FACULTY OF SCIENCE AND TECHNOLOGY

MASTER THESIS

Study programme / specialization:
Computer Science: Secure and Reliable Systems

The spring semester, 2022

Open Access

Author: Vera Yaseneva

(signature author)

Supervisor(s): Hein Meling

Thesis title: QuickFeed Security: Redesigned Authentication and Authorization Architecture

Credits (ECTS): 30

Keywords:
authentication, access control, token-based authentication, gRPC-web, authentication tokens, Go, application security, JWT

Pages: 63
+ appendix: 1

Stavanger, 15.06.2022
date/year

Approved by the Dean 30 Sep 21
Faculty of Science and Technology

University of Stavanger

VERA YASENEVA
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

QuickFeed Security: Redesigned Authentication and Authorization Architecture

Master's Thesis - Computer Science - June 2022

I, **Vera Yaseneva**, declare that this thesis titled, "QuickFeed Security: Redesigned Authentication and Authorization Architecture" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

"Now you people have names. That's because you don't know who you are. We know who we are, so we don't need names."

– Neil Gaiman, Coraline

Abstract

QuickFeed (former Autograder) is a software project developed at the University of Stavanger. The application performs automated grading of coding assignments and provides nearly instant feedback to the students in programming courses.

Authentication (establishing the identity of a user) and authorization (defining what types of data a user can access or modify) comprise an essential part of any web-based application.

QuickFeed went through multiple reworks and updates, but the authentication module remained unchanged. As a result, it is necessary to keep extra steps to ensure interoperability between the authentication module and the rest of the application.

This makes the affected parts of the QuickFeed codebase unnecessary complex, somewhat redundant, and hard to understand and maintain.

This thesis work is dedicated to redesigning the authentication and authorization architecture of Quickfeed in order to enhance security and improve maintainability and scalability of the project.

Acknowledgements

I would like to express my deepest gratitude to my supervisor and mentor, Professor Hein Meling, for helping me grow academically, professionally, and as a person.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Authentication and authorization comprise an essential part of the modern web-based software that can have a significant impact on the security of an application and, at the same time, its overall usability. When developing authentication and authorization processes for a software project it is important to achieve a balance between these two sides. On the one hand, a sign-in process must be fast and simple, otherwise it can be perceived as too cumbersome and complicated by the end user. On the other hand, the main goal of authentication and authorization is to provide security and this function cannot be compromised on.

The current authentication and authorization flow of the QuickFeed (former Autograder) software project is both robust, secure, and user-friendly but it still has a few weaknesses that can be improved upon. The goal of our thesis work is to address these weaknesses, evaluate possible solutions and, finally, choose and implement those that fit best the current architecture and future development goals of the project. These changes aim to not only improve security, but also make future development of the project easier by designing more simple, clear, and structured code.

1.1 QuickFeed's evolution

The QuickFeed web application has evolved over multiple generations including one complete rewrite and a multitude of refactorings and feature additions.

The original Autograder was designed and implemented as part of a master's thesis by Heine Furubotten in 2014 [1]. The second generation Autograder was

designed and implemented by a team of three students supervised by Prof. Meling in the summer of 2017.

The application was intended to assist in assignment grading by automated downloading, testing, and grading of code submitted by students in programming courses.

Integration with GitHub is an essential part of the application. Users sign in to Autograder with their GitHub accounts, courses are based on GitHub organizations, and students submit code by uploading it to GitHub repositories.

To automate Autograder's integration with GitHub it was necessary to store personal access tokens in the database. Tokens would be used to authenticate GitHub API clients that performed actions on the course repositories.

To keep track of authenticated users, the Autograder server kept a session store with a session for each user.

The second generation Autograder has evolved substantially since its initial design in 2017. The http-based original REST API has been replaced with gRPC calls in order to improve the application's scalability, maintainability, and performance.

In 2020, the project was renamed to QuickFeed to better reflect the goal to provide rapid feedback to students, and not only an automated grading tool.

While not strictly accurate, we will refer to QuickFeed as the version that contains the redesigned security architecture to be presented in this thesis. We will refer to Autograder when discussing the version prior to this thesis.

1.2 Motivation

While the Autograder's web interface and general purpose API went through multiple reworks, the authentication architecture remained unchanged. As a result, it is challenging to maintain interoperability between the http-based authentication endpoints and gRPC-based business logic API.

The original Autograder was designed to integrate with GitHub as an OAuth app which was the only option back when the application was developed. The app is responsible for the initial user authentication with GitHub accounts. When a user logs in with GitHub credentials, a personal access token is collected from GitHub and stored in the Autograder's database. The token is later used to access course information and students' code stored in GitHub repositories.

Storing and sending tokens with each request to GitHub is a security risk. Access tokens are strings of random characters and numbers and are stored in the Autograder database as plaintext. If a token is stolen due to a database exploit or a network attack, it can be used by the attacker to access, modify or steal some personal data of the authorized user.

Autograder creates and maintains a session for each logged in user. Session-based authentication was convenient back when both authentication and general API were based on http endpoints. Now the gRPC API cannot access the sessions directly. To allow gRPC method to access information about the logged in users it is necessary to maintain an additional map structure that links each session ID to the corresponding user ID.

As a result, there are two redundant structures for user sessions that make this part of the authentication architecture unnecessarily complex, hard to understand and maintain. Moreover, a session-based authentication process can have a negative impact on the scalability of the application. Memory consumption overhead grows in proportion to the number of users and can affect performance. Sessions are also poorly compatible with any kind of distributed architecture as it would be necessary to keep an additional session store for each service.

In the original Autograder both authentication and business logic APIs were based on http endpoints. Two middleware methods were developed back then to perform similar authorization checks for each API. Now that the general purpose API is gRPC-based the second middleware method serves no purpose and is yet another source of redundancy in the code.

The original Autograder performs access control checks on a per-method basis. As a result, even if a request comes from a user who is not authorized to call the method, it will still start executing, perform the necessary checks and only then return an error. To ensure the privileges of a user it is necessary to query the database. As a result, even a successful request has its turnaround time limited by the database read operation. Moreover, access restrictions are difficult to track, maintain and update as the required user privileges are only mentioned in the comments for each method.

Finally, QuickFeed uses an external Envoy proxy to allow browser clients to send gRPC requests to the server. The proxy requires an obscure configuration file that has to be changed in order to run the application locally which complicates the development process for every new team that starts working on the

project.

1.3 Contributions

This thesis contributes a substantial redesign of QuickFeed's security architecture to resolve the issues highlighted in Section 1.2. The redesigned security architecture includes the following features and updates:

1. **Redesigned authentication process**. We replace stateful session-based user authentication by a stateless alternative with JSON Web Tokens (JWTs). JWTs serve as portable units of user identity and can be used directly by both http and gRPC APIs. Tokens are signed to ensure the integrity of user claims and can be safely used in access control. There is no need for redundant structures and methods.
2. **Modified QuickFeed's integration with GitHub**. GitHub app is a modern solution for integration with GitHub. it can access GitHub API directly without relying on personal access tokens of authenticated users.
3. **Implemented centralized access control**. Access control module confirms the user's role in a specific course before granting access to the server. All restricted methods and required roles are collected into a single table which to make the code responsible for authorization checks easier to maintain and update.
4. **Encrypted access token storage.** For situations where storing access tokens in the database is still required we add an option to encrypt all tokens before saving. Even if a database is eventually exposed to an attack, tokens cannot be used to access information belonging to the users.
5. **Proxyless server-client communication.** Clients can now send requests directly to the QuickFeed server. There is no need to keep a complex proxy configuration up to date. It is now easy to run and test the application on a local machine.

1.4 Outline

The rest of the thesis is organized as follows:

- Chapter 2 introduces the theoretical background relevant to our thesis work
- Chapter 3 is dedicated to the replacement of session-based authentication patter with stateless JSON Web Tokens.
- Chapter 4 presents gRPC interceptors and the updated centralized access control service.
- Chapter 5 covers changes to QuickFeed's integration with GitHub.
- Chapter 6 describes and explains other security-related changes made to the QuickFeed architecture as a result of our thesis work.
- Chapter 7 discusses challenges encountered in the course of this thesis work and alternative solutions for the problems addressed in this the previous chapters.
- Chapter 8 concludes the thesis.

Chapter 2

Background

2.1 Authentication and authorization

Identity and access management (IAM) is an important set of security policies. IAM aims to ensure that only users whose identities have been verified will receive access to the system. Another goal of IAM is to guarantee that only users belonging to certain user groups will be granted authority to access application services and perform actions changing the state of the system.

Two major parts of any IAM are authentication and authorization. Authentication is the process of verifying the user's identity. Authorization validates permissions and restrictions of an authenticated user to access different resources and perform actions.

Different approaches to authentication:

- single-factor (primary) authentication with username and password or PIN.
- two-factor authentication (2FA) with a password and one-time code.
- physical device-based authentication with a smart card or a USB device.
- biometric authentication with a face or fingerprint scan.
- certificate-based authentication with digital certificates issued by a certification authority.
- Single sign-on (SSO) where a single set of credentials can be used to access multiple independent applications and services. The credentials are issued

by a trusted source (identity provider) and usually are connected to a single user's account, for example, a Google, Facebook, or GitHub account.

2.1.1 Password-based and passwordless authentication

Password-based authentication patterns are considered ”outdated” as they provide low levels of security and usability compared to the passwordless methods. A user who is required to have different passwords for each application or service is bound to reuse the same password over and over again, compromising its security.

Purely passwordless approaches like biometric or hardware-based authentication have high security, as there is no password to be stolen or kept by the user in an insecure manner. But usability is lower as the user is supposed to keep a certain item like a smart card available at all times. This is a reasonable approach for organization-level authentication processes but impractical for more general use.

Single sign-on (SSO) authentication offers the best trade-off between security and usability as long as the identity provider can be trusted [2]. The password is still needed to sign in to the identity provider account. However, the service providers that the user can access with this method do not know or store the password to the main account.

2.2 Single sign-on

With SSO a single digital identity can be used to sign in to multiple independent service providers [2].

Advantages of SSO:

- Simple (and cleaner) system design: ”outsourced” authentication can be decoupled from authorization.
- Better security: user credentials don't need to be stored (or even accessed) by applications other than the identity provider itself.
- Convenient for users: don't need to create and remember multiple passwords. There is also no need to enter the credentials when logging into

other service providers as long as the user is already logged into the identity provider system.

Modern SSO solutions rely on the idea of federated identity: a set of standards and agreements that make it possible to use and share a single user identity across multiple applications and systems.

Identity provider (IdP) is the system responsible for the creation and management of user identities. Popular IdPs are Google, Microsoft, Apple, Facebook, GitHub.

The three most known federated identity management protocols are Security Assertion Markup Language (SAML), OpenID Connect, and OAuth.

2.2.1 Security Assertion Markup Language (SAML)

SAML is a set of definitions designed for the exchange of security information like user credentials between online domains [3]. Its purpose is to provide single sign-on for enterprise consumers. There are three actors in the SAML protocol: the principal (the user who wants to log into a system), the service provider (the system the user is trying to log into), and the identity provider (the entity that knows or can verify the user's identity). Messages in SAML are XML-based.

2.2.2 OAuth 2.0

OAuth 2.0 is an authorization standard focused on granting a delegated access based on an access token to applications, APIs and remote services [4]. Its main purpose is authorization between applications. OAuth messages are in JSON format. Main actors in OAuth are resource owner (user or entity), OAuth provider (the entity hosting the resource), and OAuth consumer (the entity asking for authorization to use the resource in question) [5].

2.2.3 OpenID Connect

OpenID Connect adds the authentication layer on top of the OAuth 2.0 protocol [6]. Its purpose is to authenticate users into service provider systems with third-party identity providers. In the OpenID authentication workflow, the identity provider sends to the service provider a unique authorization code which later can be used to request the user's identity details.

2.3 Stateful and stateless authentication

After the identity of a user has been established, there is the problem of keeping this information such that the user could access different services of the same application multiple times without a need to re-establish the identity. There are two main approaches to this problem: stateful and stateless. In stateful authentication pattern server keeps sessions for authenticated users. In stateless pattern there is some kind of a bearer token with user credentials that is created during the authentication process and then passed with each user request to serve as a proof of the user's identity.

The standard approach to stateless authentication is using JSON WEB Tokens (JWTs) [7]. JWT is an open standard for the exchange of a JSON object with claims between applications and services. The data contained in claims is cryptographically signed for increased security. Structurally a JWT has three parts:

- Header specifies the token type and signing algorithm.
- Payload is a JSON object with claims. It contains information used in the authentication process to identify a user.
- Signature of concatenated Header and Payload encoded with Base64URL encoding with a secret key or RSA certificate to guarantee the token integrity.

