users so that they can use the application with a given user interface. The Keycloak service manages this protocol and the implementation is the same as in the demo application.

[1]More information about what SPADe is can be found here: `https://dspace5.zcu.cz/bitstream/11025/49934/1/A19B0615P_BP.pdf`

5.2 Design and Architecture

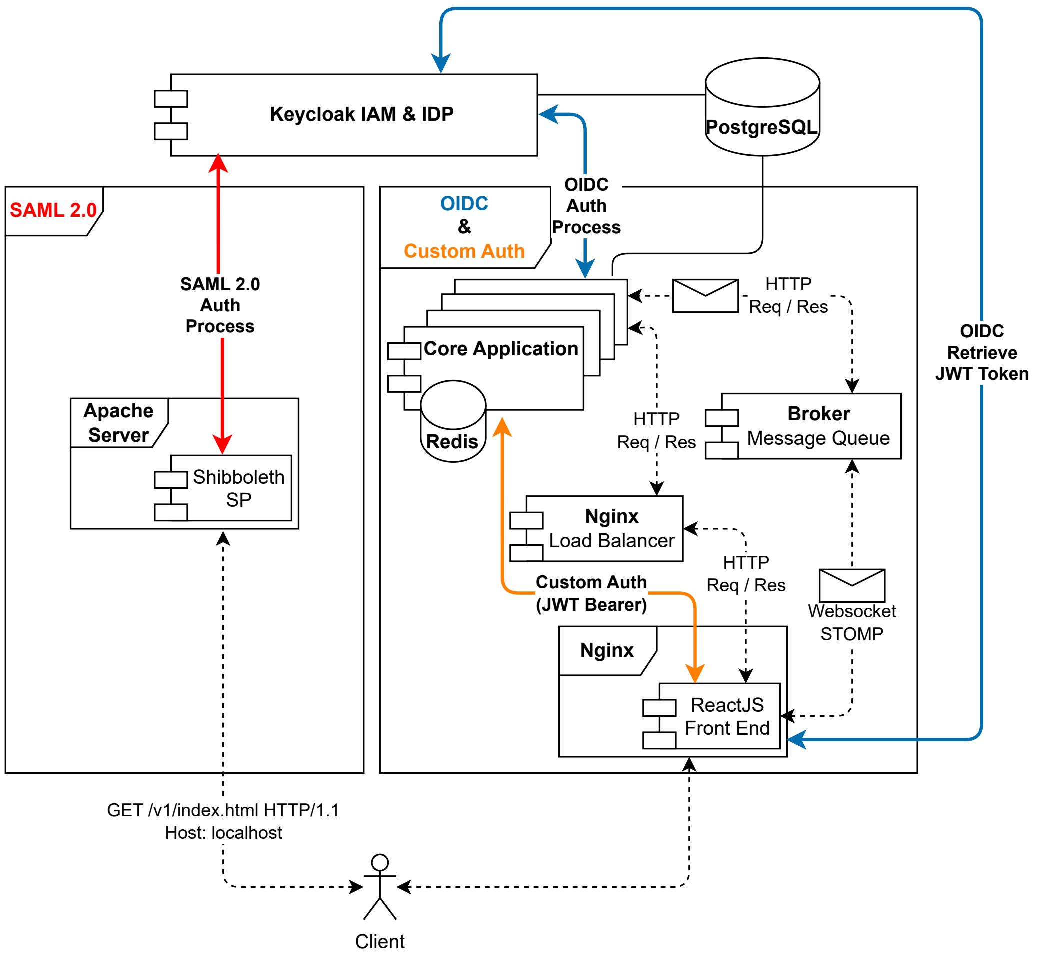


Figure 5.1: Illustrated infrastructure

5.3 Brief Description of Components

5.3.1 Docker

For the demonstration application, Docker[2] technology, particularly docker-compose, has been utilized to containerize and streamline the development workflow. This approach not only simplifies the development process but also supports the horizontal scaling of the core application, a topic that will be elaborated upon later in this document.

[2]For more information on Docker technology, refer to the official documentation: `https://docs.docker.com/desktop/`

5.3.2 Keycloak

Keycloak is a free tool developed by Red Hat that helps with managing identities and access for applications and services, it can be said that Keycloak is a whole IAM tool, like mentioned in the chapter **4.4**. It is built to handle a range of security tasks, like logging in with one set of credentials SSO, connecting with different identity systems (identity brokering) such as: AWS Cognito, Azure Active Directory, and even social media like Google and Facebook. It works well with popular authentication ways, such as OIDC and SAML 2.0 [58]. Keycloak is also up-to-date with the latest security standards and ways of doing things.

For the demo application in this master's thesis, Keycloak (version 22.0.1) was chosen to showcase the application of widely used authentication protocols, OIDC and SAML 2.0. The aim was not to build these protocols from scratch but to demonstrate their practical application and verify their security effectiveness in a cloud computing environment. This example illustrates how developers can seamlessly integrate sophisticated authentication mechanisms into applications, leveraging tools like the Spring Security framework, without reinventing the wheel and potentially introducing vulnerabilities. The principles demonstrated here are applicable across various programming languages, including C#, Python, and more.

5.3.3 Overview of the Core Application

The Core Application is the backbone of the real-time chat system, built using Java and the Spring Boot framework for handling HTTP requests and secured with Spring Security. This application is not limited to CRUD operations but also incorporates default Spring Security for authentication and authorization.

Keycloak, an external IAM solution, is integrated to demonstrate advanced authentication mechanisms beyond basic username and password strategies. This integration showcases the use of OIDC for authenticating users, with Keycloak managing the authentication process and issuing JWT tokens containing user credentials.

The application's architecture supports horizontal scaling, optimizing performance and user experience. It serves as a central hub for service requests, including message signature verification and user authentication, with specific actions varying based on the target URI. Security operations such as hashing and encryption are standard practices within the application, aimed at safeguarding data integrity and confidentiality.

For an in-depth exploration of specific security approaches, separate sections are dedicated to discussing their application within this core system, supplemented by use cases and detailed analyses. The figure below offers a visual representation of how Spring Security manages authentication across different endpoints:

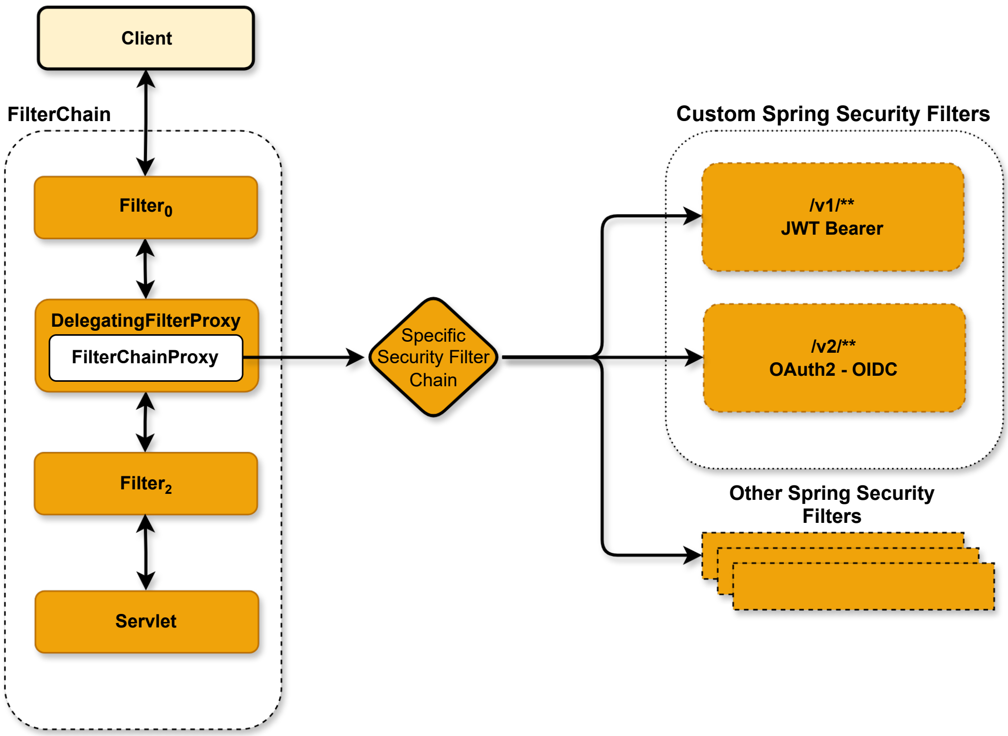


Figure 5.2: Flow of Security Filter Chains

Endpoint Overview The core application encompasses several endpoints, each designed to fulfill distinct functionalities within the chat system. A summarized description of these endpoints is provided in the tables below, outlining their purposes and access methods.

Endpoint	Method	Description
/v1/auth/signup	POST	Register a new user.
/v1/auth/login	POST	Authenticate a user.
/v1/auth/validate	POST	Validate the JWT token.
/v1/chat/save	POST	Save a chat message, requiring authorization.
/v1/auth/csrf	GET	Retrieve a CSRF token.
/v1/unAuthorized	GET	Respond to unauthorized requests.

Table 5.1: Protected with own JWT Bearer mechanism - /v1/** Endpoints

Endpoint	Method	Description
/v2/unAuthorized	GET	Testing endpoint.
/v2/public-chats	GET	Retrieve public chat conversations.
/v2/private-chats	GET	Retrieve private chat conversation.

Table 5.2: Protected with Keycloak OAuth2 OIDC protocol - /v2/** Endpoints

5.3.4 Simple Broker

The Simple Broker application represents a pivotal component within a Java Spring Boot framework, specifically designed to facilitate real-time, bi-directional communication between clients and servers through websockets. This application acts as a messaging intermediary, routing messages based on predefined topics to the Core application mentioned in the **section 5.3.3**.

Configuration and Functionality

The Simple Message Broker within this application is adeptly configured to manage messages directed to topics prefixed with '/all' and '/user', effectively enabling targeted communication. The main STOMP endpoint, accessible at '/ws', is meticulously set up to accept connections exclusively from 'http://localhost:3000', catering to a ReactJS client application with SockJS fallback options for broader compatibility.

Real-Time Communication and Authentication: This broker's architecture underpins the seamless, real-time exchange of messages, supplemented by a authentication mechanism. Utilizing JWT tokens within message headers, the application ensures that each communication client is authenticated. The authentication is mediated via the Core Application.

Important to mention: The broker primarily functions as a message router, directing communications to the core application, which then handles all authentication and authorization tasks. The broker's role is largely to facilitate message flow between clients and the core application, disconnecting clients if responses from the core application are anything other than a 200 HTTP status, indicating success. Two notable exceptions exist to the broker's routine operation. Firstly, during initial connection setup, user authentication is required for connection to the broker, although this process too is managed by the core application through a custom Websocket Authentication Interceptor class. Secondly, the broker tracks currently active chat users, enabling visibility of active participants within the chat application.

While the authentication mechanisms in the demo application are adequate for demonstration purposes, they fall short of the complexities required for real-world production environments. Beyond securing STOMP endpoints, a production-level system must regulate subscription access to these endpoints, employ strategies to monitor data flow and prevent suspicious activities, and ensure proper management of real-time communication events, such as recovery mechanisms. These enhancements are crucial for maintaining system integrity and user security.

5.3.5 ReactJS Application for User Interface

In line with the current trend of full-stack development, this master's thesis includes a ReactJS application designed to illustrate the application of security approaches discussed in the theoretical part. This application enables users to engage in a real-time chat, allowing both public messages and private messaging among online users. Authentication within this application is implemented via two methods:

1. A custom JWT Bearer token, generated server-side for user authentication and authorization, is utilized following a traditional username and password login. This process is facilitated through a login and signup form, depicted in the images: **B.9, B.10**. A graphical illustration of the data flow for this authentication method is detailed in the images later in this document: **5.4**, **5.5**.

2. OIDC authentication through Keycloak, which serves both as a resource and identity provider. In this setup, user management and

authentication processes are delegated to Keycloak. A graphical representation of the OIDC is shown in this figure: **5.6**.

However, securing applications developed with frontend technologies like ReactJS poses significant challenges. These frameworks, while powerful, rely extensively on third-party libraries, which can introduce vulnerabilities.

To mitigate these risks, the application underwent rigorous penetration testing using tools selected for their effectiveness in identifying potential security weaknesses. Remediation efforts for identified vulnerabilities were guided by OWASP and NIST principles to enhance the application's security posture.

The application features multiple user interfaces, each designed for specific functionalities. Images of the application's Graphical User Interface (GUI) are provided in the attachment **section B**. Below is a list of the application's pages along with brief descriptions and corresponding links to their images in the attachment section:

- **Main Page**: Serves as the application's landing page, providing general information without requiring user authentication **B.1**.
- **Signature Test**: An unauthenticated tab designed to demonstrate the workings of two digital signature algorithms, RSASSA and HMAC, selected for their relevance to this thesis **B.2, B.3**.
- **Chat Room**: Accessible post-authentication, this tab allows logged-in users to join and participate in chat discussions. Authentication is facilitated through a custom JWT token mechanism, securing communications with the server **B.4**.
- **Chat Settings**: Protected by OIDC authentication, this tab redirects unauthenticated users to the Keycloak login page. Upon successful authentication, users will be presented with a chat settings dashboard. This page demonstrates the use of `react-oidc-context`, a specialised JavaScript library that simplifies the integration of OIDC authentication, and shows the advantages of using well-supported external libraries over custom-built solutions, as well as some authorisation approaches, such as RBA, as not all users authenticated via Keycloak can see the stored chat history, as it requires a special role admin **B.6, B.7, B.8**.

5.3.6 Databases

The architecture of the application leverages two primary databases to handle data persistence, user management, messaging functionality, and security mechanisms. These databases are PostgreSQL and Redis, each serving distinct but crucial roles in the system's overall functionality.

PostgreSQL serves as the backbone for both Keycloak and the core application, providing a robust and reliable storage solution. In the realm of Keycloak, PostgreSQL stores identity and access management data, enabling efficient authentication processes. For the core application, PostgreSQL is utilized for two main purposes:

- **User Management:** The database is used for storing user credentials, including hash of a password.
- **Messaging Storage:** Leveraging its reliability and scalability, PostgreSQL is used for storing messages from the real-time chat feature offered by the ReactJS application. This includes both public and private messages.

Redis, known as a key-value database, is specifically employed for CSRF token storage. This choice is particularly relevant in scenarios where the core application undergoes horizontal scaling. By using Redis to manage CSRF tokens, the application ensures that security tokens are consistently available across multiple instances of the back-end application. This design choice is pivotal in maintaining the security integrity of the application, especially in distributed deployment scenarios where load balancing and session consistency are essential.

5.3.7 Nginx

In the application's infrastructure, Nginx plays a pivotal role, with two distinct servers set up to optimize performance and scalability. Each Nginx server is configured to fulfill specific responsibilities, ensuring that the web application operates smoothly and efficiently for the demonstration purposes of this Master's thesis.

Nginx as a Web Server for ReactJS: The first Nginx server is dedicated to serving the ReactJS application. This server acts as a web server, hosting the static files generated from the ReactJS build process.

Nginx as a Load Balancer for the Core Application: The second Nginx server is configured as a load balancer. This setup is crucial for the horizontal scaling of the core application, as it distributes incoming network traffic across multiple instances of the core application. By doing so, the load balancer ensures that no single instance is overwhelmed, thereby increasing the application's availability and reliability. The High Availability (HA) feature of this Nginx server plays a vital role in maintaining uninterrupted service, even in the face of high user demand or potential instance failures.

The strategic deployment of these two Nginx servers, one serving as a dedicated web server for the ReactJS application and the other as a load balancer for the core application, demonstrates a well-planned approach to leveraging Nginx's capabilities. This dual-server setup not only optimizes the delivery of web content but also ensures the scalable and resilient operation of the core application, marking a significant step towards achieving a robust and high-performing web infrastructure.

5.3.8 Apache Web Server

This demonstrational application uses the Apache Web Server to host a single HTML resource, which displays simple static text. The server employs the Shibboleth module for sophisticated authentication and authorization, enhancing the web services provided by this Apache server. This module enables SSO capabilities and the secure exchange of web resources among various institutions. Here, the primary purpose is to demonstrate the SAML 2.0 authentication approach, with Keycloak serving as the resource provider. If a user attempts to access the Apache server's resource without being logged in, they are presented with Keycloak's login form. Successful authentication allows the user access to the resource. It is worth noting that this demonstration, while basic, hints at the complex integration possibilities between the Apache server and the Shibboleth module. The configuration of the Shibboleth identity provider is detailed in an XML file named shibboleth2.xml, which includes definitions for accessing specific resources on specific locations. This file is included in the demonstration application's zip archive. However, this is merely an illustration of the SSO method using Shibboleth, Keycloak, and the Apache server for resource protection with SAML 2.0 approach, and not a comprehensive guide for securing applications with the SAML 2.0 protocol. Despite its effectiveness for this demonstration, this approach is more complex for larger systems to set up in a real production environment, and the attached configuration does not meet all necessary requirements and it demands more extensive knowledge to use it properly.

Important Note on the Demonstration Approach

The main objective of this thesis was not to develop a complete deployment solution, but to illustrate the use of different authentication protocols and other recommended tactics to secure applications, especially for cloud computing environments. A notable challenge arises from the complexity of Docker technology and the use of the Shibboleth module within the Apache server, which leads to the temporary unavailability of Keycloak as a service provider during the startup of the entire infrastructure, resulting in an error being displayed to a user accessing the resource instead of actually triggering the authentication flow. This problem occurs because the Apache server and the Shibboleth module start up faster than the Keycloak service during the Docker compose-up action. Shibboleth attempts to connect to Keycloak which is not yet available, resulting in errors when attempting to access the protected resource. The solution is to wait until all services are properly started. Then manually restart the apache-saml service in the docker and the problem should be solved.

5.4 Chosen Authentication Protocol For SPADe Project

Following the analysis conducted in the introduction (**5.1**), and informed by comprehensive cybersecurity fundamentals primarily sourced from OWASP, this section details the authentication protocol selection for SPADe. SPADe, at its core, interfaces with external entities exclusively through REST API endpoints. Accompanying this backend functionality, a user-friendly web interface developed with ReactJS technology complements the application. The existing infrastructure and methodologies applied in the demo application facilitated a straightforward integration of the selected authentication protocol into SPADe.

5.4.1 Authentiction and Authorization

Previously, the SPADe offered a custom authentication approach, managing own database (Microsoft SQL Database) with user's credentials on its own. This fact faced multiple problems, such as:

1. The front-end client developed in the ReactJS framework had a problematic solution for storing the JWT token, which was created and managed by the SPADe application itself. Since this was developed

manually without any further analysis, but with a basic guide found on the web, this solution had problems related to refresh tokens and automatic management on the client side, which ended up having many glitches that made this solution not so user friendly.

2. Since the SPADe application was completely stateless from the beginning, the JWT token management was managed by the SPADe application itself to determine whether a user was logged in or not to allow access to resources. In the previous solution, there was a great lack of advanced practices related to the management of JWT tokens. This solution only worked with a few facts passed in the token. Application was vulnerable (not only) to token compromise attack.

3. The role-based access mechanism would be much more difficult to achieve because the previous solution was not designed for roles that would be used throughout the application.

4. The SPADe project used its own management of user credentials. As in the demo application, this solution had some security risks and did not provide any additional benefits for the application itself. Although the application was protected against threats like SQL injection, making it more difficult to steal a user's data through this type of attack, but the developers of this application would have to update the encryption and hashing algorithms used for pre-processing and storing the user's password in the database in case the algorithm was found to be vulnerable to known attacks such as rainbow table attack or brute force attack, since this application had a total lack of risk-based authentication, which is offered by default by the larger IAM tools that manage user authentication on their own by default if properly configured.

5. There is a high probability that the SPADe system will expand into a larger system consisting of more independent services. In this case, some services may require different user roles or perhaps different IAM systems. In this case, Identity Federation would be beneficial because it would allow clients to authenticate in one place and be able to access other systems depending on their privileges. This would not be possible with the previous solution.

The list of problems encountered with the previous authentication, authorization and user management solution in the SPADe project described above may be even longer, as there may be other problems highlighted that were present in the project before the integration of the chosen solution.

Since the demo application included a demonstration of multiple authentication methods (SAML 2.0 mediated by shibbolethSP, OIDC, and custom username/password), the custom approach was automatically rejected as this solution did not bring any benefits except the ease of implementation, and the solution was already implemented in the previous version and had to be improved. The SAML 2.0 protocol provided a very good use case, especially for Identity Federation. However, due to the fact that the web application is written as a single page application (SPA), there would be no advantage compared to OIDC, especially since the SAML 2.0 is much harder to maintain and setup than the OIDC itself, it also does not require any middleware like shibbolethSP to mediate the authentication, so the OIDC was the obvious choice.

The OIDC was mediated by Keycloak IAM. This solution brings many advantages, as this IAM tool not only provides user management, but also allows system administrators to prepare groups and user roles, which can then be used for systems to control access to resources, making the system more robust. The whole system after the integration is illustrated in the **picture 5.3**.