2.4 Encryption

Encryption is used to conceal information in a way that it only can be revealed by the person in possession of a correct key [8].

2.4.1 Symmetric and asymmetric algorithms

A symmetric encryption algorithm (cipher) uses the same key to encode and decode. An asymmetric algorithm uses a secret key to encrypt and a public key to decrypt the information.

2.4.2 Message Authentication Codes

A Message Authentication Code (MAC) is a cryptographic checksum that is used to confirm the integrity of a message [8]. A MAC is produced by applying a signing algorithm with a secret key to the message. The resulting MAC can later be verified to ensure that the message has not been modified. A symmetric signing algorithm uses the same secret key to sign and verify a message. An asymmetric algorithm uses a private and public key pair. The message is signed with the private key while the public key can be used for verification.

Hash-based MAC (HMAC) is a symmetric signing algorithm based on cryptographic hash functions. As a result, its security relies on the underlying hash function [9].

Chapter 3

Token-based authentication

In this chapter, we explain changes introduced to components of the QuickFeed architecture in charge of authenticating users with external identity providers, maintaining user identity information internally, and performing user authorization.

Autograder has two APIs, one for user authentication, another for the server's business logic. In the first version of the project, both were http-based REST APIs. Later the general purpose API has been re-implemented as gRPC calls. However, the main purpose of the authentication API is to exchange requests with GitHub. GitHub does not support gRPC. Thus this API cannot be replaced with gRPC endpoints and has to stay as it is.

Keeping interoperability between the two APIs is essential as gRPC methods require user information gathered by the http endpoint handlers for authorization checks. One of the major goals of our thesis is redesigning the authentication process to better align with the rest of the QuickFeed architecture and, as a result, simplify how the two APIs are communicating with each other.

3.1 Current design: session-based authentication

The original Autograder uses sessions to keep track of the logged in users. Sessions are started and maintained in an http session store. However, only methods of the http API can access the store directly.

To ensure interoperability between gRPC and http APIs it is necessary to maintain a redundant key-value structure that maps IDs of user sessions to the IDs of

the corresponding users. An example structure is demonstrated in listing 3.1.

```
map[string]uint64{
    "MTY1MjcwNTY1M3xE...": 12,
    "MTY1MnjYxe4wCc6X...": 48,
    "MTYcJeX3WjnMxw2K...": 137,
    ...
}
```

Listing 3.1: An example key-value structure with session ID to user ID mappings.

Figure 3.1 illustrates the redundant session store kept to maintain communication between gRPC and http APIs.

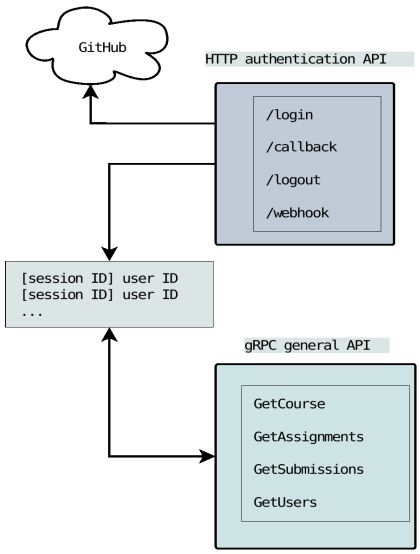

Figure 3.1: A redundant structure linking session ID to user ID ensures communication between APIs.

Session-based authentication pattern has several downsides. A session cookie only contains the ID of a session. Autograder acquires ID of the user associated with this session from the map structure. However, the ID itself does not give any information about the type of access the user can be granted. Details about user's roles in the Autograder system must be retrieved from the database.

As a result, each time a gRPC request is sent to the server, there will be a database lookup to perform an authorization check. This way each request-response completion time is limited to the speed of a database read operation.

Moreover, once created, a session will be kept in memory as long as the server is running even if the user is no longer active. There are two redundant structures

that store session-related information and both will grow in size and consume more memory as more and more users sign in.

Finally, there are scalability issues. Sessions are not directly compatible with any kind of distributed architecture. It would be necessary to maintain a separate session store for each instance of the server or each service and, possibly, solve the synchronization problem.

3.2 New design: token-based authentication

We replace the Autograder's session-based authentication process with stateless JSON Web Tokens (JWTs).

A JWT consists of three parts, as illustrated in figure 3.2: a header, a payload and a signature.

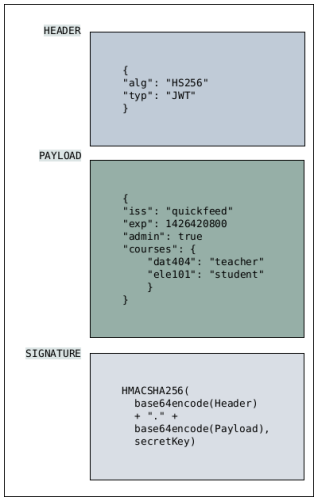

Figure 3.2: JSON Web Token structure.

The header has information about the token type and signing method.

The payload of a JWT has a set of mandatory fields that store essential information about the token, such as issuer or expiration time. A JWT can also include custom user claims that can be adjusted to store and pass around necessary information about the user, such as, for example, admin role or ID.

The signature contains base64 encoded header and payload concatenated with the secret key separated with dots and signed.

The flexibility of JWT claims can be utilized to perform server-side user authorization without any additional database lookups. This can speed up client-server interactions when compared to the session-based solution, making web-based user interface to appear more responsive.

When using a token-based authentication pattern there is no need for redundant structures in the codebase or memory as JWTs are stateless and the server has no information about tokens that have been issued.

3.2.1 QuickFeed user claims

JWT claims are used to uniquely identify the currently logged in user. Claims can contain any number of custom fields. Doing database look-ups on each request, which happens in the original implementation of the access control mechanism, essentially limits the request-response round-trip time to the database I/O speed. To allow Quickfeed server to perform user authorization checks without database queries, we include information about user privileges in the claims.

There are several roles in the Quickfeed system that determine what kind of data a user can access or what methods can be called from the client while the user is logged in.

A user can be an admin, a student enrolled in a certain course, or a member of the teaching staff in a certain course. We store this information in JWT claims, encode it, and set it as an http-only browser cookie.

QuickFeed JWT claims contain details about admin status and courses the user is enrolled in, as can be seen in listing 3.2.

```
type Claims struct {
  jwt.StandardClaims
  UserID  uint64                              `json:"user_id"`
  Admin   bool                                `json:"admin"`
  Courses map[uint64]pb.Enrollment_UserStatus `json:"courses"`
}
```

Listing 3.2: QuickFeed user claims.

To keep information about the status of a user in different courses we use a *map[uint64]pb.Enrollment_UserStatus* structure, where the course ID serves as a key, and the status of the user as a value. The *UserStatus* enumerated type of the *Enrollment* message serves as identifier of the enrollment status of a user. Implementation of the status is presented in listing 3.3.

```
message Enrollment {
    enum UserStatus {
        NONE    = 0;
        PENDING = 1;
        STUDENT = 2;
        TEACHER = 3;
```

Listing 3.3: QuickFeed enrollment status

3.2.2 Token security

A JWT uniquely identifies a QuickFeed user. The access control module trusts the information about the user stored in the JWT claims. Therefore, it is important to ensure that the token is used in a secure, reliable manner.

We set up several security measures to restrict access to the JWT cookie and ensure the authenticity and integrity of a token [10] [11].

To create a new JWT, the authentication module requests information about the user's roles in the system from the database. This information is encapsulated inside new JWT claims. The JWT is then signed and set as a secure browser cookie as illustrated in figure 3.3. Each following http or gRPC request from the authenticated client will also include this cookie.

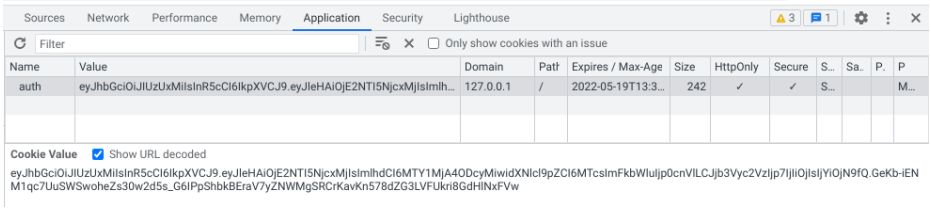

Figure 3.3: Session cookie in the browser with encoded JWT string.

We explicitly enable the *Secure* cookie option. A secure cookie will never be sent over an unprotected http connection, only over https, which makes a man-in-the-middle attack much harder to perform.

A man-in-the middle is a type of attacks where an unauthenticated person attempts to intercept a token cookie sent in a client-server request to get access to a service by impersonating the bearer of the stolen token.

Moreover, *HttpOnly* cookie field is also enabled making the cookie inaccessible to the JavaScript API. This helps to mitigate a possible cross-site scripting (XSS) attack where a malicious script is injected into a client. In this case a script will not be able to access the authentication cookie.

When a request from the client reaches the server, the signed token will be decoded and verified before the request is allowed to reach the intended service. Failed verification indicates that the content of a token has been altered and will result in the immediate denial of access. This guarantees that the contents of a JWT has not been altered by an attacker as long as the security of the signing key hasn't been compromised.

A token is signed to further prevent man-in-the-middle type of attacks where an attacker could intercept a token and change its content. Signature allows the server to verify that there were no changes to any parts of the token [11]. There are multiple signing algorithms that can be used to sign a token. Choosing an algorithm is an important decision, as there is always a tradeoff between security and speed of an algorithm.

In case of Quickfeed it seems sufficient to use a symmetric algorithm, as it is the same service that sings and verifies tokens and thus there is no need to share the signing key between parties. Therefore an asymmetric algorithm with a private and public key pair would only add complexity of managing two keys without contributing to the security of the application.

We chose hash-based message authentication code (HMAC) to sign Quickfeed authentication JWTs [9]. The security of a MAC signature depends on the cryptographic strength of the underlying hash function [8]. We will use SHA-256 hashing algorithm from the SHA-2 family [12].

SHA-256 is considered secure enough as with the modern level of technology it is resistant to collisions and exhaustive brute-force searches [13] [8]. However, compared to the alternatives like SHA-512, SHA-256 produces a shorter output that results in lower overhead when sent over the network. It is also more widely supported then the alternatives in the SHA-2 family.

In addition, HMAC offers better performance when signing and verifying compared to the asymmetric signing algorithms and is easier to use as there is only one key to consider.

When validating a token signature it is also important to verify that the field indicating the signing method hasn't been altered [14]. We perform the check when we parse the JWT cookie received with the request to the server, as shown in listing 3.4.

Finally, tokens have limited lifetime and expire after 15 minutes. A short lifetime aims to reduce the period of time during which a potentially stolen token can

```
if token.Header["alg"] != hashFunction {
    return nil, fmt.Errorf("incorrect hash function in the header:
        expected %s, got %s", hashFunction, token.Header["alg"])
  }
```

Listing 3.4: Verifying the correct signing method.

be used by an unauthorized attacker to access the service.

3.2.3 Token manager

Several methods are involved in management of JWTs. We need to create new tokens with user's data retrieved from the database. We need to update tokens that have expired or about to expire. We also need to validate tokens that are sent with client requests.

To keep all token-related types and operations in one place we develop a token manager structure and a set of methods that can be called on an instance of this manager.

```
type TokenManager struct {
  tokens      []uint64
  db          database.Database
  expireAfter time.Duration
  secret      string
  domain      string
  cookieName  string
}
```

Listing 3.5: Token manager type struct.

The token manager is presented in listing 3.5. A new instance of the manager is created just once when the QuickFeed server starts. Later the manager can be accessed by other modules to issue or verify a JWT, modify a list of tokens that need an update, or check if the current JWT is in this list.

Table 3.1 explains the purpose of the fields in a token manager.

tokens	A list of IDs of users that need a token update.
db	Database.
expireAfter	Lifetime of a token.
secret	A key used to sign and verify tokens.
domain	Domain the server is running on.
cookieName	Name of the authentication cookie, currently "auth".

Table 3.1: Token manager fields.

It is important to add an extra check to ensure that secret and domain are not empty strings during the creation of a new manager, as shown in listing 3.6.

```
func NewTokenManager(db database.Database, expireAfter time.Duration, secret, domain string)
(*TokenManager, error) {
  if secret == "" || domain == "" {
    return nil, fmt.Errorf("failed to create token manager:
        missing secret (%s) or domain (%s)", secret, domain)
  }
```

Listing 3.6: Checking that all required fields are set.

The field *tokens* of a token manager is used to store a list of user IDs that need their JWT re-issued on the next connection to the server. This condition is indicated by a Boolean field in the *User* database table. This guarantees that information about token updates is persistent across server restarts or possible crashes.

In QuickFeed, Protobuf messages are saved directly to the database Listing 3.7 shows the User message that serves as the schema for the User table in the database.

```
message User {
    uint64 ID        = 1;
    bool isAdmin     = 2;
    string name      = 3;
    string studentID = 4;
    string email     = 5;
    string avatarURL = 6;
    string login     = 7;
    bool updateToken = 8;
```

Listing 3.7: Fields of the User message.

Methods that can be called on an instance of the token manager allow us to create and validate JWTs, construct authentication cookies, check whether the user's token needs an update, and, finally, add and remove entries from the list and update the corresponding database *User* entries.

The *Update* method is called when creating a new token manager at the server start. It collects IDs of all users in the database who have their *updateToken* field set to True, as illustrated in listing 3.8.

When issuing a new JWT, the token manager fills the standard JWT fields first, then constructs the new Quickfeed claims with relevant user roles out of the data fetched from the database.

```go
// Update fetches IDs of users who need token updates from the database
func (tm *TokenManager) Update() error {
  users, err := tm.db.GetUsers()
  if err != nil {
    return fmt.Errorf("failed to update JWT tokens from database: %w", err)
  }
  var tokens []uint64
  for _, user := range users {
    if user.UpdateToken {
      tokens = append(tokens, user.ID)
    }
  }
  tm.tokens = tokens
  return nil
}
```

Listing 3.8: Populating token update list from the database on the server start.

As shown in listing 3.9, we set token creation and expiration time and add information about the user's admin role and course enrollment status to the claims. *Audience* JWT field is not necessary now that we use the same token across the whole QuickFeed service. However, with multiple services, it is possible to limit access of a token bearer to a single service by setting this field and adding corresponding authorization checks.

```go
// NewClaims creates new JWT claims for user ID
func (tm *TokenManager) NewClaims(userID uint64) (*Claims, error) {
  usr, err := tm.db.GetUserWithEnrollments(userID)
  if err != nil {
    return nil, err
  }
  newClaims := &Claims{
    StandardClaims: jwt.StandardClaims{
      ExpiresAt: time.Now().Add(tm.expireAfter).Unix(),
      IssuedAt:  time.Now().Unix(),
      Issuer:    "Quickfeed",
    },
    UserID: userID,
    Admin:  usr.IsAdmin,
  }
  userCourses := make(map[uint64]pb.Enrollment_UserStatus)
  for _, enrol := range usr.Enrollments {
    userCourses[enrol.GetCourseID()] = enrol.GetStatus()
  }
  newClaims.Courses = userCourses
  return newClaims, nil
}
```

Listing 3.9: Creating new JWT claims.

It is also easy to log out a user by setting a new cookie with the same name and empty content.

Chapter 4

Interceptors

In this chapter, we cover the development of a set of gRPC interceptors that will perform authentication and authorization checks and replace the original http middleware.

Http middleware and gRPC interceptors are very similar concepts [15]. Both are API wrappers that aim to intercept and inspect a client-server communication before a request reaches the server. The goal of such interception is to apply some additional logic to multiple API methods. A typical use of middleware or interceptors is logging, different validation checks, and user authentication.

4.1 Middleware

Http middleware wraps a group of API endpoints and inspects every request that is sent to one of these endpoints.

Middleware in the original Autograder has been designed to support the http API architecture. The middleware was intended for internal authentication checks. It would verify that there is an active session for the current user and that a record for this user exists in the database.

There are currently two middleware methods. The *PreAuth* method was intended only for the API group in charge of user authentication with GitHub. The *AccessControl* middleware was designed to wrap all http methods including those that the *PreAuth* already takes care of.

Both middleware methods perform two identical checks, as can be seen in figure 4.1.

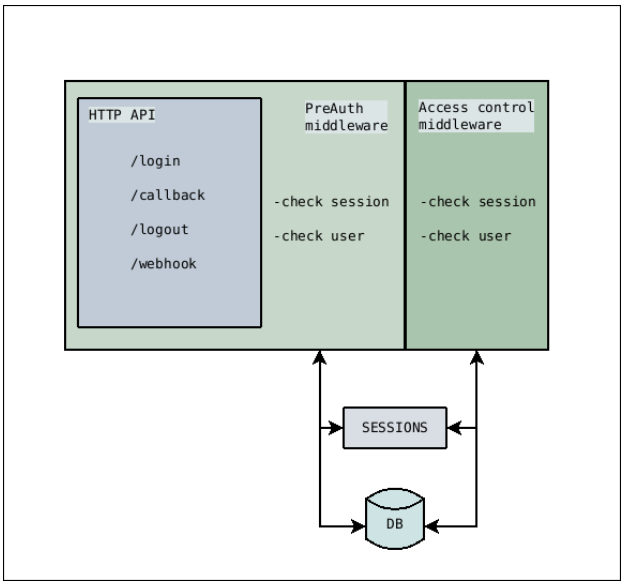

Figure 4.1: Two middleware functions performing two redundant checks.

At first, each middleware checks the session store to ensure that an active session for this user exists. If the session is found, the middleware then queries the database to make sure that there is a record for the user. If both checks are successful, the request is allowed to proceed to the Autograder server.

The main difference between the two middleware methods is in the way they handle a request from a new user. The *PreAuth* method would start a new session and redirect the unauthenticated user to GitHub to initiate a new sign in. The *AccessControl* however would just immediately deny an unauthorized user access to the server.

Now that the general purpose API has been reimplemented as gRPC calls it never invokes the *AccessControl* middleware. It is only called together with the *PreAuth* middleware when a user logs in with one of sign in buttons in the Autograder's user interface, as shown in figure 4.2

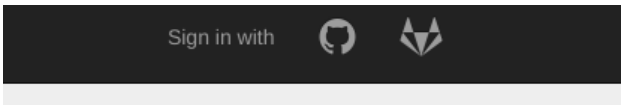

Figure 4.2: Login options: GitHub and GitLab.

As a result, the *AccessControl* middleware is already redundant and can be safely removed. Now that there is no sessions, the *PreAuth* middleware no longer serves a purpose. We completely remove the Autograder's http middleware and transfer its intended authentication responsibilities to gRPC interceptors.

4.2 Interceptors

Just like the http middleware, interceptors inspect each gRPC API request before deciding whether it can be allowed to proceed to the server.

Autograder already has two interceptors. The first one inspects gRPC message in the request payload to ensure that the required fields of the message are not empty. We leave this interceptor unchanged.

The second interceptor, *UserVerifier*, assists in communication between http and gRPC APIs. It reads the ID of a user session from the session cookie and fetches the ID of the user from the structure that links session and user IDs. *UserVerifier* then sets the user ID in the gRPC metadata for internal consumption by gRPC methods.

Now that the Quickfeed's authentication process does not rely on sessions, this interceptor becomes unnecessary. We replace it with a pair of new interceptors designed for the token-based authentication pattern.

Finally, we replace the per-method access control performed inside of each API method with another interceptor to completely prevent unauthorized requests from reaching the server.

The complete interceptor chain is presented in figure 4.3

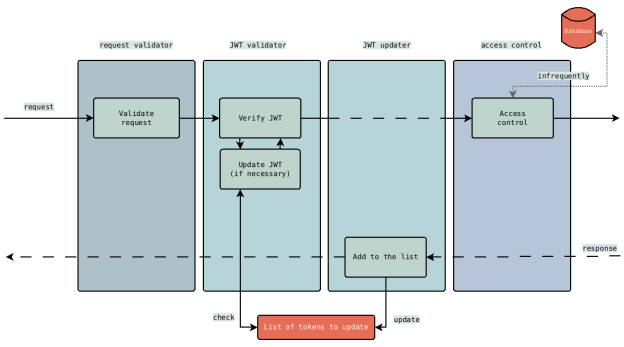

Figure 4.3: Architecture of the interceptor chain.

4.3 Authentication interceptors

In a token-based authentication pattern, it is important to validate the integrity of a token. There are also situations where an already issued JWT needs an update because it has outdated information or is about to expire.

We implement two authentication interceptors to manage issuing, updating, and validating JWTs.

4.3.1 Validating tokens

Once a user has authenticated with GitHub account, QuickFeed will create a new JWT with user information. A token will then be set as a safe browser cookie and later sent to the server with each following request.

To ensure that information inside a JWT has not been changed, we sign the token following the JWS standard before sending it to the client [16]. The dedicated *ValidateToken* interceptor method will then verify the token signature on each request to the server. The interceptor method is outlined in listing 4.1.

```
func ValidateToken(logger *zap.SugaredLogger, tokens *auth.TokenManager) grpc.UnaryServerInterceptor {
  return func(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler)
  (interface{}, error) {
    start := time.Now()
    token, err := GetFromMetadata(ctx, "cookie", tokens.GetAuthCookieName())
    if err != nil {
      logger.Error(err)
      return nil, ErrAccessDenied
    }
    claims, err := tokens.GetClaims(token)
    if err != nil {
      logger.Errorf("Failed to extract claims from JWT: %s", err)
      return nil, ErrAccessDenied
    }
```

Listing 4.1: Interceptor for token validation.

The interceptor calls a token manager method, *GetClaims*.

As can be seen in listing 4.2, inside this method the token is decoded and the signature is verified with an additional check to make sure that a correct signing algorithm is declared in the token.

This step aims to circumvent a critical vulnerability in JWT spec [14]. A JWT header includes a string that indicates the algorithm used to sign the token. However, an attacker can change the value to ”none”. As a result, a JWT with any content constructed by the attacker would always be recognized as valid.

```go
// GetClaims returns user claims after parsing and validating a signed token string
func (tm *TokenManager) GetClaims(tokenString string) (*Claims, error) {
  token, err := jwt.Parse(tokenString, func(t *jwt.Token) (interface{}, error) {
    if t.Header["alg"] != alg {
      return nil, fmt.Errorf("incorect signing algorithm, expected %s, got %s", alg, t.Header["alg"])
    }
    if _, ok := t.Method.(*jwt.SigningMethodHMAC); !ok {
      return nil, fmt.Errorf("failed to parse token: incorrect signing method")
    }
    return []byte(tm.secret), nil
  })
  if err != nil {
    return nil, err
  }
  if claims, ok := token.Claims.(*Claims); ok && token.Valid {
    return claims, nil
  }
  return nil, fmt.Errorf("failed to parse token: validation error")
}
```

Listing 4.2: Token validation when parsing a JWT.

The expected hashing algorithm is set as a package constant and verified on each request.

4.3.2 Working with gRPC metadata

As can be seen in listing 4.1, the token validator interceptor extracts the encoded and signed token string from the gRPC metadata.

Metadata is a nested dictionary sent with each gRPC request. All browser cookies are stored in the ”cookie” field of the metadata, as shown in figure 4.4.

- :authority: 127.0.0.1:8080
- accept:[application/grpc-web-text]
- accept-encoding:[gzip, deflate, br]
- accept-language:[en-US,en;q=0.9,da;q=0.8,no;q=0.7]
- content-type:[application/grpc]
- cookie:[auth=djwiegpwjmgsegmjwemgfowpe3it2fjpe]
- origin:[https://127.0.0.1:8080]
- referer:[https://127.0.0.1:8080/app/home]
- sec-ch-ua:[" Not A;Brand";v="99", "Chromium";v="101"]
- sec-ch-ua-mobile:[?0]
- sec-ch-ua-platform:["Linux"]
- sec-fetch-dest:[empty]
- sec-fetch-mode:[cors]
- sec-fetch-site:[same-origin]
- user-agent:[Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/5 37.36 (KHTML, like Gecko) Chrome/101.0.4951.41 Safari/537.36]
- x-grpc-web:[1]
- x-user-agent:[grpc-web-javascript/0.1]]

Figure 4.4: Example of metadata header sent with a gRPC request.

```go
// GetFromMetadata extracts a value from a filed of incoming metadata
// by the given key. Used to extract JWT tokens.
func GetFromMetadata(ctx context.Context, field, key string) (string, error) {
  if field == "" {
    return "", fmt.Errorf("missing metadata field name (%s)", field)
  }
  meta, ok := metadata.FromIncomingContext(ctx)
  if !ok {
    return "", fmt.Errorf("failed to read metadata")
  }
  content := meta.Get(field)
  // if there is no key, a field is expected to have only one element
  if key == "" {
    if len(content) != 1 {
      return "", fmt.Errorf("incorrect metadata content length: %d", len(content))
    }
    return content[0], nil
  }
  for _, c := range meta.Get(field) {
    _, content, ok := strings.Cut(c, key+"=")
    if !ok {
      return "", fmt.Errorf("missing %s cookie", key)
    }
    return strings.TrimSpace(content), nil
  }
  return "", fmt.Errorf("missing metadata field %s", field)
}
```

Listing 4.3: Helper method to extract values from gRPC metadata.