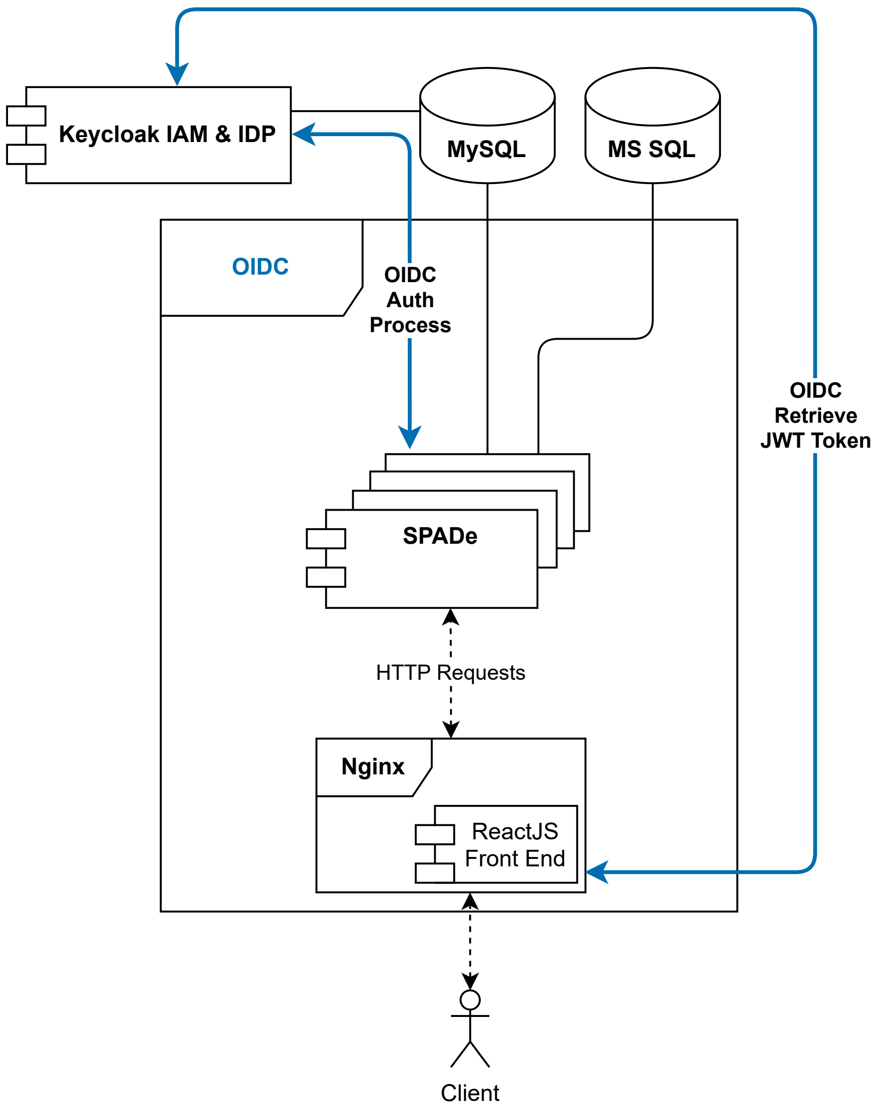


Figure 5.3: SPADe Component Diagram

5.4.2 User Management in the SPADe

With the use of IAM, an entire user management has been delegated to the Keycloak system. This means that the SPADe now allows users who are authenticated by the Keycloak IDP, which is an issuer of the token that is passed to the SPADe REST API with each request. This brings many advantages, such as the fact that the Keycloak can be a central authentication point for the whole SPADe system, so the user would just send the token issued by the Keycloak with each request. Also, since the Keycloak is a complete IAM solution, it offers basic functionalities such as advanced authentication approaches like two-factor authentication, email verification, forgotten password and much more, which was not possible with the previous solution.

However, the database (Microsoft SQL Database) used to store the user's credentials is still used. However, it is not used to store the user's credentials such as password, but to store the user's digital footprint, which allows to store the user's settings and other options offered by the application, so that the user always has his own customized environment, making the application more user-friendly.

5.5 Illustration of Implemented Security Approaches

This section provides implementation details of selected security approaches, in this case: authentication, authorisation, encryption, hashing and user management within a specific use case related to the demo application. For these simple proof of concepts, the advantages and disadvantages are also described, along with possible improvements that might be beneficial, as well as the critical parts that would make the implemented solution impossible to use in real production, and why.

5.5.1 Authentication

For illustration purposes, three authentication approaches are implemented. Those approaches are:

- Custom Username and Password
- OIDC managed by Keycloak
- SAML 2.0 managed by Keycloak

The Custom Username and Password authentication is illustrated like following:

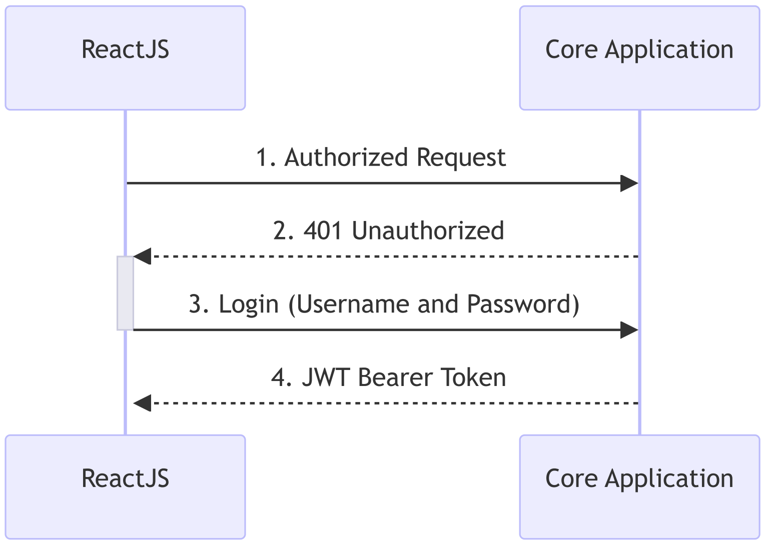


Figure 5.4: Basic Authentication Sequence Diagram

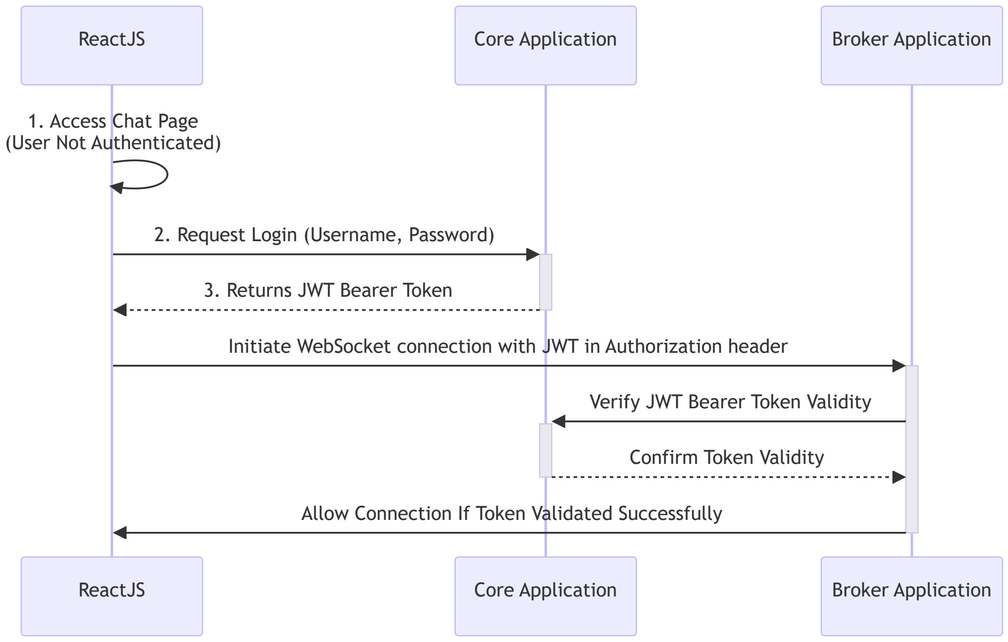

ReactJS
Core Application
Broker Application
1. Access Chat Page
(User Not Authenticated)
2. Request Login (Username, Password)
3. Returns JWT Bearer Token
Initiate WebSocket connection with JWT in Authorization header
Verify JWT Bearer Token Validity
Confirm Token Validity
Allow Connection If Token Validated Successfully
ReactJS
Core Application
Broker Application

Figure 5.5: Broker Authentication

The fundamental approach of using JWT Bearer tokens for session management and authentication, while effective for basic use cases, presents several limitations that can impede scalability and efficiency in more complex applications. Although JWT tokens facilitate stateless authentication by encapsulating user session information, their fixed lifespan poses challenges for maintaining active user sessions without manual intervention for token renewal. This issue underscores the necessity of Refresh tokens in the standard OIDC protocol, which allow for seamless session extension without requiring user re-authentication upon token expiration.

Moreover, the process of validating the JWT token with each request, especially without implementing a caching mechanism, can significantly strain server resources. In scenarios where the authentication server also serves as the core application server, this can lead to system overload, creating potential bottlenecks that affect the overall performance and reliability of the architecture. The continuous need to validate tokens, compounded by the absence of a token refresh strategy, not only increases the risk of server collapse under high load conditions but also detracts from the user experience by necessitating frequent logins.

These limitations highlight the importance of adopting more sophisticated authentication strategies, such as incorporating Refresh tokens and implementing token validation caching, to enhance system scalability, maintain uninterrupted user sessions, and mitigate the risk of server overload. Adopting these strategies ensures a more robust and user-friendly authentication system capable of supporting complex application requirements while maintaining optimal server performance.

The OpenID Connect Authentication illustration:

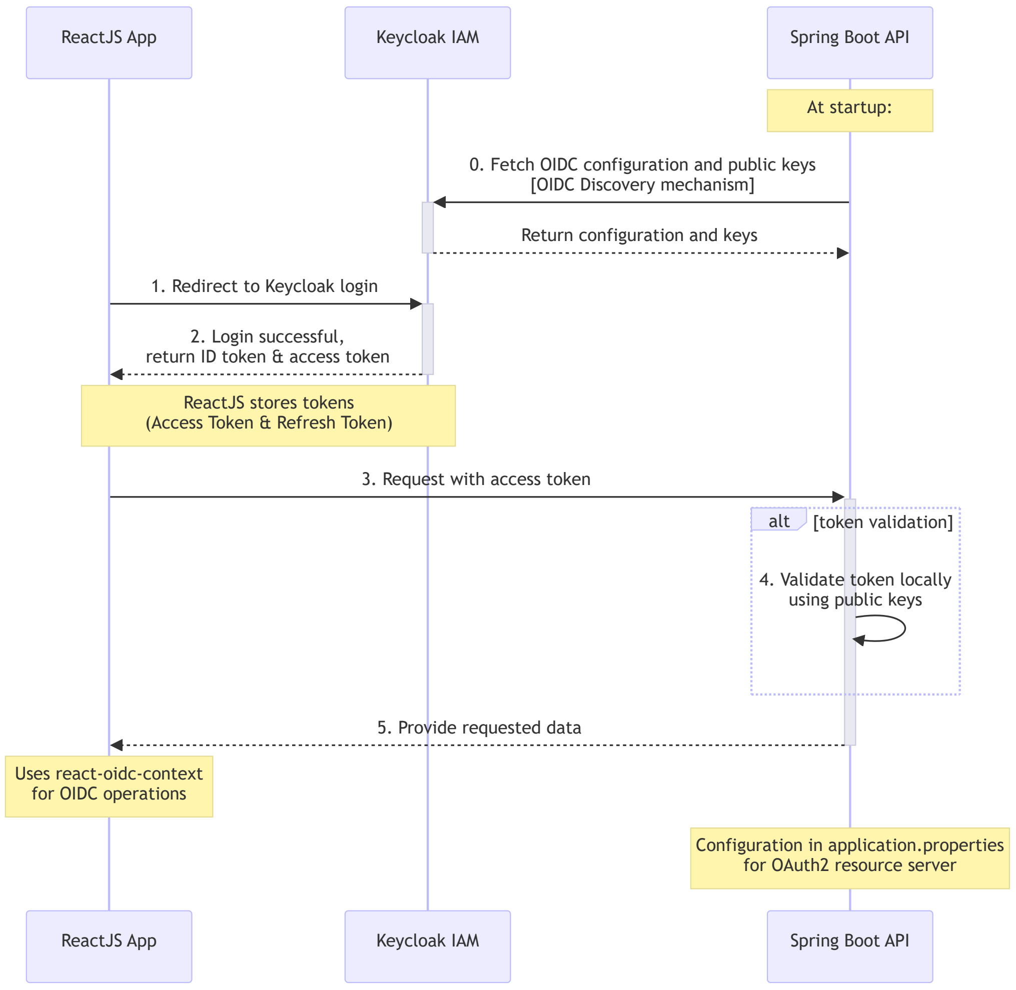

ReactJS App
Keycloak IAM
Spring Boot API
At startup:
0. Fetch OIDC configuration and public keys [OIDC Discovery mechanism]
Return configuration and keys
1. Redirect to Keycloak login
2. Login successful, return ID token & access token
ReactJS stores tokens (Access Token & Refresh Token)
3. Request with access token
alt
[token validation]
4. Validate token locally using public keys
5. Provide requested data
Uses react-oidc-context for OIDC operations
Configuration in application.properties for OAuth2 resource server
ReactJS App
Keycloak IAM
Spring Boot API

Figure 5.6: OIDC Authentication

This simple solution does not have any negative aspects that would prevent it from being used in real production, as this approach already uses a tested solution with a working IAM system, Keycloak. However, this demo contains a security risk due to the absence of the TLS on which the OIDC protocol relies. This layer is not added to the demo infrastructure as automated certificate management in Docker is complex and it would not add anything to this proof of concept of authentication/authorization protocols and how they work other than another layer of security for network traffic within the bridge.