We implement a new helper method, *GetFromMetadata*, to simplify retrieving metadata information, as demonstrated in listing 4.3.

Two other helper methods are used to set new metadata values. The *setToMetadata* method adds a given value to the inbound context for internal consumption by the gRPC API. The *setCookie* method adds a new JWT cookie to the outbound context when a response is sent back to the web client.

4.3.3 Updating tokens

The major downside of JWT is that, once a token has been generated and set as a browser cookie, it is nearly impossible to revoke it. However, it might be necessary if the user details in JWT claims are no longer correct.

For example, claims can be issued for a non-admin user who afterwards is promoted to admin. As a result, if the user sends another request before the user's JWT has expired, he will be treated as a non-admin despite the promotion. Even worse, if an admin user gets demoted, he will still have access to admin services as long as the JWT with admin claims is still valid.

However, there exists an easy solution to this problem. There is a limited set of methods that can change the role of a user. To keep track of such changes, the token manager maintains a list of users whose token needs an update because the

```
type TokenManager struct {
  tokens      []uint64
```

Listing 4.4: Token update list.

access privileges have changed and information in the JWT claims is no longer correct. This list is stored in a field of a token manager instance, as can be seen in listing 4.4.

In addition, each user record gets a new Boolean field *updateToken* which indicates whether a new JWT has to be generated for the user next time they send a request to the server. The updated user message is shown in listing 4.5. This field in a database record guarantees that the list is persistent across server restarts of possible failures.

```
message User {
    uint64 ID        = 1;
    ...
    string avatarURL = 6;
    string login     = 7;
    bool updateToken = 8;
```

Listing 4.5: New field in the User message.

When a new instance of the token manager is created when the Quickfeed server starts, the list will be populated with user IDs.

As shown in listing 4.6, the new method calls the *Update* function before returning a new instance of the token manager.

```
// NewTokenManager creates a new token manager, populating
// the token list with user IDs from the database
func NewTokenManager(db database.Database, expireAfter time.Duration, secret, domain string)
(*TokenManager, error) {
    ...
  // Collect IDs of users who require token update from database.
  if err := manager.Update(); err != nil {
    return nil, err
  }
  return manager, nil
}
```

Listing 4.6: Creating a new instance of token manager.

As can be seen in listing 4.7, if a user has *UpdateToken* field set to True, the ID of this user will be added to the list. The method is called only once when the QuickFeed server starts.

We also implement three other token manager methods that help synchronize the database with the token list.

```go
// Update fetches IDs of users who need token updates from the database
func (tm *TokenManager) Update() error {
  users, err := tm.db.GetUsers()
  if err != nil {
    return fmt.Errorf("failed to update JWT tokens from database: %w", err)
  }
  var tokens []uint64
  for _, user := range users {
    if user.UpdateToken {
      tokens = append(tokens, user.ID)
    }
  }
  tm.tokens = tokens
  return nil
}
```

Listing 4.7: Populating token list.

```go
// Add adds a new UserID to the manager and updates user record in the database
func (tm *TokenManager) Add(userID uint64) error {
  if tm.exists(userID) {
    return nil
  }
  if err := tm.update(userID, true); err != nil {
    return err
  }
  tm.tokens = append(tm.tokens, userID)
  return nil
}
```

Listing 4.8: Adding new ID to the token list.

The *Add* method shown in listing 4.8 puts a new user ID into the list. *Remove* in listing 4.9 deletes the ID from the list after a successful update.

```go
// Update removes user ID from the manager and updates user record in the database
func (tm *TokenManager) Remove(userID uint64) error {
  if !tm.exists(userID) {
    return nil
  }
  if err := tm.update(userID, false); err != nil {
    return err
  }
  var updatedTokenList []uint64
  for _, id := range tm.tokens {
    if id != userID {
      updatedTokenList = append(updatedTokenList, id)
    }
  }
  tm.tokens = updatedTokenList
  return nil
}
```

Listing 4.9: Removing ID from the list after a successful token update.

The *update* method will update the corresponding User record, as shown in listing 4.10.

Database lookups are often slow and we would like to avoid doing them for each request. However, updates to user roles are very infrequent. Most of the time the list will be empty.

```go
// update updates user record in the database
func (tm *TokenManager) update(userID uint64, updateToken bool) error {
  user, err := tm.db.GetUser(userID)
  if err != nil {
    return err
  }
  user.UpdateToken = updateToken
  if err := tm.db.UpdateUser(user); err != nil {
    return err
  }
  return nil
}
```

Listing 4.10: Updating a user record in the database.

4.3.4 Interceptor to update tokens

We implement the *UpdateTokens* interceptor that will manage the update list. The interceptor keeps a list of the four methods that change user roles in the QuickFeed system. The list is shown in listing 4.11.

```go
var methods = []string{
  "UpdateUser",
  "CreateCourse",
  "UpdateEnrollments",
  "UpdateGroup",
}
```

Listing 4.11: List of methods that change privileges of a user.

The methods and the way they can affect user privileges are described in table 4.1.

UpdateUser	Called by admin to grant or revoke admin status.
CreateCourse	User creating a course is granted full access to the course information.
UpdateEnrollments	User enrolled into a new course or promoted to TA.
UpdateGroup	Adds or removes group members.

Table 4.1: List of methods that affect roles of a user in the QuickFeed system.

The interceptor ignores requests that invoke methods not included in the list. If a listed method is going to be invoked, the interceptor lets the request through and inspects the response.

If a listed method returns an error, it is also ignored. Only if the request was successful the interceptor will add the relevant user IDs to the token update list.

The *UpdateTokens* interceptor is presented in listing 4.12.

```go
// UpdateTokens adds relevant user IDs to the list of users that need their token refreshed
// next time they sign in because their access roles might have changed
// This method only logs errors to avoid overwriting the gRPC responses.
func UpdateTokens(logger *zap.SugaredLogger, tokens *auth.TokenManager)
        grpc.UnaryServerInterceptor {
  return func(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler)
      (interface{}, error) {
    sort.Strings(methods)
    method := info.FullMethod[strings.LastIndex(info.FullMethod, "/")+1:]
    idx := sort.SearchStrings(methods, method)
    if idx < len(methods) && methods[idx] == method {
      resp, err := handler(ctx, req)
      // We only want to add the user ID to the list of tokens to update
      // if the request was successful.
      if err == nil {
        switch method {
        // User has been promoted to admin or demoted.
        case "UpdateUser":
          // Add id of the user whose info has been updated.
          if err := tokens.Add(req.(*pb.User).GetID()); err != nil {
            logger.Error(err)
          }
        // The signed in user gets a teacher role for the new course.
        case "CreateCourse":
          token, err := GetFromMetadata(ctx, "token", "")
          if err != nil {
            logger.Error(err)
          }
          claims, err := tokens.GetClaims(token)
          if err != nil {
            logger.Error(err)
          }
          if err := tokens.Add(claims.UserID); err != nil {
            logger.Error(err)
          }
        // Users has been enrolled into a course or promoted to TA.
        case "UpdateEnrollments":
          for _, enrol := range req.(*pb.Enrollments).GetEnrollments() {
            if err := tokens.Add(enrol.GetUserID()); err != nil {
              logger.Error(err)
            }
          }
        // Group is approved or modified.
        case "UpdateGroup":
          for _, user := range req.(*pb.Group).GetUsers() {
            if err := tokens.Add(user.GetID()); err != nil {
              logger.Error(err)
            }
          }
        }
      }
      return resp, err
    }
    return handler(ctx, req)
  }
}
```

Listing 4.12: Interceptor for token updates.

4.3.5 Interceptor to validate tokens

To validate JWTs in client requests and issue new tokens in case of an update we design another interceptor.

The *ValidateToken* interceptor checks whether the user who sends a request to the server needs a new JWT. If it is the case, the updated user information is requested from the database, and a new token is generated and set as a new token cookie. The interceptor is presented in listing 4.13.

```
// ValidateToken validates the integrity of a JWT in each request. It will also create and set a new JWT
// if the current token is in the update list or about to expire.
func ValidateToken(logger *zap.SugaredLogger, tokens *tokens.TokenManager) grpc.UnaryServerInterceptor {
  return func(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler)
      (interface{}, error) {
    token, err := GetFromMetadata(ctx, "cookie", tokens.GetAuthCookieName())
    if err != nil {
      logger.Error(err)
      return nil, ErrAccessDenied
    }
    claims, err := tokens.GetClaims(token)
    if err != nil {
      logger.Errorf("Failed to extract claims from JWT: %s", err)
      return nil, ErrAccessDenied
    }
    // If the token is about to expire or the user ID
    // is in the update token list, generate and set a new JWT.
    if tokens.UpdateRequired(claims) {
      updatedToken, err := tokens.NewTokenCookie(claims.UserID)
      if err != nil {
        logger.Errorf("Failed to generate new user claims %v", err)
        return nil, ErrAccessDenied
      }
      if err := tokens.Remove(claims.UserID); err != nil {
        logger.Errorf("Failed to update token list: %s", err)
        return nil, ErrAccessDenied
      }
      if err := setCookie(ctx, updatedToken.String()); err != nil {
        logger.Errorf("Failed to set auth cookie: %s", err)
      }
      token = updatedToken.Value
    }

    ctx, err = setToMetadata(ctx, "token", token)
    if err != nil {
      logger.Error(err)
      return nil, ErrAccessDenied
    }
    return handler(ctx, req)
  }
}
```

Listing 4.13: Token validator interceptor.

The interceptor calls the *UpdateRequired* token manager method presented in listing 4.14. The method returns True if the user ID is in the token update list, or the JWT has expired or is about to expire.

```go
// JWTUpdateRequired returns true if JWT update is needed for this user ID
// due to updated user role or token expiration time
func (tm *TokenManager) UpdateRequired(claims *Claims) bool {
  for _, token := range tm.tokens {
    if claims.UserID == token {
      return true
    }
  }
  if claims.ExpiresAt - time.Now().Unix() < refreshTime.Milliseconds() {
    return true
  }
  return false
}
```

Listing 4.14: Checking whether the JWT needs an update.

Finally, we define a new error type. The *ErrAccessDenied* error will be returned if a check performed by any interceptor fails. This way we can avoid redundant code in error handlers. The new error can be seen in listing 4.15.

```go
var ErrAccessDenied = status.Errorf(codes.Unauthenticated, "access denied")
```

Listing 4.15: Custom error to return when access to the server is denied.

4.4 Authorization interceptor

Quickfeed API methods request information about users, user groups, and courses from the database. Authorization is the process of confirming that the user calling the API method has permission to access this specific information. A student shouldn't be able to request or update the data of another student. Likewise, a course teacher shouldn't be able to access assignments or a list of members of another course.

In the original Autograder's authorization architecture, access control is performed on a per-method basis inside each API method. As a result, a client request always reaches the server, even when the user sending the request is not authorized to.

An example of an access control check in the original Autograder is presented in listing 4.16

The *UpdateUser* method shown in the listing can be called by a user to update their information like student number or e-mail. It also can be called by an admin to grant or revoke admin status. Because of that, there is a check that ensures that the user is either an admin or that the user record that will be updated belongs to the same user. Only if one of the checks passes the method can advance.

```go
// UpdateUser updates the current users's information and returns the updated user.
// This function can also promote a user to admin or demote a user.
// Access policy: Admin can update other users's information and promote to Admin;
// Current User if Owner can update its own information.
func (s *AutograderService) UpdateUser(ctx context.Context, in *pb.User) (*pb.Void, error) {
  usr, err := s.getCurrentUser(ctx)
  if err != nil {
    s.logger.Errorf("UpdateUser failed: authentication error: %v", err)
    return nil, ErrInvalidUserInfo
  }
  if !(usr.IsAdmin || usr.IsOwner(in.GetID())) {
    s.logger.Errorf("UpdateUser failed to update user %d: user is not admin or course creator", in.GetID())
    return nil, status.Error(codes.PermissionDenied, "only admin can update another user's information")
  }
  if _, err = s.updateUser(usr, in); err != nil {
    s.logger.Errorf("UpdateUser failed to update user %d: %v", in.GetID(), err)
    err = status.Error(codes.InvalidArgument, "failed to update user")
  }
  return &pb.Void{}, err
}
```

Listing 4.16: Access control checks inside an API method.

This implementation makes it difficult to keep track of privileges required by each method as this information is only stated in the comments.

We reassign the access control process to a dedicated gRPC interceptor.

A special enumerated type describes different roles that can limit what kinds of data a user is allowed to access. The new type can be seen in listing 4.17.

```go
type (
  role      int
  roles     []role
)
```

Listing 4.17: New role types.

There are several well-defined user roles in the QuickFeed system, as presented in listing 4.18.

```go
const (
  // user role implies that user attempts to access information about himself.
  user role = iota
  // group role implies that the user is a course student + a member of the given group.
  group
  // student role implies that the user is enrolled in the course with any role.
  student
  // teacher: user enrolled in the course with teacher status.
  teacher
  // courseAdmin: an admin user who is also enrolled into the course.
  courseAdmin
  // admin is the user with admin privileges.
  admin
)
```

Listing 4.18: Access control user roles.

A *user* can only access or modify their own information. The *group* role indicates that the user is a member of a certain student group. A *student* has read access to the course information and own grades. A *teacher* has read and write

access to the course information. A *courseAdmin* is a QuickFeed admin who is also enrolled in a certain course but not necessary as a teacher. An *admin* can promote new admins and create new courses.

All the restricted API methods and required roles are now stored in a single structure that is displayed in listing 4.19.

```
// If there are several roles that can call a method, a role with the least privilege must come first.
// If method is not in the map, there is no restrictions to call it.
var access = map[string]roles{
  "GetEnrollmentsByCourse":  {student, teacher},
  "UpdateUser":              {user, admin},
  "GetEnrollmentsByUser":    {user, admin},
  "GetSubmissions":          {user, group, teacher, courseAdmin},
  "GetGroupByUserAndCourse": {group, teacher},
  "CreateGroup":             {group, teacher},
  "GetGroup":                {group, teacher},
  "UpdateGroup":             {teacher},
  "DeleteGroup":             {teacher},
  "IsEmptyRepo":             {teacher},
  "GetGroupsByCourse":       {teacher},
  "UpdateCourse":            {teacher},
  "UpdateEnrollments":       {teacher},
  "UpdateSubmission":        {teacher},
  "RebuildSubmissions":      {teacher},
  "CreateBenchmark":         {teacher},
  "UpdateBenchmark":         {teacher},
  "DeleteBenchmark":         {teacher},
  "CreateCriterion":         {teacher},
  "UpdateCriterion":         {teacher},
  "DeleteCriterion":         {teacher},
  "CreateReview":            {teacher},
  "UpdateReview":            {teacher},
  "UpdateSubmissions":       {teacher},
  "GetReviewers":            {teacher},
  "UpdateAssignments":       {teacher},
  "GetSubmissionsByCourse":  {teacher, courseAdmin},
  "GetUserByCourse":         {teacher, admin},
  "GetOrganization":         {admin},
  "CreateCourse":            {admin},
}
```

Listing 4.19: Restricted API methods and roles.

Methods that are not in the list have no restrictions and can be called by any user.

The roles are ordered according to the Principle of Least Privilege, lowest to highest access level. The reasoning behind this order is that there are significantly more students than teachers using the QuickFeed application at any moment, and there are even fewer admins.

Therefore it is better to start the case switch with more frequent roles, as there is a higher chance that the first case will result in a hit. This can improve the average time the access control interceptor takes before it lets the request with sufficient authority proceed to the server.

When inspecting a request from an interceptor, the type of the request message can only be guessed from the method signature. However, we need to access some ID values inside the message to perform the authorization check.

For example, the message can contain the ID of a course and we want to make sure that the calling user is enrolled in the course as a student or teacher. Likewise, we might want to retrieve the IDs of a user or a group from the request.

To simplify the process of extracting a correct ID from an unknown message type, we add a dedicated interface that all the relevant message types will implement.

```
idRequest interface {
  FetchID(string) uint64
}
```

Listing 4.20: New interface.

The interface shown in listing 4.20 must implement a method that takes the role string and returns the corresponding ID. There is no need to know exactly the message type as long as it implements the *FetchID* method.

```
// FetchID returns user, group, or course ID
func (r *SubmissionRequest) FetchID(role string) uint64 {
  switch role {
  case "user":
    return r.GetUserID()
  case "group":
    return r.GetGroupID()
  case "course":
    return r.GetCourseID()
  }
  return 0
}
```

Listing 4.21: Example of FetchID method

Listing 4.21 demonstrates an example implementation of the *FetchID* interface method for the *SubmissionRequest* message type.

There are three different types of IDs we might need to extract. If a user wants to access an individual submission we need to extract the user's ID from the message to compare it to the ID of the sender.

In the same manner, we want the ID of the group when a group submission is requested to confirm that the sender is indeed a member of this group.

Finally, a teacher can request all submissions for the course. In this case, we want to fetch the ID of the course and verify that the sender is enrolled in the course as a teacher.

As a result, we can now retrieve the correct type of ID from a request based on the expected user role, as demonstrated in listing 4.22.

```
switch role {
  case user:
    if m, ok := req.(requestID); ok {
      if m.FetchID("user") == claims.UserID {
        return handler(ctx, req)
      }
    } else {
      logger.Debugf("Method %s does not implement FetchID method", method)
    }
  case student:
    if m, ok := req.(requestID); ok {
      courseID := m.FetchID("course")
      status, ok := claims.Courses[courseID]
      if ok && status == pb.Enrollment_STUDENT {
        return handler(ctx, req)
      }
    } else {
      logger.Debugf("Method %s does not implement FetchID method", method)
    }
  case admin:
    if claims.Admin {
      return handler(ctx, req)
    }
    ...
```

Listing 4.22: Access control checks for different user roles.

Now that all the restricted API methods and expected user roles are collected in one place, it is easy to identify which user groups can call a certain method. It also should be easier to maintain and update the access control logic.

Chapter 5

GitHub integration

In this chapter, we present changes we introduce to the components of the QuickFeed architecture in control of communication with GitHub.

Integration with GitHub constitutes a significant part of the QuickFeed functionality. A QuickFeed user is based on a GitHub account. A QuickFeed course is connected to a GitHub organization. Student enrolled in a course get personal repositories to submit their solutions to coding assignments. A student group is a GitHub team.

To submit a solution to an assignment, students push the solution code to personal or group repositories. After each new push, QuickFeed receives an event notification from GitHub. QuickFeed then downloads, builds and tests the submitted code and returns feedback to the students.

There are three distinct components of the QuickFeed architecture responsible for communication with GitHub. The **web/auth** package supports user authentication with GitHub accounts. The **scm** package is responsible for general GitHub API calls. The **web/hooks** package provides support for the GitHub event subscription mechanism, webhooks. Changes introduced by this thesis only affect the first two modules. We leave the webhook functionality unchanged.

Additionally, Quickfeed offers some limited support for GitLab API and sign in. Integration with GitLab is beyond the scope of this thesis. However, we make sure that the proposed solution can be easily made compatible with GitLab in the future.

5.1 Single sign on with GitHub

The single sign on (SSO) authentication pattern allows a user to sign in to multiple independent service providers with the same ID from a trusted identity provider. In QuickFeed, users log in using their GitHub accounts.

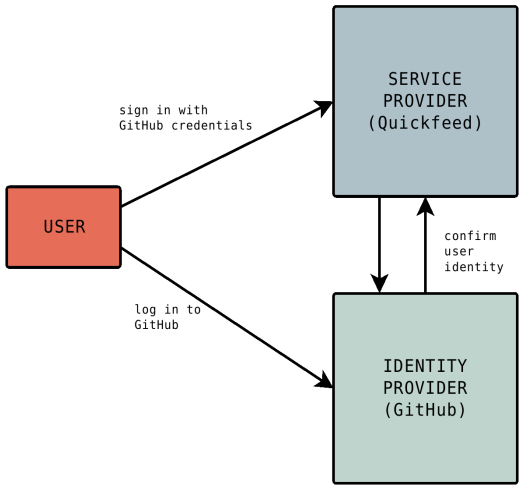

Figure 5.1: Single Sign On with a GitHub account.

QuickFeed uses GitHub OAuth 2.0 protocol to authenticate users and authorize the application to perform a limited set of actions on the user's behalf. Quickfeed serves as the Service Provider in the OAuth scheme, while GitHub is the identity provider, as illustrated in figure 5.1.

The message exchange between the user, QuickFeed, and GitHub is outlined in figure 5.2. An unauthorized user who attempts to sign in to QuickFeed will be redirected to the GitHub log in page. Note that if the user is already logged in to GitHub account, the redirect happens in the background and does not affect the web client interface at all.

GitHub then redirects back to QuickFeed with a unique identification code. QuickFeed exchanges the code to a personal access token and then uses the token to verify user information with GitHub.

When the identity of the user is confirmed, QuickFeed creates a new JSON Web Token (JWT) with user details and sets it as a browser cookie. Next time the

user client sends a request to the QuickFeed server, the JWT cookie is sent along and the user is recognized as authenticated by the QuickFeed API.

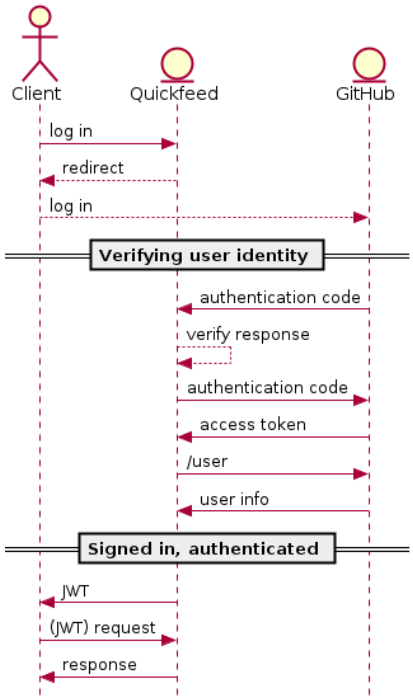

Figure 5.2: Authentication flow with GitHub.

5.2 GitHub integration: OAuth app

GitHub offers two integration options for applications like QuickFeed: an OAuth app and a GitHub app [17]. When the original Autograder was designed, OAuth apps were the only solution. Thus the Autograder's GitHub integration is implemented as an OAuth app.

The main downside of an OAuth app is that its functionality is limited to user authentication. In order to access the GitHub API, it was necessary to store users' personal access tokens in the database. Autograder would create GitHub API clients based on these access tokens in order to access and update course information on GitHub.

Storing and sending tokens over the network is problematic. If the database file is compromised or a request to GitHub is intercepted by an attacker, an access

token can be used to impersonate the user to get access to GitHub services.

Moreover, because the OAuth app is only responsible for authentication with GitHub, and the Autograder's **scm** module that communicates with GitHub API must use access tokens, these two GitHub-related components of the application architecture are completely disconnected, as demonstrated in figure 5.3.

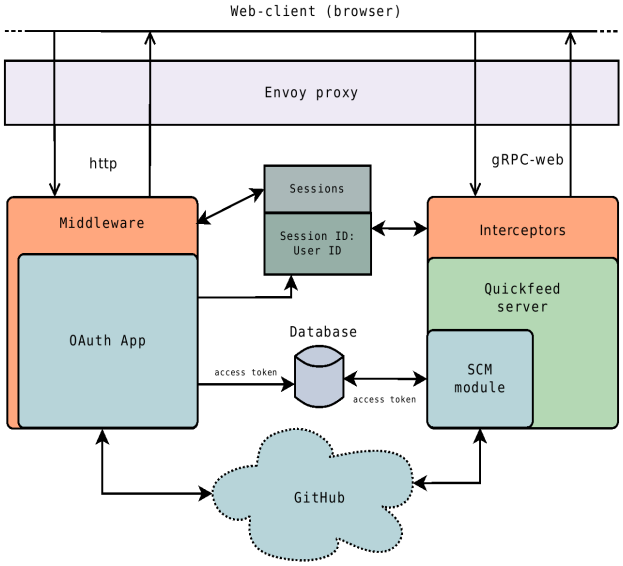

Figure 5.3: QuickFeed architecture with OAuth app.

We propose replacing the OAuth app in the Autograder's architecture with another, more modern integration option, a GitHub app.

5.3 GitHub integration: GitHub app

GitHub apps support the OAuth 2.0 authentication pattern just like OAuth apps. But in addition, a GitHub app can be installed on a course organization and granted read or write permissions to the organization's content. When installed, a GitHub app can invoke GitHub API on this organization directly, without personal access tokens.