The Security Assertion Markup Language (SAML 2.0) illustration:

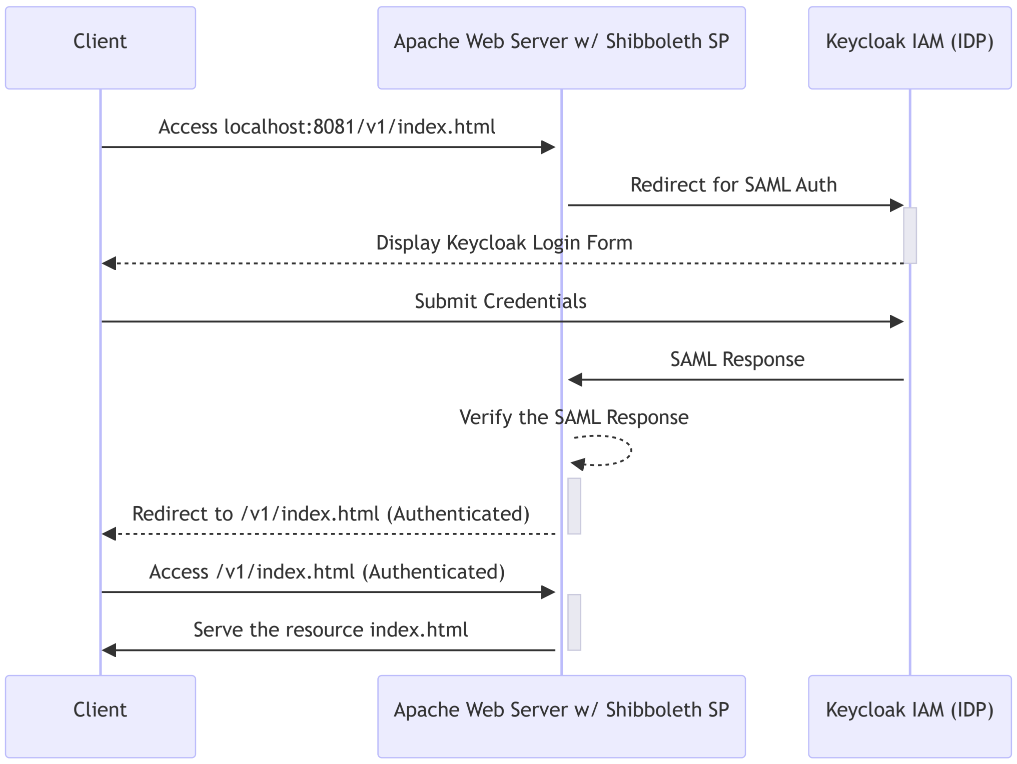

Client
Apache Web Server w/ Shibboleth SP
Keycloak IAM (IDP)
Access localhost:8081/v1/index.html
Redirect for SAML Auth
Display Keycloak Login Form
Submit Credentials
SAML Response
Verify the SAML Response
Redirect to /v1/index.html (Authenticated)
Access /v1/index.html (Authenticated)
Serve the resource index.html
Client
Apache Web Server w/ Shibboleth SP
Keycloak IAM (IDP)

Figure 5.7: SAML Authentication Flow

Although this solution not only works very well as a SSO principle, but is also considered to be a very secure protocol, its configuration can be very complex and not as friendly as the OIDC configuration and the discovery mechanism that allows clients to dynamically obtain the necessary public keys for signature verification. This protocol is used in the demo application mainly to demonstrate the use case and the differences between OIDC and SAML 2.0 approaches. OIDC is more user friendly as it is considered to be lighter and more suitable for web and mobile applications. SAML 2.0 an opposite when talking about SPAs applications (Sundas Coudry [59]). This is one of the reasons why SAML 2.0 was not selected during the analysis as a suitable candidate for integration within the SPADe project.

It is also important to mention that the configuration for Shibboleth SP, as defined in the shibboleth2.xml file, is simplified for demonstration purposes and may not suffice for production environments. Real-world deployment demands a more intricate setup, highlighting a potential drawback of this approach due to the need for comprehensive understanding of Shibboleth SP's integration with the Apache web server.

5.5.2 Authorization

In the theoretical part of this thesis, some authorization approaches are mentioned, such as RBA, which defines access policies based on defined roles. This thesis highlights the application of RBA) through the use of groups and roles managed by Keycloak IAM, particularly within the context of the chat settings page. In the demonstration application, RBA is effectively utilized where all requests targeting `/v2/**` endpoints in the core application are secured by Keycloak through OIDC.

Furthermore, the chat settings page is accessible only to authenticated users. However, to retrieve resources such as the history of private and public chats, users must possess the admin role; otherwise, they receive a 403 Unauthorized response. This ensures that sensitive information is strictly managed and accessible only by users with appropriate authorization levels. In overall, the application uses two roles:

- **basic_user**: Least privilege role that allows the user to access the chat settings page, but does not allow the user to access the chat history. How it then looks is illustrated in the **picture B.7**.
- **admin**: Role that is required to access the history of the chat. How it then looks is illustrated in the **picture B.8**.

5.5.3 Message Signing

Since the theoretical part mentioned the importance of data signing, as shown in the **picture 4.3**, a message signing is chosen to demonstrate how the integrity of data transferred between different applications can be established. This process is illustrated in the following **figure 5.8**:

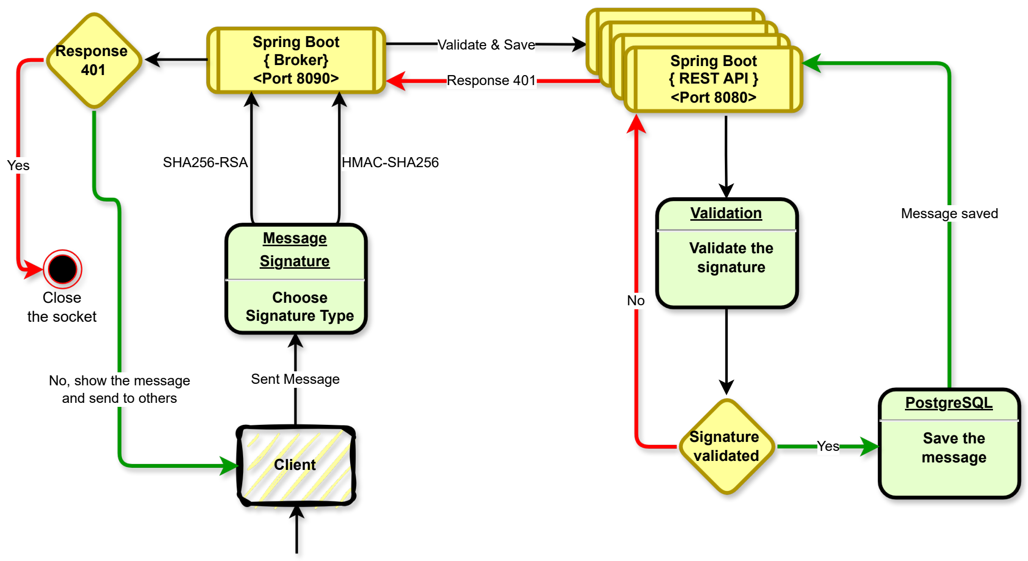


Figure 5.8: Message Authorization Flow

As shown in the figure, a user can choose whether messages created in the chat application are signed using the RSA-SHA256 or HMAC-SHA256 approach (both approaches are described in the next subsection **(5.5.4)**. This functionality works very well and is thoroughly tested with unit tests to ensure that this signing mechanism works well. However, the solution is very primitive for demonstration purposes only, as both approaches require keys to encrypt the message and verify the signatures. This is solved by defining the key statically in the code, a practice that should never be used in real production, and the keys should be generated dynamically with a secure mechanism. Another thing that needs to be considered is how to transfer the public key to the receiver so that signature verification can be done.

5.5.4 Encryption and Hashing

To demonstrate the encryption and hashing techniques detailed in the theoretical section **(4.2)**, the demo application integrates the RSA-SHA256 and HMAC-SHA256 algorithms, as mentioned in subsection **(5.5.3)**. This selection illustrates the key management differences inherent to encryption methods. RSA, an asymmetric encryption system, relies on a key pair (private and public) for encrypting and decrypting data. In contrast, HMAC uses a symmetric encryption model, necessitating a single key for both encryption and decryption tasks. Furthermore, we employ the Secure Hash Algorithm (SHA-256) for hashing encrypted messages, chosen for its robust security that renders collision discovery computationally perharps infeasible.

Besides RSA-SHA256 and HMAC-SHA256, the demo application also incorporates bcrypt for password encryption, as discussed in the Password Policy and Password Storing section **Section (4.4.4)**. Bcrypt, a cryptographic hash function created in 1999 by Niels Provos and David Mazières based on the Blowfish cipher, is designed for secure password hashing and storage, offering resilience against dictionary attacks [60]. This method is pivotal when handling custom Username and Password authentication mechanisms, ensuring user credentials are securely stored in the database. Integrating the core application with an IAM solution, outlined in the next subsection **User Management (5.5.5)**, further minimizes password vulnerability by offloading user management to services equipped with advanced security measures against suspicious activities.

5.5.5 User Management and Trust Management

The demonstration of User Management within the project showcases two distinct methodologies, allowing for a comparison between traditional and modern approaches. The first method involves direct management of user credentials, where information is collected during registration and saved into the database for authentication at login. This traditional approach, while effective, introduces complexities related to maintaining system security and ensuring that user data remains uncompromised, including a correct way to save the user's password using the recommendations of OWASP.

In contrast, the second method leverages a comprehensive IAM system, in this case Keycloak. This approach streamlines user registration, password changes, allowing email verification, and incorporates advanced security features such as MFA. It enables seamless integration with various IDPs like banking systems, eGovernment platforms, Azure Active Directory, AWS Cognito, and others, including the possibility of connecting with another Keycloak instance. The chief advantage of this approach is the IAM system's capability to manage user data independently, which means that user authentication is not handled by the core application itself, but by the keycloak IAM, with the application relying on it as a trusted IDP for authentication purposes. However, it is crucial for the application to regard the IDP as a secure and reliable source, aligning with the Trust Management principles discussed in **section (4.6)**.

The second approach, the integration of the IAM tool into the demo application, as well as many of the advantages resulting from the analysis discussed in the theoretical part of this thesis, brought many advantages that were considered to be very useful. Including the delegation of user management to the keycloak itself, reducing the need to maintain user credentials in a separate database, and creating an admin portal where it would be possible to maintain users in general. There is a high probability that the SPADe itself will be extended for use cases that would require reimplementation of the current solution if the IAM had not been used. Using the IAM tool brings many benefits that many existing solutions offer, including Identity Federation.

6 Testing and Quality Assurance of The Demo Application

Ensuring the security and integrity of applications in today's digital landscape is crucial. This chapter focuses on evaluating the security measures implemented within the demo application, serving as a proof of concept. A crucial aspect of this evaluation involves conducting thorough penetration testing to identify and rectify potential vulnerabilities that could compromise the application's security. To achieve this, a careful selection of penetration testing tools was undertaken, aimed at uncovering weaknesses that need to be addressed. For the purposes of this master's thesis, the main tool selected is ZAP proxy testing tool.

Identified vulnerabilities could include, but are not limited to, the following:

- XSS vulnerabilities that allow attackers to inject malicious scripts.
- Misconfigurations in server setups (Nginx or Apache) that could lead to unauthorized access.
- CSRF vulnerabilities that could trick a user into performing unintended actions.
- Use of outdated or vulnerable libraries that introduce security flaws into the application.
- Improperly configured HTTP headers that could expose the application to various attacks.

But how exactly is software security testing defined by other sources? The page provided by ZAP[1] defines the process as follows

"Software security testing is the process of assessing and testing a system to discover security risks and vulnerabilities of the system and its data. There is no universal terminology but for our purposes, we define assessments as the analysis and discovery of vulnerabilities without attempting to actually

[1] `https://www.zaproxy.org/getting-started/`

exploit those vulnerabilities. We define testing as the discovery and attempted exploitation of vulnerabilities.[61]"

6.1 ZAP Proxy

The ZAP is a free, open-source tool for security testing of web applications. It works as an intercepting proxy between a browser and web app, examining and modifying data to identify security vulnerabilities. ZAP functions both independently and as a daemon, compatible with various environments and network proxies. Designed for users ranging from novice developers to security experts, it supports all major operating systems and Docker, with extensibility through add-ons from the ZAP Proxy Marketplace. The open-source nature of ZAP encourages community involvement in its development, enhancing its features and addressing security needs [61] .

6.1.1 Vulnerabilities Identified and Addressed by ZAP Proxy Tool

This subsection outlines vulnerabilities identified by the ZAP Proxy tool in the demo application. Given the complexity of web application development, it is possible for initial builds to contain security gaps as developers may not cover everything since the beginning. Here, not only are the detected vulnerabilities listed, but also the applied fixes are briefly explained. For detected issues deemed non-critical or irrelevant to the application's security posture, a reason why is provided for choosing not to address them. It is important to note that while ZAP offers valuable insights and recommendations, not all flagged items may constitute actionable security risks.

The following vulnerabilities were detected by ZAP:

- **Vulnerability 1**: Cloud Metadata Potentially Exposed
- **Vulnerability 2**: Cookies Lacking the HttpOnly Flag
- **Vulnerability 3**: Cookies Missing the SameSite Attribute
- **Vulnerability 4**: Content Security Policy (CSP) Header Not Set
- **Vulnerability 5**: Anti-CSRF Tokens Check

Implemented corrections for these vulnerabilities are as follows:

6.1.2 Corrections For Vulnerabilities

Correction for Vulnerability 1

This vulnerability is related to a misconfigured Nginx server that could potentially provide too much information that could be exploited. The vulnerability is described in more detail here: `https://www.zaproxy.org/docs/alerts/90034/`. However, this vulnerability is mainly related to cloud providers where this application is not hosted, the fix was to add additional information to the Nginx configuration, which may look like this:

```
    # Disabled the display of Nginx Version Number
    server_tokens off;

    server {
        listen 80;
        ...

        # automatic redirect to 403
        # if an attacker tried to abuse this vulnerability

        location /latest/meta-data/ {
            return 403;
        }
        ...
    }

   # Reverse proxy to core instance (scalable - HA)
   server {
    listen 8080;

    location / {
      if ($http_host ~* "^(\d+\.){3}\d+$") {
         return 403;
      }

      # 4. Disable Unnecessary HTTP methods
      if ($request_method !~ ^(GET|OPTIONS|POST|UPGRADE)$) {
              return 405;  # Method Not Allowed
      }

      # 5. Disable TRACE and TRACK
      if ($request_method = TRACE) {
        return 405;  # Method Not Allowed
      }

      # 6. Modify Nginx Web Server Configuration/SSL for X-
   XSS protection
```

```
      add_header X-XSS-Protection "1; mode=block";

      if ($http_host = "169.254.169.254") {
        return 403;
      }

      proxy_pass http://back-end/;
      proxy_http_version 1.1;
      proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /latest/meta-data/ {
        return 403;
    }

  }
```

Listing 6.1: nginx.conf file

However, as strongly recommended on the official Nginx site[2] and also in the OWASP Foundation, the main protection is not to use this in any case to block the Cloud Metadata Attack:

```
    # Don't ever use 'proxy_pass' like this!
    location / {
        proxy_pass http://$host; # To repeat: don't do this!
    }
```

Correction for Vulnerability 2

This vulnerability was caused by sending the Cookie in a HTTP header with a CSRF Token to prevent CSRF attacks from the server to the client without setting the so-called HTTP only flag. The fix is illustrated by the snippet of the code:

```

    ResponseCookie cookie = ResponseCookie
                    .from(CSRF_PARAMETER_NAME, key)
                    .httpOnly(true)
                    .secure(false)
                    .maxAge(Duration.ofHours(2))
                    .sameSite("Lax")
                    .build();
```

Listing 6.2: Creation of Reponse Cookie Correctly

[2]https://www.nginx.com/blog/trust-no-one-perils-of-trusting-user-input/

Note that the secure is set to false. This is due to the omission of TLS. Otherwise it should be set to true, as this flag ensures that the cookie is only sent via HTTPS.

Correction for Vulnerability 3

This vulnerability was fixed at the same time as vulnerability no. 2 above. As the cookie should be set on a specific site to avoid a false cross-site request (CSRF) attack. The code snippet shows how to set the cookie to be sent only on the same site (sameSite method). The "Lax" option ensures that the cookie is not sent in cross-domain GET requests.

Correction for Vulnerability 4

Considering the nature of the application and the measures in place, it seems unlikely that the application is vulnerable to XSS attacks. This is because there are no apparent opportunities for Hypertext Markup Language (HTML) injection within the application, and no critical endpoints have been identified that might allow such vulnerabilities to be exploited by external parties. Additionally, areas of potential concern, such as the chat room, have undergone thorough testing and have mechanisms in place to ensure that all text is appropriately escaped. From the admin's perspective, viewing saved messages appears to be secure, as these too are properly escaped. Therefore, most of the CSP vulnerabilities identified by the ZAP Proxy Tool may not pose a significant risk or cause any tangible harm to the application.

It is also important to note that most applications used by many users may suffer almost daily from XSS attacks, as there is no way to secure all the possibilities that may occur in very large systems that perform many operations and offer many functionalities, including data insertion and data manipulation in general. Therefore, a quick reaction to feedback from users and proper monitoring of the systems are key practices in any system to increase the robustness of the system. Proper monitoring can help to identify problems in the system more easily or in some cases it can prevent attacks by early detection.

Correction for Vulnerability 5

The CSRF attack prevention was done by adding another database, the Redis database, which is used as a CSRF token store that is shared across all instances of the core application that can be load balanced. As this approach may be primitive and not suitable for real production, for demonstration purposes, the ReactJS client calls the core application for a new CSRF token each time the client's application context changes (the client visits another page on the ReactJS or refreshes the page). This token is then sent from the ReactJS application to the server with each request, otherwise the client is not authorized to make API calls and must re-login to the application. The token is sent as a cookie in the HTTP header. How the cookie is created is illustrated in the **Correction for Vulnerability 2** code snippet. Each generated token is stored in the Redis database and then checked against the token sent by the client to see if it is valid.

The place at code where this CSRF layer protection is added in is defined in Spring Security Filter Chain. The definition looks like this:

```
    http
        .csrf(e -> {
             e.csrfTokenRepository(redisCsrfTokenRepository);
             e.ignoringRequestMatchers("/v1/chat/**", "/v1/auth/validate");
                })
```

Listing 6.3: Security Filter Chain CSRF protection Setup

There are also other ways to handle CSRF protection. One of them is to integrate the OWASP CSRFGuard package, or to set a custom csrf token for each form used on the site.

7 Conclusion

This thesis focused on an exploration of microservices architecture, dealing with the mechanisms of communication between services and the pivotal role of cybersecurity, particularly emphasizing the Application Security Verification Standard (ASVS) made by Open Web Application Security Project (OWASP) . It showcased various cybersecurity methodologies through a demonstration application, covering critical areas such as authentication, authorization, protocols for secure access, encryption, hashing, secure data transfer, Identity and Access Management (IAM), and Trust Management (TM). These components collectively ensure the safe delegation of trust and authorization across services.

The demonstration application, illustrating a microservices architecture, utilized websockets for real-time communication and Representational State Transfer Application Programming Interface (REST API) for standard interactions. The analysis of authentication methods and protocols revealed the complexity and potential security pitfalls of manual implementation versus the benefits of employing a comprehensive Indentity and Access Management (IAM) solution like Keycloak. Through practical application, the thesis compared protocols like OpenID Connect (OIDC) and Security Assertion Markup Language (SAML), ultimately integrating OpenID Connect (OIDC) into the SPADe project for its robust security features and ease of use.

In assessing the quality and security of the demo application, the Zed Attack Proxy (ZAP) tool was employed to identify and mitigate critical vulnerabilities. This process of testing and refining ensures that the application, while proof of concept, adheres to some security standards.

For future work in this field, the emphasis on established and tested security practices cannot be overstated. Adopting methodologies recommended by authoritative sources such as Open Web Application Security Project (OWASP) and National Institute of Standards and Technology (NIST) is crucial. These sources provide up-to-date guidance, ensuring applications are not only reliable but also secure against potential breaches, thereby safeguarding sensitive data, perhaps effectively, as today all applications running especially in remotely accessible environments, like cloud environments, have to face the reality that there is a high probability that even if it is a small application, it may (but does not necessarily have to) be subject to some kind of attack mentioned in the **section 4.7.3**.