To support the new GitHub integration type, we expand the QuickFeed's **scm** package with several new methods and types. SCM is an abbreviation for Source Control Manager. An scm client is a client used to access GitHub or GitLab APIs.

We implement a new SCMMaker type to create and store scm clients. The new type is shown in listing 5.1. It is easy to add GitLab functionality in the future by simply adding an extra field with GitLab specific configuration.

```
type SCMMaker struct {
  scms         *Scms
  githubConfig *GithubConfig
}
```

Listing 5.1: SCMMaker is used to create and store scm clients.

Quickfeed creates an instance of SCMMaker and a set of clients for each active installation when the server starts by calling the *NewSCMMaker* method that can be seen in listing 5.2.

```
func NewSCMMaker() (*SCMMaker, error) {
  config := newAppConfig()
  if !config.Valid() {
    return nil, fmt.Errorf("error configuring GitHub App: %+v", config)
  }
  appKey, err := key.FromFile(config.keyPath)
  if err != nil {
    wd, _ := os.Getwd()
    return nil, fmt.Errorf("wd %s error reading key from file: %s", wd, err)
  }
  appClientConfig, err := app.NewConfig(config.appID, appKey)
  if err != nil {
    return nil, fmt.Errorf("error creating GitHub application client: %s", err)
  }
  config.appConfig = appClientConfig
  return &SCMMaker{
    githubConfig: config,
    scms:         NewScms(),
  }, nil
}
```

Listing 5.2: Creating a new instance of SCMMaker.

GitHub app parameters are stored in the *GithubConfig* structure presented in listing 5.3.

```
type GithubConfig struct {
  appID     string
  clientID  string
  secret    string
  keyPath   string
  appConfig *app.Config
}
```

Listing 5.3: GitHub app configuration.

Application ID, client ID, and secret are all generated by GitHub when a new app is created. These parameters must be provided as environmental variables together with a path to the file with the app's private key. Method to create a new instance of the configuration is shown in listing 5.4.

```
func newAppConfig() *GithubConfig {
  return &GithubConfig{
    appID:    os.Getenv(AppEnv),
    clientID: os.Getenv(KeyEnv),
    secret:   os.Getenv(SecretEnv),
    keyPath:  os.Getenv(KeyPath),
  }
}
```

Listing 5.4: Setting up GitHub configuration parameters.

Each parameter is necessary to communicate with GitHub API. We add a new *valid()* method that can be used to confirm that all the parameters are set.

```
func (conf *GithubConfig) valid() bool {
  return conf.appID != "" && conf.keyPath != "" &&
    conf.clientID != "" && conf.secret != ""
}
```

Listing 5.5: Verifying that the configuration parameters are not empty.

To perform actions and modify the content of a course organization on GitHub, we need to create an installation client for each course.

```
// Creates a new scm client with access to the course organization.
func (sm *SCMMaker) NewInstallationClient(ctx context.Context, courseOrg string) (*github.Client, error) {
  resp, err := sm.githubConfig.appConfig.Client().Get(InstallationAPI)
  if err != nil {
    return nil, fmt.Errorf("error fetching installations for GitHub app %s: %w", sm.githubConfig.appID, err)
  }
  defer resp.Body.Close()
  body, err := io.ReadAll(resp.Body)
  if err != nil {
    return nil, fmt.Errorf("error reading installation response: %w", err)
  }
  var installations []*github.Installation
  if err := json.Unmarshal(body, &installations); err != nil {
    return nil, fmt.Errorf("error unmarshalling installation response: %w", err)
  }
  var installationID int64
  for _, inst := range installations {
    if *inst.Account.Login == courseOrg {
      installationID = *inst.ID
      break
    }
  }
  if installationID == 0 {
    return nil, fmt.Errorf("cannot find GitHub app installation for organization %s", courseOrg)
  }
  install, err := sm.githubConfig.appConfig.InstallationConfig(strconv.Itoa(int(installationID)))
  if err != nil {
    return nil, fmt.Errorf("error configuring github client for installation: %w", err)
  }
  return github.NewClient(install.Client(ctx)), nil
}
```

Listing 5.6: Creating a new installation client for a course organization.

As shown in listing 5.6, to create a new installation client GitHub app takes the organization name, retrieves a list of all active installations, and, if the app is already installed, creates and returns a new client.

Because all the current scm methods are designed for a token-based client, we also update these methods to use the new client type. Finally, we update the related tests and add a new set of tests to cover the new SCMMaker methods.

The final QuickFeed architecture based on a GitHub app is illustrated in figure 5.4.

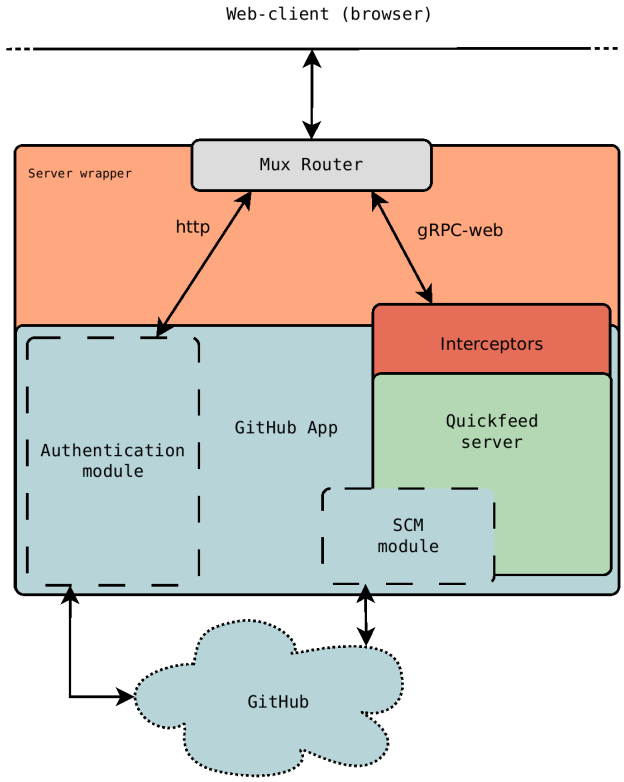

Figure 5.4: QuickFeed architecture with GitHub app integration.

Now both components of the QuickFeed architecture that rely on communication with GitHub are assisted with the QuickFeed's GitHub app. As a result, both **auth** and **scm** are now closely connected and easier to supervise and maintain.

Chapter 6

Supplementary architecture changes

In this chapter, we explain other changes introduced to the QuickFeed architecture by this thesis.

6.1 Proxyless gRPC-web

To enable gRPC requests from a browser client the original Autograder relied on an external Envoy proxy. Proxy separated gRPC and http requests and redirected these to the predefined ports.

Envoy was configured with two separate clusters to which it redirected http and gRPC-web requests.

Correspondingly, Autograder runs two separate servers. An http webserver was serving static files and handling authentication API calls. A gRPC backend service served business logic gRPC API endpoints.

The client-proxy-server architecture of the original Autograder is outlined in figure 6.1.

Envoy runs as a separate process and thus needs to be started manually or with a script each time the QuickFeed server starts.

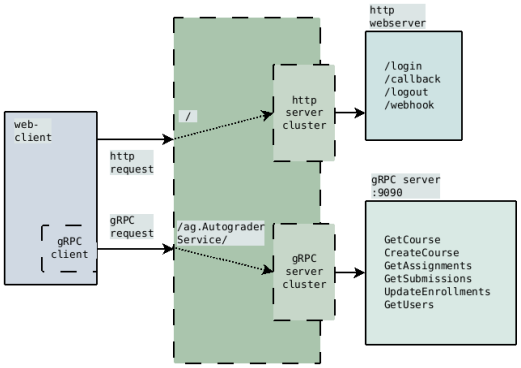

Figure 6.1: Quickfeed architecture with Envoy proxy.

Envoy reads configuration from a YAML file. However, this configuration must be changed depending on whether the server runs in production or locally. This has always been a challenging step in the development of the server for each new student team that would start working on the project.

Moreover, when some configuration options are discontinued or replaced, the proxy stops working and it is difficult to debug the cause.

We replace the original gRPC-web solution based on Envoy proxy with the alternative server wrapper developed by Improbable Engineering [18]. The wrapper implements an http router that allows multiplexing gRPC and http requests on the same port. The updated architecture can be seen in figure 6.2.

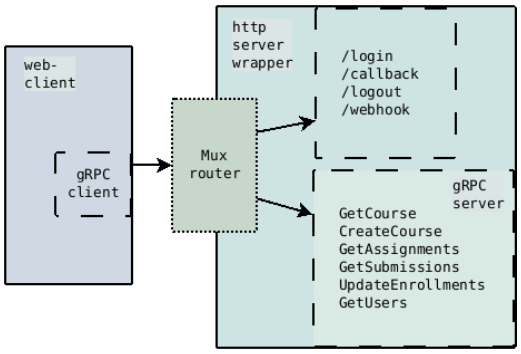

Figure 6.2: QuickFeed architecture with a multilplex server wrapper.

With the server wrapper, it is now possible to compile and run QuickFeed directly on a local machine without any additional steps. However, in production,

it might still be recommended to use a proxy for load balancing. However, there will be no need for configuration modifications in order to support gRPC-web requests.

6.2 Centralized configuration package

QuickFeed requires multiple parameters to configure and start the server. Some of them are passed as command-line options, some are stored in files, and some must be read from the environment.

In the original Autograder, constants with paths to the files, URL endpoints, and names of the expected environmental variables are scattered across the codebase. Keeping track of all of them is inconvenient, and we need to add several new parameters to support the updated architecture.

To simplify parameter management we introduce a new centralized **config** package. The new config will collect all the required parameters when the QuickFeed server starts and keep them for later use.

Config is a single structure that divides the stored parameters into three groups: endpoints, paths, and secrets. The definition of *Config* can be seen in listing 6.1.

```
type Config struct {
  Endpoints *Endpoints
  Secrets   *Secrets
  Paths     *Paths
}
```

Listing 6.1: Server configuration struct.

The *Endpoints* group is a collection of all http URL endpoints that QuickFeed uses to run the server and exchange information with GitHub. The collection can be seen in listing 6.2.

```
type Endpoints struct {
  BaseURL       string
  LoginURL      string
  CallbackURL   string
  LogoutURL     string
  WebhookURL    string
  InstallAppURL string
  Public        string
  PortNumber    string
}
```

Listing 6.2: Collection of URL endpoints.

The *Paths* group collects paths to the files with certificates and keys, as shown in listing 6.3.

```
type Paths struct {
  CertPath    string
  CertKeyPath string
  AppKeyPath  string
}
```

Listing 6.3: Collection of file paths.

The *Secrets* group is a collection of verification codes and keys. It can be seen in listing 6.4.

```
type Secrets struct {
  WebhookSecret  string
  CallbackSecret string
  TokenSecret    string
  key            *[32]byte
}
```

Listing 6.4: Collection of secrets.

All configuration parameters are read or generated by QuickFeed when the server starts. The struct is then kept as a part of the main service. Methods that consume different configuration details are generally defined as methods on the main service, which means they always have access to the necessary configuration parameters.

With centralized configuration, all the constants representing endpoints, paths, and environmental variables are no longer spread across multiple packages. As a result, it is easier to keep track, maintain and add new QuickFeed parameters.

Finally, some variables have to be used in several different packages, which would lead to multiple internal exports. With a separate configuration package, there is no need for imports: all server parameters are stored inside a single instance of the *Config* and can be easily used by any QuickFeed's business logic method.

6.3 Protofiles

The original Autograder has a single main protofile which contains all the gRPC methods along with every possible message type. Protofiles describe methods and types in Protocol Buffer, a language-neutral format. They are easy to read and are supposed to be self-documenting. However, with a long and unorganized file like the original **ag.proto** it is still challenging because of the amount of protobuf messages stored in one place.

We split the original protofile in three: message types are moved to the **types.proto** and **requests.proto**, while the API methods are left in the **ag.proto**.