Used Shortcuts

AD Active Directory. 32

AES Advanced Encryption Standard. 24

AMQP Advanced Message Queuing Protocol. 18

API Application Programming Interface. 11, 14, 34, 46, 62, 63, 91

APIs Application Programming Interfaces. 16, 29, 62

ASVS Application Security Verification Standard. 19, 20

AWS Amazon Web Services. 42, 44, 66, 84

CI/CD Continuous Integration/Continuous Delivery. 64

CSP Content Security Policy. 87, 90

CSRF Cross-Site Request Forgery. 62, 68, 71, 86, 87, 89–91

DAP Directory Access Protocol. 57

DDoS Distributed Denial of Service. 62

DES Data Encryption Standard. 24

DoS Denial of Service. 60, 62

GDPR General Data Protection Regulation. 19

gRPC gRPC Remote Procedure Calls. 14, 15

GUI Graphical User Interface. 70

HA High Availability. 72

HMAC Hash-Based Message Authentication Codes. 23, 24

HTML Hypertext Markup Language. 90

HTTP Hypertext Transfer Protocol. 14, 16, 46, 66, 69, 86, 89, 91

HTTPS Hypertext Transfer Protocol Secure. 28, 48, 90

IAM Identity and Access Management. 10, 19, 28, 42, 43, 53, 59, 60, 66, 74–76, 80, 82, 84, 85

IBM International Business Machines Corporation. 42

ID Identifier. 50

IDP Identity Provider. 32, 49, 51–53, 76, 84

IDs Identifiers. 32

IETF Internet Engineering Task Force. 27, 46

IoT Internet of Things. 18

IP Internet Protocol. 32, 35, 57, 61

IT Information Technology. 35

JWT JSON Web Token. 46, 50, 59, 63, 66, 68, 69, 73, 74, 78, 79

KYC Know Your Customer. 29

LDAP Lightweight Directory Access Protocol. 43, 49, 57

MAC Media Access Control. 61

MFA Multi-Factor Authentication. 43, 84

MQ Message Queue. 14, 15, 17, 18

MQTT Message Queuing Telemetry Transport. 18

NCSC National Cyber Security Centre. 30

NFC Near Field Communication. 40

NIST National Institute of Standards and Technology. 10, 20, 28, 30, 41, 61, 70

OAuth Open Authorization. 46, 47, 50

OIDC OpenID Connect. 42, 44, 49–51, 60, 63, 64, 66, 69, 70, 75, 77, 79, 81, 82

OP OpenID Connect Provider. 49, 50

OWASP Open Web Application Security Project. 10, 19, 20, 23, 28–32, 60, 61, 70, 73, 84, 89, 91

PKI Public Key Infrastructure. 36

RBA Risk-Based Authentication. 28, 30, 33, 40, 41, 44, 59, 70, 82

RBAC Role-Based Access Control. 42, 59

REST Representational State Transfer. 14–17

REST API Representational State Transfer Application Programming Interface. 63, 73, 76

RP Relying Parties. 49

SAML 2.0 Security Assertion Markup Language. 42, 49, 51, 52, 63, 66, 72, 75, 77, 81

SOA Service Oriented Architecture. 11, 12, 14

SP Service Provider. 52, 53, 81

SPA Single Page Application. 48, 81

SQL Structured Query Language. 23, 60, 62, 74

SSH Secure Shell Protocol. 37

SSL Secure Sockets Layer. 21, 27, 28, 37

SSO Single Sign-On. 35, 43, 50, 60, 63, 66, 72, 81

STOMP Streaming Text Oriented Messaging Protocol. 18, 63, 68, 69

TLS Transport Layer Security. 14, 21, 24, 26–28, 37, 80, 90

TM Trust Management. 10, 59

URI Uniform Resource Identifier. 48, 53, 66

URL Uniform Resource Locator. 16, 31, 60

USB Universal Serial BUS. 40

VoIP Voice Over IP. 27

XML Extensible Markup Language. 51–53

XSS Cross-Site Scripting. 60, 62, 86, 90

ZAP Zed Attack Proxy. 31, 64, 86, 87, 90

Bibliography

[1] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[2] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages," in *Proceedings of the XP2017 Scientific Workshops*, ser. XP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3120459.3120483

[3] R. Chandramouli, "Microservices-based application systems," *NIST Special Publication*, vol. 800, no. 204, pp. 800–204, 2019.

[4] Microsoft, "Design interservice communication for microservices," https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication, accessed: 2024-05-02.

[5] V. Surwase, "Rest api modeling languages-a developer's perspective," *Int. J. Sci. Technol. Eng*, vol. 2, no. 10, pp. 634–637, 2016.

[6] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[7] What is a message queue? Accessed: 2024-03-20. [Online]. Available: https://aws.amazon.com/message-queue/

[8] "What are message queues," accessed: 2024-03-20. [Online]. Available: https://www.ibm.com/topics/message-queues

[9] Cloudflare, "What is serverless computing?" https://www.cloudflare.com/learning/serverless/what-is-serverless/, accessed: 2024-04-04.

[10] CloudAMQP, "Rabbitmq and websockets, part 1: Amqp, mqtt, and stomp," https://www.cloudamqp.com/blog/rabbitmq-and-websockets-part-1-amqp-mqtt-stomp.html, 2023, accessed: 2024-05-02.

[11] Owasp application security verification standard. [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/

[12] "OWASP API Security Top Ten 2019," https://owasp.org/API-Security/editions/2019/en/0x11-t10/, 2019, accessed: 2024-02-25.

[13] "OWASP API Security Top Ten 2023," https://owasp.org/API-Security/editions/2023/en/0x11-t10/, 2023, accessed: 2024-02-25.

[14] GeeksForGeeks, "What is hashing?" https://www.geeksforgeeks.org/what-is-hashing/, 2024, accessed: 2024-05-02.

[15] OWASP, "Methods for enhancing password storage," https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#methods-for-enhancing-password-storage, accessed: 2024-05-02.

[16] GeeksForGeeks, "Hash functions in system security," https://www.geeksforgeeks.org/hash-functions-system-security/, 2024, accessed: 2024-05-02.

[17] A. Alessandrini, "Hashing, salts and pepper," https://www.linkedin.com/pulse/hashing-salts-pepper-andrea-alessandrini-kjvqf/, 2023, accessed: 2024-05-02.

[18] Cloudflare, "What is encryption?" accessed: 2024-05-02. [Online]. Available: https://www.cloudflare.com/learning/ssl/what-is-encryption/

[19] GeeksForGeeks, "Difference between aes and des ciphers," https://www.geeksforgeeks.org/difference-between-aes-and-des-ciphers/, 2023, accessed: 2024-05-02.

[20] Cloudflare, "What is asymmetric encryption?" https://www.cloudflare.com/learning/ssl/what-is-asymmetric-encryption/, accessed: 2024-05-02.

[21] M. C. Kate Brush, "asymmetric cryptography," https://www.techtarget.com/searchsecurity/definition/asymmetric-cryptography, 2024, accessed: 2024-05-02.

[22] Okta, "Hmac (hash-based message authentication codes) definition," https://www.okta.com/identity-101/hmac/, 2023, accessed: 2024-05-02.

[23] N. Gupta, "Symmetric vs. asymmetric encryption – what are differences?" https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences, accessed: 2024-04-06.

[24] SavvySecurity, “What is the difference between a digital signature and a digital certificate,” https://cheapsslsecurity.com/blog/digital-signature-vs-digital-certificate-the-difference-explained/, accessed: 2024-04-06.

[25] Geeksforgeeks, “Difference between hashing and encryption,” https://www.geeksforgeeks.org/difference-between-hashing-and-encryption/, 2023, accessed: 2024-05-02.

[26] cloudflare, “Introduction to ssl/tls,” https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/, accessed: 2024-04-06.

[27] “What is identity and access management (iam)?” accessed: 2024-03-23. [Online]. Available: https://www.oracle.com/cz/security/identity-management/what-is-iam/

[28] N. I. of Standards and Technology, “Guideline for identifying an information system as a national security system,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 800-59, 2003, adapted from CNSSI 4009 for Authentication.

[29] OWASP Foundation, “Authentication Cheat Sheet,” https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html, n.d., accessed: 17-02-2024.

[30] “API2:2023 Broken Authentication - OWASP API Security Project,” https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/, accessed: 2024-02-29.

[31] S. Agarwal, “Best practices for username and password authentication,” 2024, accessed: 2024-03-25.

[32] “Password Storage Cheat Sheet - OWASP Cheat Sheet Series,” https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html, accessed: 2024-03-03.

[33] “What is ip address authentication?” accessed: 2024-03-25. [Online]. Available: https://connect.ebsco.com/s/article/What-is-IP-Address-Authentication?language=en_US

[34] “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” https://datatracker.ietf.org/doc/html/rfc5280, 2008, accessed: 2024-03-04.

[35] "Structure of an ssl (x.509) certificate," https://dev.to/wayofthepie/structure-of-an-ssl-x-509-certificate-16b, 2020, accessed: 2024-05-01.

[36] "X.509 authentication service," https://www.geeksforgeeks.org/x-509-authentication-service/, 2023, accessed: 2024-05-01.

[37] "Cryptography Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html, OWASP, 2021, accessed: 2024-03-04.

[38] P. Pullanieswaran, "Webauthn - a short introduction," 2024, accessed: 2024-03-25.

[39] A. Dagnelies, "Webauthn - passwordless registration/login flows," https://dev.to/dagnelies/webauthn-flows-51ic, 2022, accessed: 2024-04-06.

[40] S. Wiefling, L. Lo Iacono, and M. Dürmuth, "Is this really you? an empirical study on risk-based authentication applied in the wild," in *ICT Systems Security and Privacy Protection*, G. Dhillon, F. Karlsson, K. Hedström, and A. Zúquete, Eds. Cham: Springer International Publishing, 2019, pp. 134–148.

[41] "Identity and access management (iam) technologies and tools," accessed: 2024-03-23. [Online]. Available: https://clearskye.com/identity-and-access-management/iam-technologies-tools

[42] IETF, "The oauth 2.0 authorization framework," https://tools.ietf.org/html/rfc6749, 2012.

[43] Okta, "What is oauth 2.0?" https://auth0.com/intro-to-iam/what-is-oauth-2, accessed: 2024-05-01.

[44] Oracle, "Oauth 2.0," https://docs.oracle.com/cd/E82085_01/160027/JOS%20Implementation%20Guide/Output/oauth.htm, accessed: 2024-05-01.

[45] auth0, "Authorization code flow with proof key for code exchange (pkce)," https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce.

[46] OpenID Foundation, "Openid connect," https://openid.net/connect/, 2022.

[47] Mozilla, "Openid connect," https://infosec.mozilla.org/guidelines/iam/openid_connect.html, accessed: 2024-05-01.

[48] NHS, “Openid connect (oidc) overview,” https://digital.nhs.uk/services/care-identity-service/applications-and-services/cis2-authentication/guidance-for-developers/openid-connect-overview, 2023, accessed: 2024-05-01.

[49] N. Kushwaha, “Understanding openid connect (oidc),” https://www.learncsdesign.com/understanding-openid-connect-oidc/, accessed: 2024-05-01.

[50] L. Kakavas, “Saml authentication and the elastic stack,” https://www.elastic.co/blog/how-to-enable-saml-authentication-in-kibana-and-elasticsearch, 2018, accessed: 2024-05-01.

[51] Fortinet, “Kerberos authentication,” https://www.fortinet.com/resources/cyberglossary/kerberos-authentication, accessed: 2024-04-06.

[52] E. Kost. (2024) What is ldap? how it works, uses, and security risks. Accessed: 2024-03-04. [Online]. Available: https://www.upguard.com/blog/ldap

[53] “Lightweight directory access protocol (ldap),” https://www.geeksforgeeks.org/lightweight-directory-access-protocol-ldap/, 2019, accessed: 2024-05-01.

[54] M. Blaze, J. Ioannidis, and A. D. Keromytis, “Experience with the keynote trust management system: Applications and future directions,” in *Trust Management*, P. Nixon and S. Terzis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 284–300.

[55] M. Blaze, J. Feigenbaum, and J. Lacy, “Decentralized trust management,” in *Proceedings 1996 IEEE Symposium on Security and Privacy*, 1996, pp. 164–173.

[56] “Input validation cheat sheet,” accessed: 2024-03-22. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[57] “So what is cloudflare,” accessed: 2024-03-22. [Online]. Available: https://www.cloudflare.com/learning/what-is-cloudflare/

[58] Keycloak, “Keycloak documentation,” https://www.keycloak.org/documentation.html, 2024, accessed: 2024-03-29.

[59] S. Choudry, “Why you wouldn’t use saml in a spa and mobile app,” https://www.identityserver.com/articles/why-you-wouldn-t-use-saml-in-a-spa-and-mobile-app, 2022, accessed: 2024-05-02.

[60] M. Grigutytė, “What is bcrypt and how does it work?” https://nordvpn.com/blog/what-is-bcrypt/, 2023, accessed: 2024-03-31.

[61] zap, “Getting started,” https://www.zaproxy.org/getting-started/, accessed: 2024-04-07.

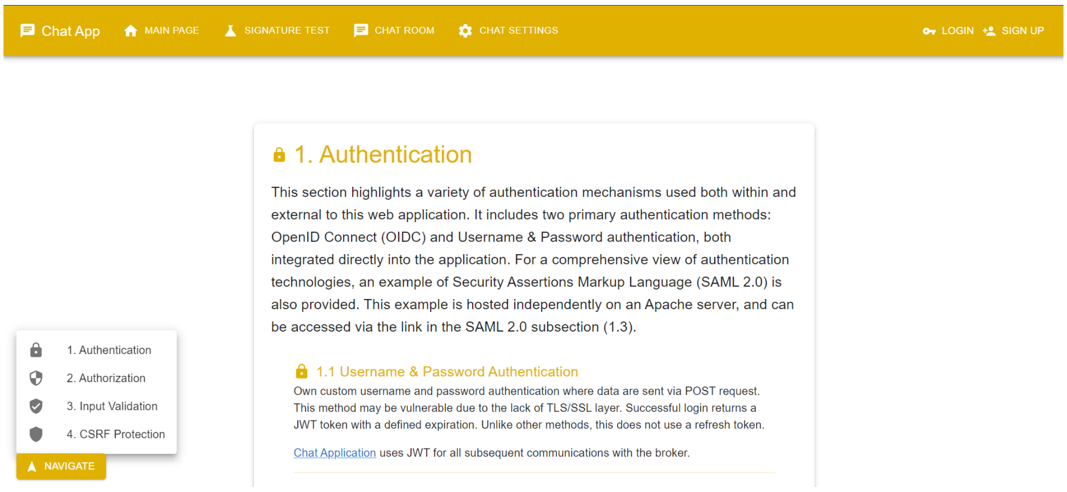
Attachments

A Description of the directory structure of the submitted file

Directory/File	**Description**
`Application_and_libraries/`	Main folder containing all application-specific subfolders and configuration files.
`1 - Demo_Application/`	Contains subdirectories for the components of the demo application and a Docker Compose file to automate the build and run process:
apache-saml	Setup files for Apache with Shibboleth SP, used in Docker.
backend	Java and resource files of the core application, including secrets and Docker configuration.
broker	Source code and Maven project files.
frontend	ReactJS project files, Nginx configuration, and Docker setup.
imports	Contains Keycloak exported realms automatically imported during the Docker Compose up phase.
docker-compose.yml	Automates the entire application setup and deployment.
`2 - SPADe_SPAWn/`	Contains subdirectories for the SPADe and SPAWn applications along with Docker Compose configurations:
imports	Keycloak realms for automatic import during Docker Compose up.
mssql	Dockerfile and scripts for setting up the MSSQL environment.
spade	Spring Boot application with necessary Docker setup files.
spawn	ReactJS application with Docker and Nginx configurations.
docker-compose.yml	Automates the building and running of the application.
`Poster/`	Contains PDF and Publisher files of the project poster.
`Text_thesis/`	Source files and the compiled PDF document of the thesis.

Directory/File	**Description**
`Readme.txt`	Provides general information about the project setup and usage.

B Images of the graphical user interface of the demo application



1. Authentication

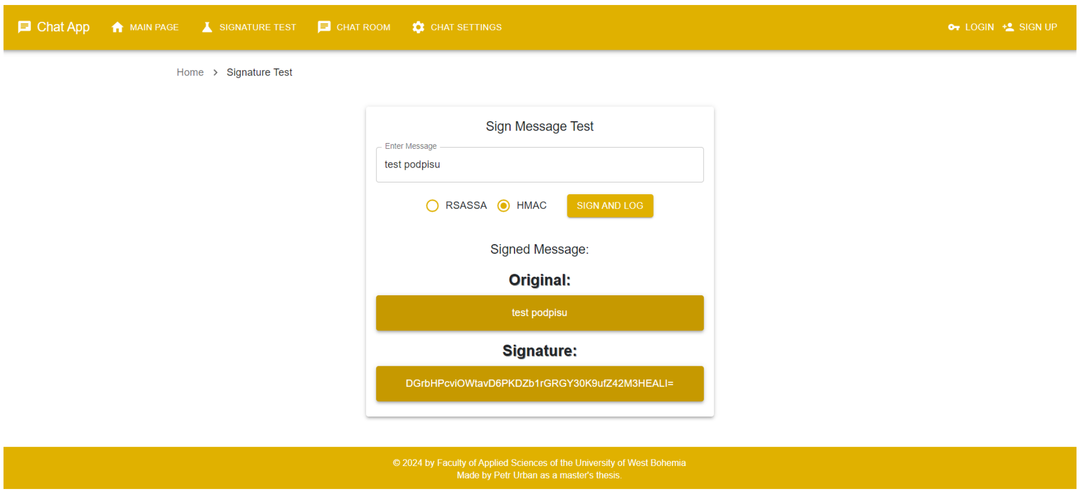
This section highlights a variety of authentication mechanisms used both within and external to this web application. It includes two primary authentication methods: OpenID Connect (OIDC) and Username & Password authentication, both integrated directly into the application. For a comprehensive view of authentication technologies, an example of Security Assertions Markup Language (SAML 2.0) is also provided. This example is hosted independently on an Apache server, and can be accessed via the link in the SAML 2.0 subsection (1.3).

1.1 Username & Password Authentication

Own custom username and password authentication where data are sent via POST request. This method may be vulnerable due to the lack of TLS/SSL layer. Successful login returns a JWT token with a defined expiration. Unlike other methods, this does not use a refresh token.

Chat Application uses JWT for all subsequent communications with the broker.

Figure B.1: Main Page



test podpisu

RSASSA [✓ HMAC]

Signed Message:

Original:

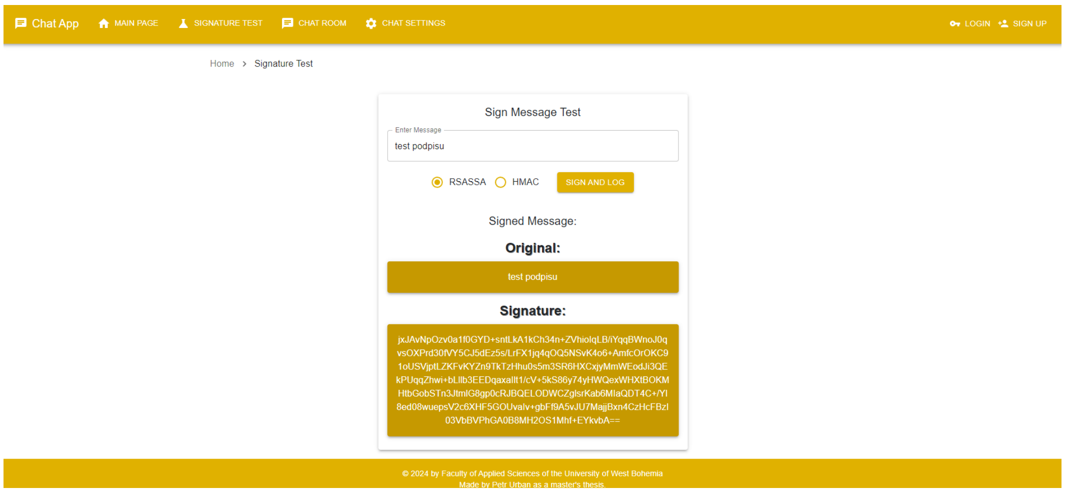
test podpisu

Signature:

DGrbHPcviOWtavD6PKDZb1rGRGY30K9ufZ42M3HEALI=

© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.2: Signature Test Page - HMAC



Sign Message Test

Enter Message: test podpisu

[✓ RSASSA] HMAC

Signed Message:

Original:

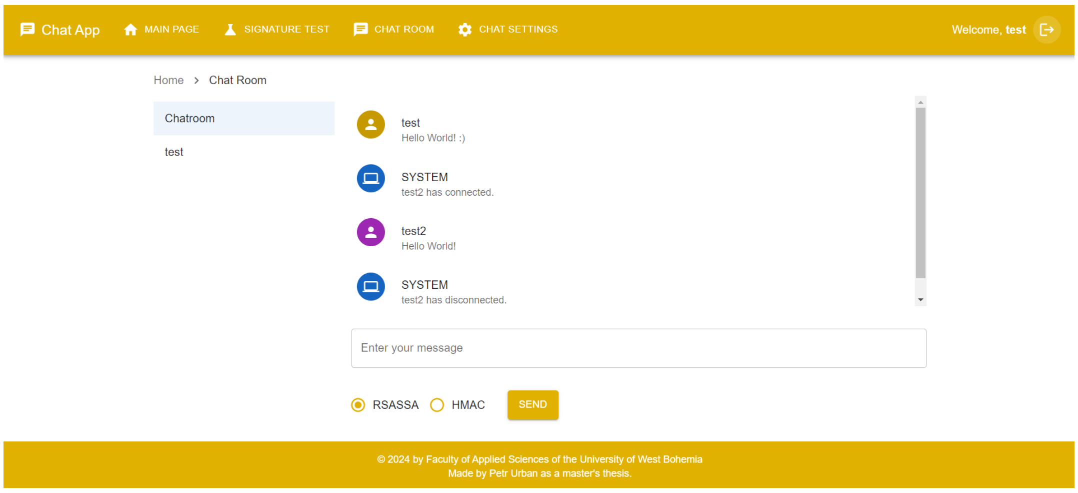
test podpisu

Signature:

jxJAvNpOzv0a1f0GYD+sntLkA1kCh34n+ZVhioIqLB/iYqqBWnoJ0q
vsOXPrd30fVY5CJ5dEz5s/LrFX1jq4qOQ5NSvK4o6+AmfcOrOKC9
1oUSVjptLZKFvKYZn9TkTzHhu0s5m3SR6HXCxjyMmWEodJi3QE
kPUqqZhwi+bLIlb3EEDqaxallt1/cV+5kS86y74yHWQexWHXtBOKM
HtbGobSTn3JtmIG8gp0cRJBQELODWCZglsrKab6MIaQDT4C+/YI
8ed08wuepsV2c6XHF5GOUvaIv+gbFf9A5vJU7MajjBxn4CzHcFBzl
03VbBVPhGA0B8MH2OS1Mhf+EYkvbA==

© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.3: Signature Test Page - RSASSA



test
Hello World! :)

SYSTEM
test2 has connected.

test2
Hello World!