The **types.proto** protofile contains message types that are saved directly to the database. Two examples of such messages are shown in listing 6.5.

```
message User {
    uint64 ID        = 1;
    bool isAdmin     = 2;
    string name      = 3;
    string studentID = 4;
    string email     = 5;
    string avatarURL = 6;
    string login     = 7;
    bool updateToken = 8;

    repeated RemoteIdentity remoteIdentities = 9;
    repeated Enrollment enrollments          = 10;
}

message Course {
    uint64 ID                       = 1;
    uint64 courseCreatorID          = 2;
    string name                     = 3;
    string code                     = 4 [(go.field) = { tags: 'gorm:"uniqueIndex:idx_unique_course"' }];
    uint32 year                     = 5 [(go.field) = { tags: 'gorm:"uniqueIndex:idx_unique_course"' }];
    string tag                      = 6;
    string provider                 = 7;
    uint64 organizationID           = 8;
    string organizationPath         = 9;  // The organization's SCM name, e.g. uis-dat520-2020.
    uint32 slipDays                 = 10;
    string dockerfile               = 11;
    int64 installationID            = 12;  // TODO(vera): we most probably don't need it anymore
    Enrollment.UserStatus enrolled = 13 [(go.field) = { tags: 'gorm:"-"' }];

    repeated Enrollment enrollments = 14;
    repeated Assignment assignments = 15;
    repeated Group groups           = 16;
}
```

Listing 6.5: Example of message types in types.proto.

The **requests.proto** file is a collection of messages that are only used as the payload in gRPC requests and responses. Two examples of such methods can be found in listing 6.6.

```
message GroupRequest {
    uint64 userID   = 1;
    uint64 groupID  = 2;
    uint64 courseID = 3;
}

message Status {
    uint64 Code  = 1;
    string Error = 2;
}
```

Listing 6.6: Messages used in API calls are kept in requests.proto.

With three smaller files, the gRPC API definitions are logically separated into three smaller groups that are easier to read and maintain.

6.4 Encrypting access tokens

A student enrolling in a new course will receive three invitations to the course repositories. The invitations will be sent to the e-mail address associated with the student's GitHub account. The invitations must be accepted before the student gets full access to the course content.

Originally students were expected to accept the invitation manually. This was a constant source of enrollment delays and extra work for teaching staff because invitations would be ignored, forgotten, and expired, or would land in the spam folder. Automated accepting of invitations was added to Autograder to simplify this process.

However, invitation management functionality is still not supported by GitHub apps. Therefore personal access tokens still have to be used for this service.

An access token is a string of random characters and numbers. In Autograder access tokens were stored in the database in plaintext. This is not recommended because of a potential security risk.

We propose to encrypt all access tokens before saving them to the database and only decrypt the necessary token directly before use. However, to preserve compatibility with older databases we add this functionality as an option. It is deactivated by default and can be enabled by a command-line flag shown in listing 6.7.

```
1    withEncryption = flag.Bool("e", false, "encrypt access tokens")
```

Listing 6.7: Command-line option to encrypt access tokens

The implemented encryption scheme involves two secret keys. One key is used to encode and decode access tokens. We will refer to it as the master key. Another key, a passphrase, is used to encrypt the master key.

The encrypted master key is stored in a file. If encryption is enabled, QuickFeed will read the path to the file from the environment, as shown in listing 6.8.

```
1  if *withEncryption {
2    if err := serverConfig.ReadKey(true); err != nil {
3      log.Fatal(err)
4    }
5  }
```

Listing 6.8: Reading encryption key only if encryption is enabled.

We leave it to the user to decide where the file will be stored. However, we strongly recommend placing it outside the QuickFeed's working directory so that

it is more difficult to discover.

When the encrypted master key has been collected from a file, it needs to be deciphered with the correct passphrase. We add two alternatives to how the passphrase can be obtained by the QuickFeed: from the environment or by asking for user input. The *ReadKey* method is presented in listing 6.9.

```
func (c *Config) ReadKey(fromEnv bool) error {
  var pass string
  if fromEnv {
    pass = os.Getenv("KEYPASS")
  }
  var inputBytes []byte
  if pass == "" {
    fmt.Println("Key: ")
    input, err := term.ReadPassword(int(os.Stdin.Fd()))
    if err != nil {
      return err
    }
    inputBytes = input
  } else {
    inputBytes = []byte(pass)
  }
```

Listing 6.9: Reading the passphrase key from the environment or a user input.

The deciphered master key is stored in memory as an unexported (private) field in the configuration. This way the key cannot be accessed from any other module of the application following the Encapsulation principle.

We add *WithEncryption* helper method for modules that make use of access token encryption. The method controls whether the encryption option has been enabled or not and is shown in listing 6.10.

```
func (c *Config) WithEncryption() bool {
  return c.Secrets.key != nil
}
```

Listing 6.10: Helper method that indicates whether encryption is enabled.

Listing 6.11 presents the *Cipher* method used to encrypt personal access tokens. To encrypt an access token we need to generate a random 24-byte nonce.

```
func (c *Config) Cipher(token string) (string, error) {
  nonce := new([24]byte)
  _, err := io.ReadFull(rand.Reader, nonce[:])
  if err != nil {
    return "", fmt.Errorf("failed to geerate nonce: %w", err)
  }
  out := make([]byte, 24)
  copy(out, nonce[:])
  out = secretbox.Seal(out, []byte(token), nonce, c.Secrets.key)
  return base64.RawStdEncoding.EncodeToString(out), nil
}
```

Listing 6.11: Encrypting a string.

If encryption is enabled, tokens will be encoded during the initial user authentication with the GitHub account, as can be seen in listing 6.12.

```
accessToken := githubToken.AccessToken
if config.WithEncryption() {
  accessToken, err = config.Cipher(accessToken)
  if err != nil {
    unauthorized(logger, w, callback, "failed to encrypt access token: %v", err)
    return
  }
}
```

Listing 6.12: Encrypting access tokens during authentication.

We use the *secretbox* Go package to encrypt the master key and access tokens [19]. It is simple, secure, and designed specifically for encryption of shorter messages which makes it a good choice for QuickFeed.

Finally, we supply a short script that generates two cryptographically secure random 32-byte keys with the *crypto/rand* Go library, encrypts the master key, and writes it to a file. The script prints out the base64 encoded passphrase to the standard output. Therefore it is only recommended to run it on a secure personal device and never on the production server.

6.5 Dependencies

There are several dependencies in the Autograder's authentication module that we want to remove. Relying on third-party dependencies is always a risk. There is no guarantee that the packages will be kept up to date or compatible with other dependencies the project might rely upon. There is also a danger of some potential vulnerability in the framework that will not be noticed and patched in time. This is why we propose to limit the dependencies to those that are strictly necessary for the functionality of the application. The rest can be either removed or replaced with standard Go libraries.

The *echo* package is a web framework that was used in the original Autograder to organize and manage API endpoints of the http-based REST API [20]. The general API has since then been replaced by gRPC calls. However, the Autograder server still has to maintain four http endpoints for user authentication because GitHub does not support gRPC.

Echo was convenient to use in the original Autograder architecture to group many API methods together and run the http middleware. However, importing a dedicated framework now that it will be used by a very few API endpoints seems

excessive. Furthermore, we replace both middleware pieces with gRPC interceptors as explained in chapter 4.

Go offers the *net/http* package which provides all the basic functionality for http/https communication [21]. We remove the *echo* dependency and use basic http handler functions instead.

Another dependency we propose to remove is the *gothic* library which is essentially a simplified wrapper for a more comprehensive *Goth* package [22]. Both packages are intended for user authentication with third-party identity providers like GitHub or GitLab.

Gothic is convenient to use in combination with sessions. However, most of its functionality in charge of OAuth 2.0 authentication spec is based on another standard Go library, *oauth2* [23]. Now that we replaced sessions with JSON Web Tokens, there are no benefits to importing the *gothic* package. Instead we use the *oauth2* library directly.

Finally, we completely remove the *Gorilla sessions* package dependency. Gorilla offers a session store infrastructure that is no longer necessary.

```
    func OAuth2Login(logger *zap.SugaredLogger, db database.Database) echo.HandlerFunc {
  return func(c echo.Context) error {
    w := c.Response()
    r := c.Request()
    provider, err := gothic.GetProviderName(r)
    if err != nil {
      logger.Error(err.Error())
      return echo.NewHTTPError(http.StatusBadRequest, err.Error())
    }
    var teacher int
    if strings.HasSuffix(provider, TeacherSuffix) {
      teacher = 1
    }
    qv := r.URL.Query()
    logger.Debugf("qv: %v", qv)
    redirect := extractRedirectURL(r, Redirect)
    logger.Debugf("redirect: %v", redirect)
    qv.Set(State, strconv.Itoa(teacher)+redirect)
    logger.Debugf("State: %v", strconv.Itoa(teacher)+redirect)
    r.URL.RawQuery = qv.Encode()
    logger.Debugf("RawQuery: %v", r.URL.RawQuery)

    url, err := gothic.GetAuthURL(w, r)
    if err != nil {
      logger.Error(err.Error())
      return echo.NewHTTPError(http.StatusBadRequest, err.Error())
    }
    logger.Debugf("Redirecting to %s to perform authentication; AuthURL: %v", provider, url)
    return c.Redirect(http.StatusTemporaryRedirect, url)
  }
}
```

Listing 6.13: The original user authentication handler.

Without excessive dependencies, authentication methods become shorter and

cleaner. Listings 6.13 and 6.14 compare the original and updated initial authentication handlers.

The purpose of the *OAuth2Login* handler method is a simple redirect to the identity provider's log in page.

The original handler method shown in listing 6.13 uses the *gothic* package to extract the correct provider from a query parameter and later to generate a correct redirect URL. The *echo* package serves as a simple wrapper around the original *net/http* methods without offering any additional utility. Multiple log messages are produced to keep track of all the details.

```
func OAuth2Login(logger *zap.SugaredLogger, db database.Database, config oauth2.Config, secret string)
http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    if r.Method != "GET" {
      unauthorized(logger, w, login, "request method: %s", r.Method)
      return
    }
        // URL format: "/auth/login/login/:provider"
    provider := strings.Split(r.URL.Path, "/")[3]
    redirectURL := config.AuthCodeURL(secret)
    logger.Debugf("Redirecting to %s to perform authentication; AuthURL: %v", provider, redirectURL)
    http.Redirect(w, r, redirectURL, http.StatusTemporaryRedirect)
  }
}
```

Listing 6.14: The reworked authentication handler.

The reworked method can be found in listing 6.14. There is no need for an explicit extraction of the correct name of the identity provider unless it is required for logging. The *oauth2.Config* structure contains all the provider-specific parameters and will generate a redirect URL for any configured provider.

As a result, most of the authentication handlers have become shorter and easier to read, understand and maintain.

Chapter 7

Discussion

In this chapter, we discuss the results of the changes introduced to the authentication and authorization architecture of QuickFeed by the thesis.

7.1 Token based authentication

We have replaced the stateful session-based authentication pattern used in the original Autograder with stateless JSON Web Tokens (JWTs). Using JWTs offers several advantages, such as easier interoperability between the http and gRPC APIs of QuickFeed, better scalability, and cleaner architecture due to the removal of redundant structures and methods.

7.1.1 Replacing sessions and http middleware

Figures 7.1 and 7.2 demonstrate the QuickFeed authentication flow with and without sessions and middleware.

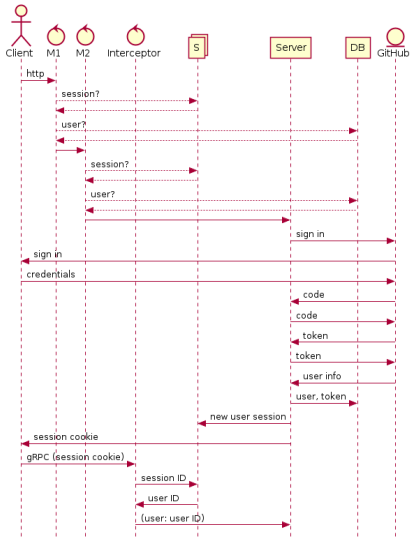

Figure 7.1: User login sequence with http middleware and sessions.

With middleware (M1 and M2) shown in figure 7.1, each middleware method queries the session S store and the database DB before the user is redirected to the log in page of the identity provider (here GitHub). When a signed in user calls gRPC API, the session store is requested again to retrieve the ID of the user.

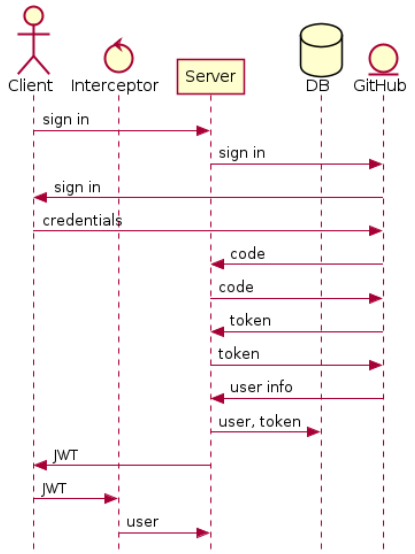

Figure 7.2: User login sequence with tokens and interceptors.