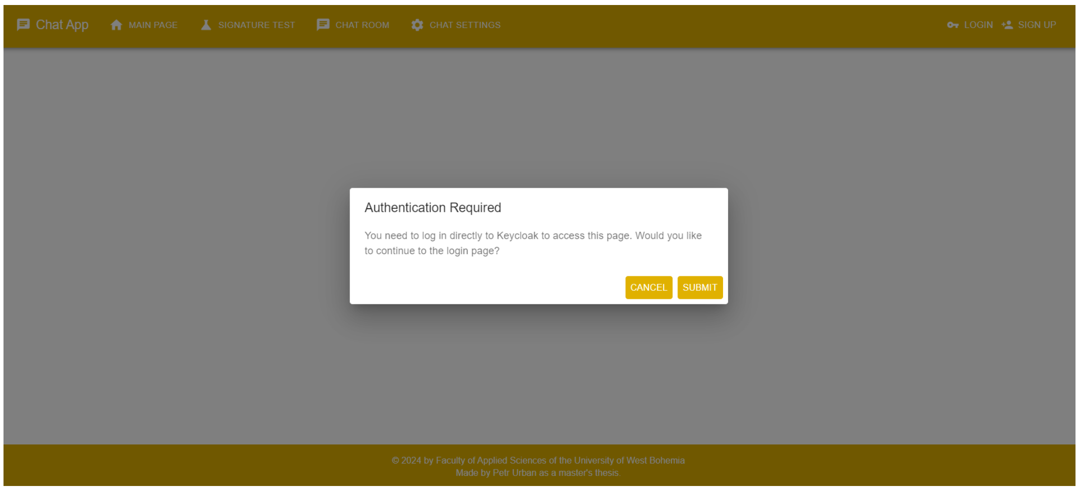
SYSTEM
test2 has disconnected.

Enter your message

[✓ RSASSA] HMAC

© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.4: Chat Room

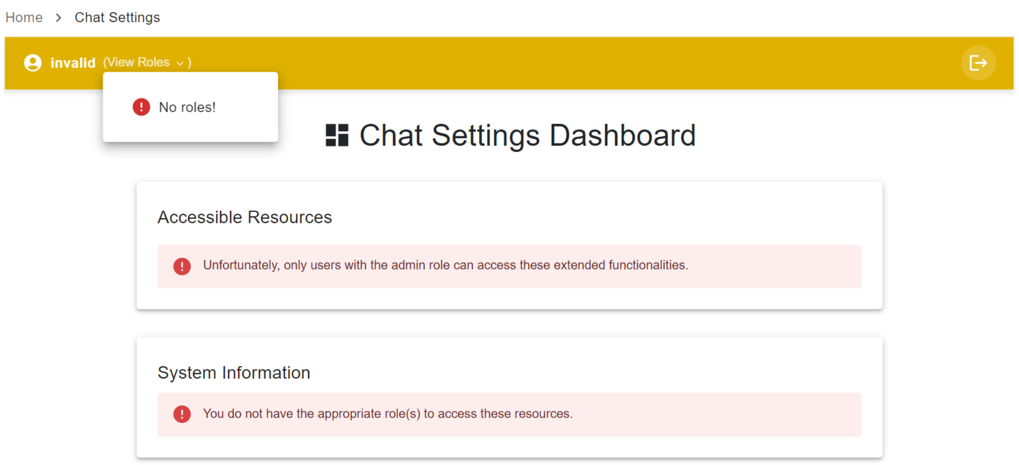


Authentication Required

You need to log in directly to Keycloak to access this page. Would you like to continue to the login page?

© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.5: Chat Settings - Authentication



No roles!

Chat Settings Dashboard

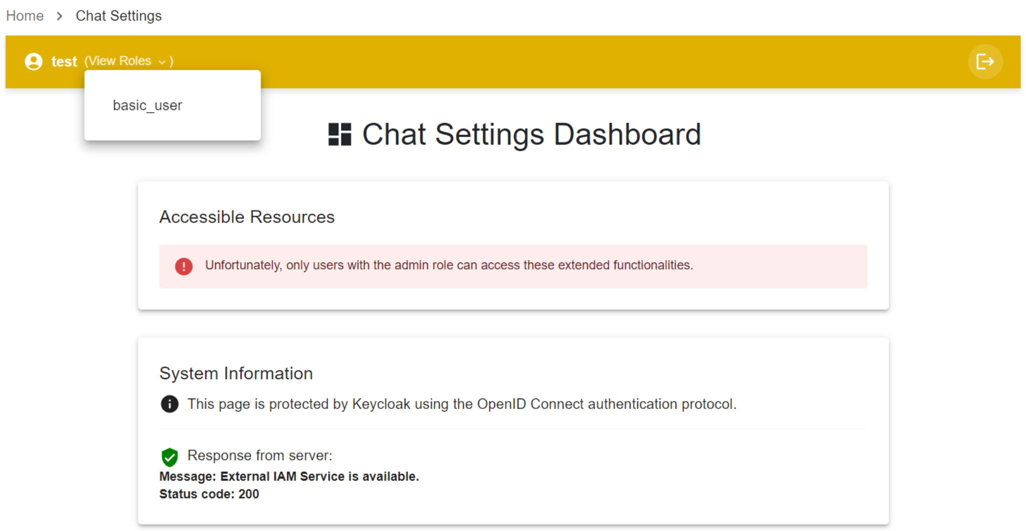
Accessible Resources

Unfortunately, only users with the admin role can access these extended functionalities.

System Information

You do not have the appropriate role(s) to access these resources.

Figure B.6: Chat Settings RBA - User Without Roles

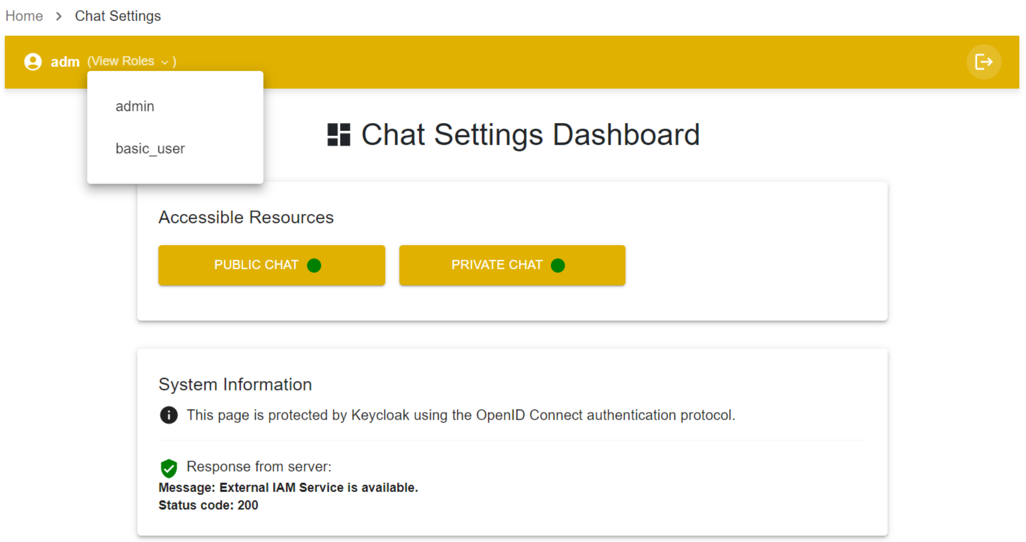


Unfortunately, only users with the admin role can access these extended functionalities.

This page is protected by Keycloak using the OpenID Connect authentication protocol.

Response from server:
Message: External IAM Service is available.
Status code: 200

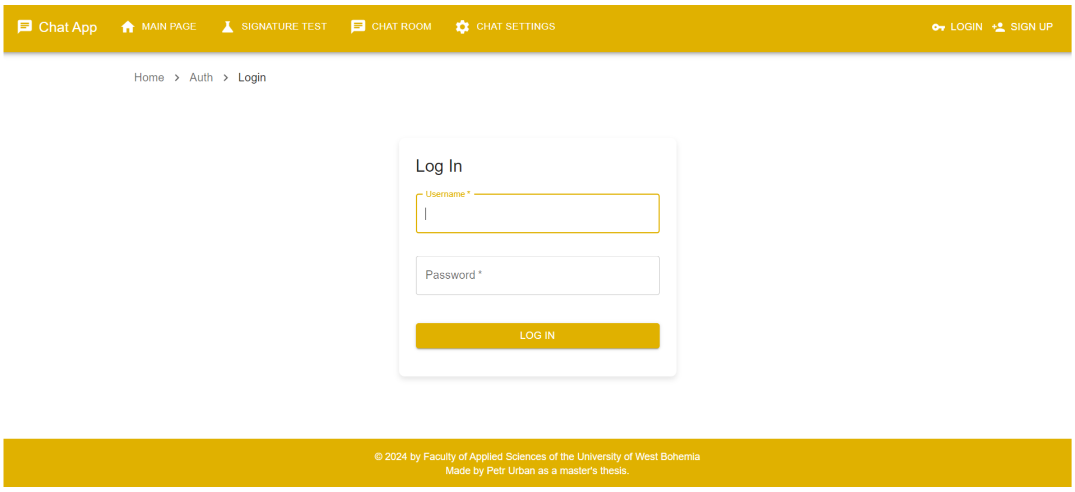
Figure B.7: Chat Settings RBA - Casual User



This page is protected by Keycloak using the OpenID Connect authentication protocol.

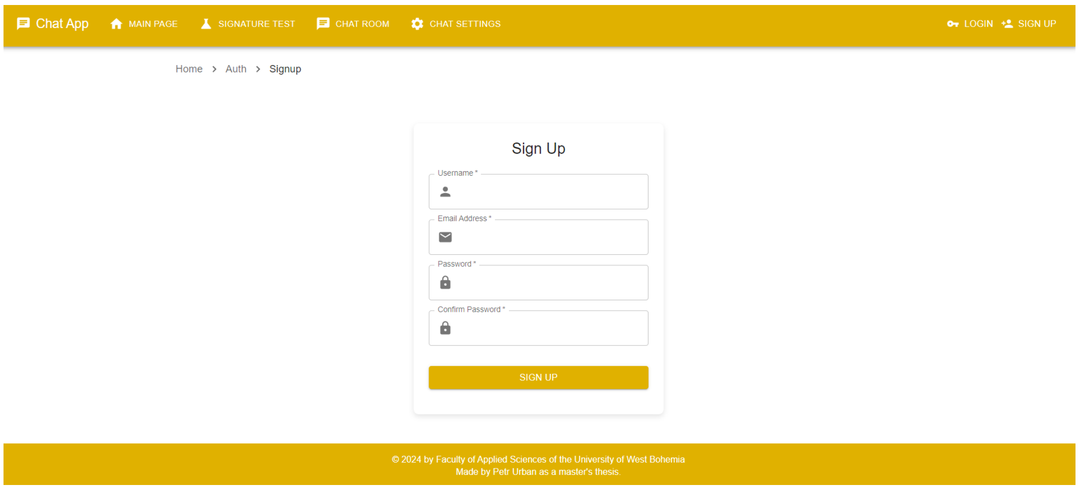
Response from server:
Message: External IAM Service is available.
Status code: 200

Figure B.8: Chat Settings RBA - Admin User



© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.9: Login Page



© 2024 by Faculty of Applied Sciences of the University of West Bohemia
Made by Petr Urban as a master's thesis.

Figure B.10: Signup Page