With the token-based pattern shown in figure 7.2, the user is redirected to the identity provider page immediately. The JWT is then set as a secure cookie and used by the authentication interceptor directly.

The resulting authentication architecture is cleaner and more consistent now that the redundant middleware pieces and session stores are removed. The growing userbase no longer affects the application's resource consumption because there is no user state to be stored in memory. Furthermore, a JWT-based solution offers better support for horizontal scaling, as the same JWT token can be passed across multiple instances of the application as well as different services.

7.1.2 Token size

One of the known disadvantages of JWTs is that a JWT cookie can be significantly larger than a session cookie. JWT contains multiple required fields and, in addition, custom claims with user roles. In comparison, a session cookie only needs to include a session identifier. An authentication cookie must be sent with each request to the server. As a result, a larger cookie can lead to increased network overhead.

However, an average Autograder's session cookie is almost one-third bigger than an average JWT cookie. This can be explained by the fact that QuickFeed's user claims are relatively small, with the total JWT string averaging about 238 characters, while a session ID has a fixed size of 360 characters.

7.1.3 JWT security

JWTs are signed and base64 encoded. Quickfeed verifies the signature before the request is allowed to the server API. This guarantees that the payload of a JWT has not been modified.

However, it is still possible to read the content of the encoded payload by decoding the base64 string in the JWT cookie. If it is desirable to keep the payload secret it is possible to update the token manager with an option to encrypt the JWT before sending it to the client.

Moreover, a JWT is stored in the browser cookie. Even though the cookie itself is secure and cannot be accessed by a script, nothing prevents another user from simply copy-pasting the cookie and using it from another machine to impersonate the original user.

We cannot prevent physical access by others to the web clients of QuickFeed users and must trust the users to do so. However, it could be useful to add a client-side interceptor that would add some information about the currently logged in user to each request. This information could then be compared by the server with JWT claims in the same request. This would add a verification check to ensure that the user in the web-client state and the user in the JWT claims is in fact the same user.

7.1.4 Updating tokens

In a stateless authentication pattern, the server has no information about the already issued JWTs. Access control needs to rely on the information about user roles stored in the JWT that is sent along with a request from a client.

However, the role of a user can change. A user can be promoted to admin, enroll in a new course, or promoted from a student to a teaching assistant. Such a change means that the user must be able to access new categories of information. However, the JWT stored in the user's browser can have outdated information about the role of the user and, as a result, access to the information can be incorrectly restricted.

To avoid this complication we create a list with IDs of all users who will need new claims next time they send a request to the QuickFeed server. It might seem like a stateful solution contradicting the stateless design of the token-based authentication pattern.

However, changes to user roles in the QuickFeed system are very uncommon. Only one or two users a year are promoted to admins. Demotions are almost non-existent. A lot of students get enrolled in new courses twice a year, but these events are limited to a very few weeks at the start of a new semester. Most of the time the list will be empty.

7.1.5 Alternative solutions

We have considered an alternative to JWT authentication tokens, macaroons [24]. Macaroon is a new token format that is more abstract and flexible compared to JWTs. There are no required fields or predefined encoding algorithms. The main feature of macaroons is caveats: multiple restrictions that can be added to the original token to limit its scopes. Adding a new caveat creates a new token that

can be sent with a specific API call, or delegated to another service or even a new user.

However, we found macaroons less suitable for our goals. First of all, the library ecosystem that supports macaroons is scarce. JWTs, on the opposite, are widely supported.

Furthermore, the purpose of macaroons seems to be aimed at systems where clients have to produce new tokens with different restrictions without contacting the server that has issued the original token. QuickFeed's authentication architecture limits the management of the tokens to the dedicated module of the server. Token modification by the user clients is undesirable.

Finally, as caveats can be added in any number and order, it is difficult to design comprehensive authentication checks. We pre-define all the fields that can be present in the user claims of a JWT. As a result, authentication and authorization checks are simple and straightforward as there is never an unexpected field.

7.2 GitHub integration: access tokens

GitHub creates personal access tokens when users authenticate to QuickFeed with their GitHub accounts. In the original Autograder, access tokens were saved in the database and used to create GitHub API clients. Clients were used for automation of the interactions with course organizations on GitHub such as creating course repositories, adding students to a course, promoting teaching assistants, and creating student groups. Clients based on access tokens are the only option when using an OAuth app to integrate with GitHub as the original Autograder did. Back when the application was designed OAuth apps were the only option.

We have replaced the Autograder's OAuth app with a more modern integration solution, a GitHub app. A GitHub app can authenticate students with their GitHub accounts just like OAuth apps do. However, a GitHub app does not use personal access tokens to access course organizations on GitHub.

Unfortunately, GitHub apps do not support automated accepting of repository invitations which is an important feature in QuickFeed. As a result, it is still necessary to rely on personal access tokens.

In Autograder access tokens were saved in the database as plaintext strings. We added an option to encrypt access tokens when saving, and only decrypt di-

rectly before use. Storing encrypted tokens is safer even if the database file is compromised. However, not storing access tokens at all is the best solution.

GitHub continues to expand the number of API endpoints that can be accessed by a GitHub app. Hopefully, support for accepting repository invitations will also be added in the future. In this case, we recommend immediately stopping the practice of storing access tokens in the database and transferring this functionality to the QuickFeed's GitHub app.

7.3 Proxyless gRPC-web

The original Autograder's client-server architecture required a proxy in order to support gRPC calls from browser clients. Envoy proxy had to be started as a separate process and required a complex configuration. Additionally, the configuration needed to be changed depending on whether the Autograder server was running in production or on a local machine for development and testing. We changed the architecture such that there is no need for a proxy in the middle.

However, it is possible that using a proxy can be beneficial for the QuickFeed's deployment setup. The changes we made to the architecture still allow using a proxy to perform load balancing or redirects. However, a configuration would be much simpler compared to the original Autograder as there is no need to detect and separate gRPC calls. Both http and gRPC requests can now be sent directly to the QuickFeed server.

This way it is easier for new teams that start working on the project to run the server on their own machines without being slowed down by learning the complex proxy configuration.

7.4 Access control

Authentication checks in the original Autograder were performed inside the API methods. To understand what role a user should have to call the method it was necessary to find the method's definition and read the note describing the access restrictions in the comments.

We redesigned this authorization mechanism as a single access control interceptor module. The interceptor inspects requests before deciding whether they

can be allowed to proceed to the QuickFeed server.

We collect all the restricted API methods and required roles such as admin, course teacher, course student, or group member in a single table. This simplifies development and maintaining of the access control module as it is now easy to find out the required roles for each method.

In most cases, we can rely on the information about the role of a user from the JWT claims. This way it is no longer necessary to query the database to confirm the role before granting access to the server.

Reading JWT claims provides information about the admin role and whether the user is a student or teacher in each course. However, some methods can only be called by a course student who is also a member of a certain student group. There is no information about groups in the QuickFeed's JWT claims. As a result, it is necessary to fetch this information from the database.

It might be practical to also include the group information in the claims. However, this would cause an increase in the number of JWT updates. Additionally, this would mean that the interceptor module responsible for maintaining a list of outdated tokens would need to inspect more methods.

7.5 Signing JWTs

We sign JWTs to guarantee the integrity of the content. We use HMAC for signing which relies on a cryptographic hash function.

The choice of the hash function signing can be difficult. To decide between SHA-256 and SHA-512 there are several things to consider. Both offer a comparable level of security. However, the performance can differ [25].

SHA-256 is generally faster, however, it can still be outperformed by SHA-512 when hashing longer strings. SHA-512 is also expected to be faster on 64-bit platforms.

Finally, SHA-512 produces a longer output which can be important for QuickFeed.

Signed JWTs are sent over the network with each request. As a result, the possible network overhead of sending a longer string must also be considered.

In the end, we chose SHA-256 to sign JWTs with because it is secure, widely supported, and results in lower network overhead compared to SHA-512.

Our tests demonstrate that both hashing algorithms show very close performance. Signing with SHA-256 averages 89.92 μs, compared to SHA-512 that on average was taking 90.52 μs.

Chapter 8

Conclusion

This thesis presents a redesign of the authentication and authorization processes in the QuickFeed software project.

We designed a set of solutions to improve the security of the application. We took away some unnecessary dependencies on third-party libraries. Moreover, we added missing verification checks when communicating with GitHub to authenticate a user. Finally, we replaced session-based internal authentication with a stateless token-based process.

JSON Web Tokens (JWTs) are now used to carry the identity of a user internally to ensure interoperability between the http and gRPC APIs of QuickFeed. JWTs are signed on creation and verified on each client request to ensure the integrity of the content. The payload of a JWT contains user claims that must be trusted by the centralized access control module. To guarantee that information in the claims is always up to date, JWTs have a short lifetime of 15 minutes. Claims are updated from the database when a JWT is about to expire or if the role of a user in the Quickfeed system has changed.

We modernized the way QuickFeed integrates with GitHub. We also added an option to encrypt personal access tokens issued by GitHub.

Changes introduced by this thesis also provide support for better application scalability. Stateless token-based authentication allows using the same JWT to establish the identity of a user across multiple services. Due to the stateless authentication pattern, the server no longer keeps any structures with user state that would grow in size in proportion to the number of authenticated users.

Finally, we reorganized the authentication and authorization architecture by

removing an external proxy between the QuickFeed's server and clients. We also took away redundant structures and methods and gathered all the user authorization utilities in a single place. A dedicated access control table keeps track of all restricted API methods and user roles. It is now easier to understand, maintain and update the modules in charge of user authentication and authorization.

Appendix A

Source code

The source code developed in this thesis work can be found in the **auth-rework** brach of the QuickFeed project at https://github.com/quickfeed/quickfeed/tree/auth-rework.

The script that generates a master key and passphrase can be found at https://github.com/0xf8f8ff/keytool.

Bibliography

[1] Quickfeed project. https://github.com/quickfeed/quickfeed. (Accessed: 06.06.2022).

[2] Single sign-on. https://auth0.com/docs/authenticate/single-sign-on. (Accessed: 14.06.2022).

[3] Saml specifications. http://saml.xml.org/saml-specifications. (Accessed: 14.06.2022).

[4] Oauth 2.0. https://oauth.net/2/. (Accessed: 14.06.2022).

[5] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL https://www.rfc-editor.org/info/rfc6749.

[6] Open id connect protocol. https://auth0.com/docs/authenticate/protocols/openid-connect-protocol. (Accessed: 14.06.2022).

[7] J. Bradley M. Jones and N. Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL https://www.rfc-editor.org/info/rfc7519.

[8] W. Stallings. *Cryptography and network security. Principles and practice*. Pearson Education Limited, 2017. ISBN 9781292158587.

[9] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997. URL https://www.rfc-editor.org/info/rfc2104.

[10] Using http cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies. (Accessed: 06.06.2022).

[11] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The dos and don'ts of client authentication on the web. https:

//www.usenix.org/conference/10th-usenix-security-symposium/dos-and-donts-client-authentication-web, 2001. (Accessed: 06.06.2022).

[12] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634, August 2006. URL https://www.rfc-editor.org/info/rfc4634.

[13] PhD Svetlin Nakov. *Practical cryptography for developers*. 2018. ISBN 9786190008705.

[14] Critical vulnerabilities in json web token libraries. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries. (Accessed: 06.06.2022).

[15] Interceptors in grpc-web. https://grpc.io/blog/grpc-web-interceptor. (Accessed: 06.06.2022).

[16] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015. URL https://www.rfc-editor.org/info/rfc7515.

[17] Differences between github apps and oauth apps. https://docs.github.com/en/developers/apps/getting-started-with-apps/differences-between-github-apps-and-oauth-apps. (Accessed: 06.06.2022).

[18] Improbable Engineering. Grpc-web. https://github.com/improbable-eng/grpc-web/tree/master/go/grpcweb. (Accessed: 05.06.2022).

[19] Secretbox. https://pkg.go.dev/golang.org/x/crypto/nacl/secretbox. (Accessed: 06.06.2022).

[20] Echo http framework. https://echo.labstack.com. (Accessed: 06.06.2022).

[21] Go net/http standard library. https://pkg.go.dev/net/http. (Accessed: 06.06.2022).

[22] Gothic package. https://pkg.go.dev/github.com/markbates/goth/gothic. (Accessed: 06.06.2022).

[23] Oauth2 package. https://pkg.go.dev/golang.org/x/oauth2. (Accessed: 06.06.2022).

[24] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[25] L. Latinov. Md5, sha-1, sha-256 and sha-512 speed performance. https://automationrhapsody.com/md5-sha-1-sha-256-sha-512-speed-performance/.

University of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: drmakete lab on Unsplash

© 2022 Vera Yaseneva